Materials for CSC 372, Spring 2016
The University of Arizona
William H. Mitchell (whm)

This PDF is a collection of almost all formal instructional materials for CSC 372, Spring 2016.

Not included:

- Solutions for assignments
- Piazza posts

Class was held TH 14:00-15:15 in BIOW 208. There were 30 class meetings, including the mid-term, from Thursday, Jan 14 through Tuesday, May 5. Ben Gaska and Patrick Hickey were undergraduate teaching assistants.

65 students did Assignment 1, a survey. 64 were on the final grade roster. 56 students took the final exam.

Grade distribution:

A	27
B	13
C	7
D	9
E	3

TCE "Overall rating of the course" was 4.81 out of 5.00. (62.71% Response)

This container comprises these files:

all-materials-cover.pdf	This document
syllabus.pdf	Syllabus
intro.pdf	Introductory slides (~.7 lectures)
haskell.pdf	Haskell slides (~9.3 lectures)
ruby.pdf	Ruby slides (~9 lectures)
prolog.pdf	Prolog slides (~8.5 lectures)
a1.pdf through a10.pdf and av.pdf	Assignments
exam-archive/372.s16-quizzes.pdf	All quizzes and solutions
exam[12]{,sol}.pdf	Exams and solutions
tester.pdf	"Using the Tester"
icon.pdf	A small set of slides on Icon, used in conjunction with an "Icon by Observation" exercise for the last class.
unixstuff.pdf	"UNIX Stuff for 372", a recommended supplementary document but not part of the course materials.

CSC 372: Comparative Programming Languages
The University of Arizona
Spring 2016

Instructor

William H. Mitchell (whm)
Office hours and contact info: *Posted on Piazza*

Schedule

Lectures: Tuesdays and Thursdays 14:00-15:15 in BIOW 208

Teaching Assistants

Undergraduate TAs:
Ben Gaska (bengaska)
Patrick Hickey (patrickhickey)

Office hours and contact info: *Posted on Piazza*

Website and handouts

We'll be using Piazza for announcements, discussions, and more. **If you haven't already signed up, do it now!** Go to piazza.com and follow their instructions.

All handouts will be posted on Piazza, on the Resources tab of the Resources page. All materials can also be directly accessed here: http://cs.arizona.edu/classes/cs372/spring16

Leftover handouts will be placed on the metal bookshelves near the freight elevator at the west end of the eighth floor of Gould-Simpson.

If you don't make use of paper handouts, let me know, and I'll reduce the copy count accordingly.

Prerequisites

CSC 127B or CSC 227. CSC major status.

Knowledge of Java, as covered in CSC 127A/B (or CSC 227), will be assumed.

The prerequisites are very modest but this is a 300-level computer science class with a lot of diverse content. As an analogy, imagine a 300-level math class with only high-school algebra as a prerequisite but that will venture into some exotic math that's not much like algebra at all. My task is to respect the prerequisites but also cover a body of material that's appropriate for a 300-level CS class.

Course Objectives

The purpose of this course is to explore some alternative ways of specifying computation and to help you understand and harness the forces that a programming language can exert. We'll spend a lot of time working with three languages: Haskell, Ruby, and Prolog. Functional programming will be studied using Haskell. Ruby will be used to explore imperative and object-oriented programming using a language with dynamic type checking. Prolog will transport us into the very different world of logic programming. You'll learn about some interesting elements of other languages.

Upon successfully completing the course you'll be around a "2" on a 1-5 scale (5 high) with Haskell, Ruby, and Prolog. You will understand the characteristics of the programming paradigms supported by those languages and be able to apply some of the techniques in other languages. You'll have an increased ability to learn new languages. You'll have some idea about whether you want to pursue further study of programming languages.

Textbooks

No textbooks are required.

It is my intention that the lectures, handouts, and Piazza postings will provide all the information needed to successfully complete the course; but it's often good to see things explained in multiple ways. Below are some books that I recommend as good supplements but it's important to understand that the routes we'll be taking through the languages don't directly correspond to any book; the handouts you'll be getting are the primary "text".

Haskell:

I consider *Learn You a Haskell for Great Good!* by Miran Lipovača to be an excellent Haskell book and the best beginner's Haskell book on Safari.

I think of *Programming in Haskell*, by Hutton and on Safari, to be a good book once you've got the basics down, but I think it moves too fast to stand alone as a first book on Haskell, unless one has had prior exposure to functional programming.

Real World Haskell, by O'Sullivan, Stewart, and Goerzen, has some good introductory material as well as some interesting real-world examples. It's on Safari.

The Haskell School of Expression, by Paul Hudak, was one of the first Haskell books I owned. I didn't find it very helpful at the time but I've since grown to appreciate it. It's on Safari.

If you don't mind spending some money, then I recommend *Haskell: The Craft of Functional Programming*, 3rd edition, by Simon Thompson. It's very thorough; it doesn't make the sort of leaps that can leave beginners puzzled.

Ruby:

Safari has numerous Ruby books but it doesn't have the Ruby book I currently like the best, which is this:
Programming Ruby 1.9 & 2.0 (4th edition): The Pragmatic Programmers' Guide
http://pragprog.com/book/ruby4/programming-ruby-1-9-2-0

The Ruby book on Safari that I currently like the best is *The Ruby Programming Language* by David Flannagan and Yukihiro Matsumoto.

Prolog:

Safari has zero Prolog titles but an excellent text, *Programming in Prolog*, 5th edition, by Clocksin and Mellish, is available as a PDF that can be downloaded through the library. **Get a copy NOW**, in case licensing agreements change during the semester. Here's a link:
http://ezproxy.library.arizona.edu/login?url=http://dx.doi.org/10.1007/978-3-642-55481-0

Another Prolog book I really like is freely available on the net, albeit as pages scanned as images:
Prolog Programming in Depth, by Covington, Nute, and Vellino
http://www.covingtoninnovations.com/books/PPID.pdf

I'll make available on Piazza an OCR'd version of Covington's book with a hidden text layer added, so it can be searched.

Note that Safari can be accessed off-campus using the VPN or by using the library's proxy. Here's the proxy-based URL for Safari: http://proquest.safaribooksonline.com.ezproxy1.library.arizona.edu/ Hit it and then click "Start Using Safari" under "Academic License & Public Library Users".

Grading Structure

My goal is for **everyone** to earn an "A" in this course.

The final grade will comprise the following:

Assignments	60%
Pop quizzes	5%
Mid-term exam	13%
Final exam	22%

Final grades will be based on a ten-point scale: 90 or better is a guaranteed A, 80 or better is a B, and so forth. The lower bounds may be adjusted downwards to accommodate clustering of grades and/or other factors.

It is my goal that you will need to spend, on average, no more than ten hours per week, counting lectures, to learn the course material and get an "A" in this course. If you find that you're needing to spend more time than that, let's talk about it.

You are strongly encouraged to contest any assigned score that you feel is not fair.

There is no attendance component in the grade—if you find that my lectures aren't worth your time, feel free to cut class!

The mid-term exam is tentatively scheduled for Thursday, March 10, the last class before Spring Break. It will cover all the Haskell material and some amount of Ruby.

My reading of http://www.registrar.arizona.edu/schedule2161/exams/tr.htm indicates that the final exam will be Monday, May 9, 2016, 3:30-5:30 pm, in our regular classroom. The final exam will be comprehensive but with more emphasis on Ruby and Prolog than Haskell.

My goal is to distribute language-specific points as evenly as possible between the three languages, across both assignments and exams.

Assignments

Assignments will largely consist of programming problems but other types of problems, such as short answer questions, essay questions, and diagrams, may appear as well. There will be a video project, too. As I write this I anticipate that there will be seven major assignments, one every two weeks, and some number of minor assignments but those counts and timings are subject to change. There will be a total of 600 points worth of assignments. Based on 600 total assignment points, a ten point homework problem corresponds to one point on your final average.

For programming problems great emphasis will be placed on the ability to deliver code whose output exactly matches the specification. Failure to achieve that will typically result in large point deductions, sometimes the full value of the problem.

My view is that it's a Bad Thing to give any credit for code that doesn't work. **Programs that don't compile or that mostly don't work will earn a grade of zero, regardless of how close to working they might be.** Additionally, non-general solutions, which might have the expected output "wired-in", will very likely earn a grade of zero.

Unless specifically requested in an assignment write-up, no comments are required in any code. Use comments as you see fit.

My view is that programming assignments are to help you learn the course material—I don't view an assignment as a take-home exam. As a rule, you'll learn more if you can get through an assignment without asking for a lot of help; but if you reach a point where you simply aren't making progress and you're running out of ideas or time, then you surely should ask for help.

Each assignment will specify a precise due date and time. **As a rule, late assignment submissions are not accepted and result in a grade of zero. There are no late days.**

Extensions may be granted to the class as a whole if problems arise with an assignment or with departmental or university computing facilities.

Extensions for individuals may be granted in certain cases. **The key factor that leads to an extension is that due to circumstances beyond the student's control, the student's work is, was, or will be impeded, and it is impractical for the student to make up for the lost time.**

Accident, illness, friend/family crises, and significantly disruptive failures of technology are examples of circumstances that I generally consider to be beyond a student's control. On the other hand, for example, an extension due to assignments or exams in other classes is extremely unlikely. Travel, such as an interviewing trip, may merit an extension, but pre-trip discussion and approval of the extension is strongly recommended. Unexpected hours at a job, such as needing to fill in for a sick co-worker, may warrant an extension. **Ultimately, however, each situation is unique; you are strongly encouraged to contact me if you believe an extension may be warranted.** If you believe an extension is warranted, DO NOT work on an assignment (or even think about it) past the deadline; wait for an extension to be granted.

Extensions are granted in the form of an amount of time, such as eight hours. An eight-hour extension can be pictured as a count-down timer with an initial setting of eight hours. The timer runs whenever you are working on the assignment, whether that be typing in code or simply thinking about it.

Here's a scenario involving an extension:

> Your new laptop catches fire and burns itself into a cinder eight hours before midnight deadline on a Wednesday. I would likely grant you an eight-hour extension. On Friday, your next chance to get to the store, you get a new laptop and you're operational by Friday night. You might spend four hours Friday night working on the assignment, take off all day on Saturday and Sunday, spend two more hours on Monday and get absolutely stuck, and finish it up after a couple of questions during office hours on Wednesday.

You will be on your honor to keep written track of the time spent during an extension and not exceed the amount granted.

All holidays or special events observed by organized religions will be honored for those students who show affiliation with that particular religion. Absences pre-approved by the UA Dean of Students (or Dean's designee) will be honored.

Bug Bounties

A "bug bounty" of one assignment point of extra credit will be awarded to the first student to report a particular bug in an assignment. Bugs might take the form of errors in examples, ambiguous statements, incomplete specifications, etc. As a rule, simple misspellings and minor typographical errors won't qualify for a bug bounty point; but each situation is unique—you are encouraged to report any bugs you find. Any number of bug bounty points may be earned for an assignment and will be added to the grade for that assignment.

Bug bounty points may also awarded for bugs in the slides, my Piazza postings, quizzes, exams, and this syllabus. Such points are added to the next assignment.

Please report bugs by mail or IM, not via Piazza, to give me a chance to filter any false positives.

Don't interrupt lectures to point out minor bugs on slides but do speak up if an error seems serious. My usual practice is to only put code and commands onto a slide with copy/paste after testing it first, but if you see an error in code or commands, speak up.

Quizzes

There will be some number of "pop" quizzes. Quizzes will typically be a handful of questions, be allocated three minutes or less, and be worth 1-5 quiz points. There will be a total of 50 quiz points, corresponding to 5% of the final grade. Quizzes may be conducted at any time during the class period. In some cases a quiz may be given simply to see what portion of the class grasped some just-presented material and be graded on the basis of participation rather than correctness.

Computing Facilities

You're free to develop solutions for programming problems on any machine you wish but your solutions will be graded on the

CS machine named "lectura", so it's important to test your solutions on lectura before you submit them for grading.

By virtue of being enrolled in this class you should already have a CS computing account with the same name as your UA NetID but a password that's likely different (a good practice). With that account you can login on any of the CS instructional machines, including lectura. The files in your home directory tree are stored on a server and appear the same no matter which CS machine you're logged into.

My UNIX slides from 352 in Fall 2015 have details about logging into lectura, resetting your password, and more. Those slides are here: http://www.cs.arizona.edu/classes/cs352/fall15/unix.pdf. See slides 14-21 for details about logging in. Slides 74-102 discuss options for editing files on lectura and tools like WinSCP and DropSync, to keep directories on lectura synchronized with corresponding directories your laptop.

The CS computing facilities are described in http://cs.arizona.edu/computing/facilities. The FAQs at http://faq.cs.arizona.edu have lots of answers.

Office Hours

I truly enjoy working with students. I believe that interaction with students outside the classroom is a vital aspect of teaching a course. I will do everything possible to make myself accessible to you.

I prefer to conduct office hours in a group-style, round-robin manner. You needn't wait in the hallway if I'm working with another student; you may join us and listen in if you so desire. If several persons each have questions, I will handle one question at a time from each person in turn. I will often give priority to short questions (i.e., questions with short answers) and to persons having other commitments that constrain their waiting time. (Speak up when you fall in either of these categories.) If for some reason you would like to speak with me in private, let me know; I will clear the office. Or, make an appointment to meet with me outside of office hours.

Students who make proactive, not reactive, use of office hours usually achieve the best results. Proactive use of office hours includes asking questions about material on the slides and in the texts, asking questions about how to better use tools, discussing how to approach problems, etc. If you're familiar with Covey's *The Seven Habits of Highly Effective People* you might recognize those as "Quadrant II" activities—important, but not urgent.

Reactive use of office hours is typically centered around simply getting code to work one way or another (Covey's "Quadrant I"—important and urgent).

Piazza

As mentioned above, we'll be using Piazza. All students are strongly encouraged to participate freely. I hope you'll post questions and comments related to the course material, recommendations for handy tools, URLs for interesting web posts and videos, and whatever else you think is worth mentioning as long as it relates to the course material in some way. Posts about CS-related job opportunities, and events like programming contests and hackathons are welcome, too.

The initial post in any discussion thread that I start is considered to be part of the course material—you are responsible for reading each and every one, and taking action when appropriate. For example, I might ask you to read an article, or experiment with a web site of interest. You may learn more by reading follow-ups by classmates and by me, but at exam- or quiz-time I won't expect you to have read each and every follow-up.

When answering questions on Piazza, the TAs and I will give priority to well-focused questions. And, it's often the case that the task of developing a well-focused question will lead you to an answer on your own.

You can make anonymous posts, but be aware that such posts are only anonymous to classmates, not me or the TAs.

Needless to say, BE CAREFUL that Piazza posts don't give away the solution for a problem. If at all in doubt, send mail instead. (See **Mail** below.) A common case for a post that gives away a solution is when a student is very close to the solution, but doesn't realize it, perhaps because a tiny piece is missing. Haskell and Prolog solutions are often very short, sometimes just a line or two, so posting any code or describing any elements of a solution is very risky.

The typical penalty for giving away a problem is that the guilty student will be required to create an equally great problem as a replacement.

Piazza's "Private posts"—posts seen only by me and the TAs—are disabled because every once in a while somebody forgets to mark an intended private post as private, and everybody sees their code, or more.

Part of the idea of Piazza is to facilitate students helping each other, so don't hesitate to answer questions when you're so inclined. If a post is plain wrong, I'll add a correction. If a post is great, I'll endorse it. If I'm silent, then its quality is probably somewhere in the middle.

If you post a question and later solve it yourself, save everybody some time by posting a follow-up saying the question has been resolved.

Mail

For private communication with the TAs and me, mail to `372s16@cs.arizona.edu`. To ensure quality and accuracy, I want to see all mail between students and TA's. If you're replying to a TA's response to a question, use your mailer's "Reply All" to follow the Cc:'s.

If an email message asks a question or raises a point that I think should be shared with the class, I'll share it. If I think the post deserves kudos, I'll identify the author unless the post specifically requests anonymity, in which case I'll say something like "A student wrote..."

Instant Messaging

IM can be very effective, especially for short questions. Skype is my IM tool of choice. My id on Skype and other popular IM services is `x77686d`. If an IM session starts to run long, I'll often suggest adding in voice and maybe http://join.me, too. Don't pay much attention to my Skype status—sometimes I'm not available when I'm "Online", and at other times I might be invisible but happy to answer. I recommend you start with an "ayt" ("Are you there?") and then follow with your question if I respond.

It's hard to characterize what's best posted on Piazza versus asking on IM so I won't wrestle with that question here.

Original Thoughts

In the movie comedy "Broadcast News" a 14 year-old high-school valedictorian receives a post-commencement beating from a group of bullies. After picking himself up, one of the things he shouts to wound his departing attackers is, "You'll never have an original thought!" That notion of an "original thought" has stayed with me. I hope that you'll have some original thoughts during this semester.

I offer an award of a half-point on your final average for each Original Thought. Observations, analogies, quotable quotes, and clever uses of tools and language constructs are some examples of things that have qualified as Original Thoughts in my classes. Note that an Original Thought does not need to be something that's probably never been thought of before; it just needs to be something that I consider to be reasonably original for you.

Sometimes I'll point out that something you said in class or office hours, or wrote in your observations for an assignment, strikes me as an Original Thought. If you self-identify a potential Original Thought, let me know. In some cases I'll see a glimmer of an Original Thought in something, and encourage you to explore the idea a bit more.

Of course, an Original Thought needs to be something you’ve thought up yourself—don’t send in something you found elsewhere, like a quote, just because it strikes you as being original!

The "bar" rises with each Original Thought for an individual—it's harder to earn a second Original Thought than a first, and a third is harder still.

Academic Integrity

It is unfortunate that this section need be included but experience sadly shows that some students are willing to sacrifice their integrity to obtain a grade they have not earned. For those students who would never cheat, I apologize for the inclusion of this section.

Capsule summary: Don't cheat in my class. Don't make it possible for anybody else to cheat. One strike and you're out!

You are responsible for understanding and complying with the University's Code of Academic Integrity. It can be found at http://deanofstudents.arizona.edu/codeofacademicintegrity. Among other provisions, the Code demands that the work you submit is your own and that submitted work will not subsequently be tampered with. Copying of another student's programs or data is prohibited when they are part of an assignment; it is immaterial whether the copying is by computer, photograph or other means. Allowing such copying, thus facilitating academic dishonesty, is also a Code violation.

In addition to ruining one's grade and damaging one's future, the processing of an academic integrity case requires hours of work by myself and others. I am happy to spend hours helping a student who is earnestly trying to learn the material, but I truly loathe every minute spent on academic integrity cases. Even the very simplest of cases often takes 3-4 hours of my time, not to mention the time of office staff and others.

A violation of the Code of Academic Integrity will typically result in ALL of the following sanctions:

1. **Assignment of failing grade for the course**
2. **A permanent transcript annotation, like "FAILING GRADE ASSIGNED DUE TO CHEATING"**
3. **Recommendation of a one-semester suspension from the university**

When a student is caught cheating and I remind them of the above sanctions they often respond by sending me an email message that is hundreds of words long. They usually start by saying that a failing grade is surely sufficient punishment and that they have learned their lesson; they won't cheat again. They go on to say that some employers won't consider hiring a student whose transcript shows evidence of cheating. (True.) Then they talk about how a suspension from the university will cause them to lose scholarships, job offers, visas, and more. Some mention ill or aging family members who might not live to see them graduate if a suspension is imposed.

If you ever find yourself considering whether cheating is worth the risk, first imagine how the three sanctions above would impact you and your family. I'm willing to go to great lengths to help students learn the course material but if a student cheats, I'm equally willing to impose great penalties.

It is difficult to concisely and completely describe where reasonable collaboration stops and cheating begins, but here are some guidelines:

- It is surely cheating to submit code or text that was not written by you. Exception: If I work you through any part of a solution, either individually or in a group setting, you may freely use any code developed in that process. However, you may not pass along that code to anyone else.

- It is surely cheating to send another student any portion of a solution.

- I consider it to be reasonable to work together on assignments to get to the point of understanding the problems and the language or library elements that are required for solutions.

- I consider it to be reasonable to help another student find a bug if (1) you've finished that problem, and (2) your help consists of asking questions that help the other student to see the problem for themselves. If you find yourself about to dictate code to a classmate, STOP! (Note that occasionally during office hours you will see me dictate code to a student but that's because I've decided that's the best way to help them learn given the full situation at hand. You do not have that prerogative.)

- I consider it to be reasonable to exchange test cases unless test cases are one of the deliverables for a problem.

- If you receive help on a solution but are unable to fully explain how it works, it is surely a mistake to submit it as your own work.

- If your gut feeling is that you're cheating on an assignment or helping somebody else cheat, you probably are.

- **The vast resources of the Internet raise some interesting issues. See the section below on *Google, Stack Overflow, and more.***

- If in the heat of the moment you submit a solution that is not fully yours, or give your work away, and you later reconsider your actions, you and any recipients will at worst lose points for the work involved if you confess before your act is discovered. Conversely, further dishonesty when confronted, which invariably increases the time expenditure, raises the likelihood of more extensive penalties, like a recommendation for a multiple-semester suspension or outright expulsion from the University.

Cheating almost always starts when one student has easy access to the solutions of another. You are expected to take whatever steps are necessary to guard your solutions from others in the class. For example, you should have an unguessable, uncommon password and never share it. Hardcopy should not go into a recycling bin—take it home and dump it in a recycle bin when the course is over. Personal machines, be them laptops or home desktops, should be behind a hardware firewall or running a software firewall.

Failure to take reasonable precautions to ensure the privacy of your solutions may be construed to be facilitating dishonesty, a Code violation. For example, having a weak password such as, but not limited to, your first name, your last name, your phone number, your initials, your sweetie's name, your pet's name, a reversed name, a sequence of consecutive keys, or a common password could be viewed as facilitation of dishonesty. For one list of common passwords see http://cs.arizona.edu/~whm/common-passwords.txt (If you know of a bigger, better list, let me know.) Hint: Don't look for your password by doing `grep MY-PASSWORD common-passwords.txt`—commands like `ps` and `w` show the arguments for currently-running commands!

For some interesting reading about password choice, see http://wpengine.com/unmasked.

Leaving a logged-in and unlocked machine unattended in the presence of another person, even a friend, is questionable. Use a screen locker!

Be very careful when typing a password in the presence of others, be them friends or strangers. The single worst cheating case I've ever handled started when the password of a two-fingered typist was surreptitiously observed. **Don't hesitate to ask someone, even a friend, to turn their head when you're typing your password.**

Malware is a Federal Crime

"*Malware*, short for malicious software, is any software used to disrupt computer operation, gather sensitive information, or gain access to private computer systems."—Wikipedia

In various situations my TAs and I will be running your code. Do not be tempted to slip some malware into your code, even for "harmless fun". Introducing malware into a computer system is a federal crime—see http://fas.org/sgp/crs/misc/97-1025.pdf. I will request permanent expulsion from the university for any student who submits malware, and recommend that the university notify the FBI for further action.

Google, Stack Overflow, and more

Things like Google and Stack Overflow are incredible resources for working professionals but when used to directly search for solutions for problems on an academic assignment they can cause learning and/or creative opportunities to be forever lost.

My expectation is that the material presented in class, practice via exercises, suggested readings, resources cited, and any prior assignments provide everything you need to do every problem on each assignment. I challenge you to limit your Googling to searches that expand your knowledge of the material, and not try to dig up quick solutions to problems. (And, of course, using any code found in such searches would be cheating.)

Posts on websites, IRC channels, mailing lists, etc. that solicit the answer for a problem or a significant piece thereof

will be considered to be cheating! Example:

> *"I'm learning Haskell and trying to write a function that returns True iff the parentheses in a string are properly matched. Any suggestions?"*

Accessibility and Accommodations

It is the University's goal that learning experiences be as accessible as possible. If you anticipate or experience physical or academic barriers based on disability, please let me know immediately so that we can discuss options. You are also welcome to contact Disability Resources (520-621-3268) to establish reasonable accommodations.

Please be aware that the accessible tables and chairs in the classroom should remain available for students who find that standard classroom seating is not usable.

Comparative Programming Languages

CSC 372

Spring 2016

psst...Sign up for Piazza while you're waiting!

Instructor

William Mitchell (`whm`)

I'm a consultant/contractor doing software development and training of software developers. Lots with Java, C++, C, ActionScript, Ruby, Icon, and more. Linux stuff, too.

Occasionally teach a CS course. (337, 352, 372, and others)

Adjunct instructor, not a professor.

Education:

BS CS (North Carolina State University, 1981)
MS CS (University of Arizona, 1984)

Incorrect to say "Dr. Mitchell" or "Professor Mitchell"!

Topic Sequence

- Functional programming with Haskell
- Imperative and object-oriented programming using dynamic typing with Ruby
- Logic programming with Prolog
- Whatever else in the realm of programming languages that we find interesting and have time for.

Note: We'll cover a selection of elements from the languages, not everything.

Themes running through the course

- Discerning the philosophy of a language and how it's manifested.
- Understanding tensions and tradeoffs in language design.
- Acquiring a critical eye for language design.
- Assessing the "mental footprint" of a language.
- Learning how to learn a language. (LHtLaL)

Syllabus highlights

Prereqs, Piazza, Mail

Prerequisites

- CSC 127B or CSC 227; CSC major.
- But, this is a 300-level class!
- Post-127B/227 knowledge of Java is assumed.

Piazza

- Our forum
- Sign up if you haven't already!
- Private posts disabled—use mail
- See Piazza for up to date office hours
- If I start a thread, that first post is part of course material

Mail

- 372s16@cs.arizona.edu goes to whm and TAs
- **For anything more than "Thanks!" use "Reply All" to follow the Cc:'s**

Teaching Assistants

Undergraduate TAs:

- Ben Gaska (`bengaska`)
- Patrick Hickey (`patrickhickey`)

Both Ben and Patrick have had 372 with me.

Office location, hours, and exceptions are in a pinned post on Piazza.

Textbooks

- No texts are required.
- Lectures, handouts, and Piazza postings might be all you need.
- Syllabus and slides have recommendations for supplementary texts, most of which are on Safari.
- Because we only cover a subset of each language, suggested supplementary readings are somewhat problematic.

Grading

Grading

Assignments	60%
Pop quizzes	5%
Mid-term exam	13%
Final exam	22%

Ten-point scale: >= 90 is A, etc. Might go lower.

Original Thoughts

- Half-point on final average for each

Assignments

Assignments—things like:

- Coding in the various languages
- Short answer and essay questions
- Diagrams
- One video project

Late assignments are not accepted!

No late days!

But, extensions for situations beyond your control.

We'll be using lectura

You can develop solutions on your own machines but they'll be graded on the CS machine "lectura", and thus should be tested there.

turnin on lectura will be used for submitting assignments.

Mail us (372s16@cs.arizona.edu) **TODAY** if you haven't worked on lectura or have some gaps in your knowledge; we'll be happy to help you get up and running there.

If you haven't had 352, I recommend you skim through my UNIX slides:

http://cs.arizona.edu/classes/cs352/fall15/unix.pdf

Office hours

- I love office hours!
- Guaranteed hours posted on Piazza
- Open-door policy otherwise
- In-person interaction is most effective
- Skype preferred for IM
- **http://join.me** preferred for screen sharing
- OK to call my mobile but don't leave voice mail! (Send e-mail instead.)

Suggestions for success

- Attend every lecture.
- Arrive on time for lectures.
- Try at least one example on every slide. Try some what-ifs, too.
- Read the write-up for an assignment the day it's handed out.
- Start on assignments early. Don't be a regular in the Thursday Night Club.
- Don't leave any points on the table.
- Don't hesitate to ask for help on an assignment.
- Don't make bad assumptions.

NO CHEATING!

Capsule summary:

Don't cheat in my class!

Don't make it easy for anybody else to cheat!

<u>One strike and you're out!</u>

For a first offense expect these penalties:

- Failing grade for course
- Permanent transcript annotation
- Recommendation for one semester <u>university</u> suspension

A typical first step on the road to ruin is sharing your solutions with your best friend, roommate, etc., who swears to just learn from your work and absolutely not turn it in as their work.

No asking the world for help!

The material covered in lectures, posted on Piazza, etc. should be all you need to do the assignments.

I challenge you to not search the web for solutions for problems on assignments!

Posting problem-specific questions on websites, IRC channels, mailing lists, etc. will be considered to be cheating!

> Example: *I'm learning Haskell and trying to write a function that returns True iff the parentheses in a string are properly matched. Any suggestions?*

My Teaching Philosophy

- I work for you!

- My goal: everybody earns an "A" and averages less than ten hours per week on this course, counting lecture time.

- Effective use of office hours, e-mail, and IM can equalize differences in learning speed.

- I should be able to answer every pertinent question about course material.

- My goal is zero defects in slides, assignments, etc.

  Bug Bounty: One assignment point

- Everything I'll expect you to know on exams will be covered in class, on assignments, or on Piazza.

READ THE SYLLABUS!

(On the Piazza Resources page)

Assignment 1

Assignment 1

- On Piazza
- It's a survey
- Due Tuesday, January 19, 2:00pm
- Worth 10 points
- Maybe 10 minutes to complete
- Thanks for doing it!

Pictures &
Name memorization

Basic questions about programming languages

What is a programming language?

A simple definition:

A system for describing computation.

It is generally agreed that in order for a language to be considered a programming language it must be *Turing Complete*.

One way to prove a language is Turing Complete is to use it to implement a *Turing Machine*, a theoretical device capable of performing any algorithmic computation.

Curio: **github.com/elitheeli/stupid-machines**

What language is most commonly mis-listed on resumes as a programming language?

Does it matter what language is used?

The two extremes:

- If you’ve seen one language you’ve seen them all. Just pick one and get to work.

- Nothing impacts software development so much as the language being used. We must choose very carefully!

Why study programming languages?

- Learn new ways to think about computation.
- Learn to see languages from a critical viewpoint.
- Improve basis for choosing languages for a task.
- Add some tools to the “toolbox”.
- Increase ability to design a new language.

It's been said that a programmer should learn a new language every year.

How old are programming languages?

Plankalkül 1945
Short Code 1949
FORTRAN 1957
ALGOL 1958
COBOL 1959
LISP 1960
BASIC 1964
PL/I 1965
SNOBOL4 1967
SIMULA 67 1967
Pascal 1971
C 1972
Prolog 1972
Smalltalk 1972
ML 1977
Icon 1979
Ada 1980
C++ 1983
Objective-C 1983
Erlang 1986
Perl 1987
Haskell 1990
Python 1990
Ruby 2/24/93
Java 1995
JavaScript 1995
PHP 3 1998
C# 2000
D 2001
Scala 2003
Clojure 2007
Go 2008
Dart 2011
Rust 2012
Corelet 2013
Hack 2014
Swift 2014
Goaldi 2015

How are languages related to each other?

Some of the many *attempts* at a family tree of languages:

digibarn.com/collections/posters/tongues/

levenez.com/lang/

rigaux.org/language-study/diagram.html

www.seas.gwu.edu/~mmburke/courses/csci210-su10/tester-endo.pdf

(Seems to be based on hopl.info data.)

How many languages are there?

en.wikipedia.org/wiki/
Alphabetical_list_of_programming_languages
(700 +/-)

The Language List
people.ku.edu/~nkinners/LangList/Extras/langlist.htm
"about 2,500", but lots of new ones missing

Online Historical Encyclopaedia of Programming Languages
hopl.info
8,945 but has things like "JAVA BEANS" and minor variants like both ANSI Pascal and ISO Pascal.

Bottom line: Nobody knows how many programming languages have been created but it's in the thousands.

What languages are popular right now?

Measured by job postings:

indeed.com/jobtrends

The TIOBE index (multiple factors):

www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Measured by GitHub repositories:

github.com/blog/2047-language-trends-on-github
adambard.com/blog/top-github-languages-2014/

RedMonk

redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

What *is* a good way to measure language popularity?

How do languages help us?

- Free the programmer from details

```
int i = 5;
x = y + z * q;
```

- Detect careless errors

```
int f(String s, char c);
...
int i = f('i', "Testing");
```

- Provide constructs to concisely express a computation

```
for (int i = 1; i <= 10; i++)
    ...
```

How languages help, continued

- Provide portability

  Examples:

  - C provides moderate source-level portability.
  - Java was designed with binary portability in mind.

- Facilitate using a paradigm, such as functional, object-oriented, or logic programming.

How are languages specified?

The specification of a language has two key facets.

- Syntax:

  Specifies the sequences of symbols that are valid programs in the language.

- Semantics:

  Specifies the meaning of a sequence of symbols.

Some languages have specifications that are approved as international standards. Others are defined by nothing more than the behavior of a lone implementation.

Syntax vs. semantics

Consider this expression:

```
    a[i] = x
```

What are some languages in which it is syntactically valid?

In each of those languages, what is the meaning of it?

What are various meanings for these expressions?

```
    x || y
    x y
    *x
```

Building blocks

What are the building blocks of a language?

- Data types
- Operators
- Control structures
- Support for encapsulation
  - Functions
  - Abstract types / Classes
  - Packages / Modules
- Error / Exception handling
- Standard library

What are qualities a language might have?

- Simplicity (“mental footprint”)
- Expressive power
- Readability of programs
- Orthogonality
- Reliability of programs
- Run-time efficiency
- Practical development project size
- Support for a style of programming

What are some tensions between these qualities?

What factors affect popularity?

- Available implementations
- Documentation
- Community
- Vectors of “infection”
- Ability to occupy a niche
- Availability of supporting tools, like debuggers and IDEs
- Cost

The philosophy of a language

What is the philosophy of a language? How is it manifested?

C
- Close to the machine
- Few constraints on the programmer
- High run-time efficiency
- "What you write is what you get."

C++
- Close to both machine and problem being solved
- Support object-oriented programming
- "As close to C as possible, but no closer." — Stroustrup

PostScript
- Page description
- Intended for generation by machines, not humans

What is the philosophy of Java?

A Little U of A CS History

The Founding of UA CS

The UA CS department was founded by Ralph Griswold in 1971. (Hint: know this! Mnemonic aid: ASCII 'G' is 71.)

Griswold was Head of Programming Research at Bell Labs before coming to UA.

Griswold and his team at Bell Labs created the SNOBOL family of languages, culminating with SNOBOL4.

Griswold's interest and prominence in programming languages naturally influenced the course of research at UA.

UA CS's language heritage

In the 1970s and 1980s UA Computer Science was recognized worldwide for its research in programming languages.

These are some of the languages created here:

Cg
EZ
Icon
Leo
MPD
Ratsno
Rebus
Seque
SIL2
SL5
SR
SuccessoR
Y
Goaldi (in progress!)

Along with language design, lots of work was focused on language implementation techniques.

My intersection with Griswold's work

I learned FORTRAN IV and BASIC in a summer school course at Wake Forest during summer after high school.

In first trip to library at NCSU, took home a stack of books on programming languages, including SNOBOL4. Was totally mystified.

Learned PL/I in two-course introduction to computer science sequence.

Took a one-unit course on SNOBOL4 during sophomore year. Used SPITBOL whenever possible in courses thereafter.

Attended a colloquium at NCSU where Ralph Griswold presented a new programming language, named Icon.

Ported Icon to an IBM mainframe and DEC's VAX/VMS.

Went to graduate school here at UA, and worked on Icon as a graduate research assistant for Dr. Griswold.

Functional Programming with Haskell

CSC 372, Spring 2016
The University of Arizona
William H. Mitchell
whm@cs

Programming Paradigms

Paradigms

Thomas Kuhn's *The Structure of Scientific Revolutions* (1962) describes a *paradigm* as a scientific achievement that is...

- "...sufficiently unprecedented to attract an enduring group of adherents away from competing modes of scientific activity."

- "...sufficiently open-ended to leave all sorts of problems for the redefined group of practitioners to resolve."

Kuhn cites works such as Newton's *Principia*, Lavoisier's *Chemistry*, and Lyell's *Geology* as serving to document paradigms.

Paradigms, continued

A paradigm provides a conceptual framework for understanding and solving problems.

A paradigm has a world view, a vocabulary, and a set of techniques that can be applied to solve a problem.

(Another theme for us.)

A question to keep in mind:

What are the problems that programming paradigms attempt to solve?

The procedural programming paradigm

From the early days of programming into the 1980s the dominant paradigm was *procedural programming*:

> Programs are composed of bodies of code (procedures) that manipulate individual data elements or structures.

Much study was focused on how best to decompose a large computation into a set of procedures and a sequence of calls.

Languages like FORTRAN, COBOL, Pascal, and C facilitate procedural programming.

Java programs with a single class are typically examples of procedural programming.

The object-oriented programming paradigm

In the 1990s, object-oriented programming became the dominant paradigm. Problems are solved by creating systems of objects that interact.

> "*Instead of a bit-grinding processor plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires."*—Dan Ingalls

Study shifted from how to decompose computations into procedures to how to model systems as interacting objects.

Languages like C++ and Java facilitate use of an object-oriented paradigm.

The influence of paradigms

The programming paradigm(s) we know affect how we approach problems.

If we use the procedural paradigm, we'll first think about breaking down a computation into a series of steps.

If we use the object-oriented paradigm, we'll first think about modeling the problem with a set of objects and then consider their interactions.

Language support for programming paradigms

If a language makes it easy and efficient to use a particular paradigm, we say that the language supports the paradigm.

What language features are required to support procedural programming?

- The ability to break programs into procedures.

What language features does OO programming require, for OO programming as you know it?

- Ability to define classes that comprise data and methods
- Ability to specify inheritance between classes

Multiple paradigms

Paradigms in a field of science are often incompatible.
Example: geocentric vs. heliocentric model of the universe

Can a programming language support multiple paradigms?
Yes! We can do procedural programming with Java.

The programming language Leda fully supports the procedural, imperative, object-oriented, functional, and logic programming paradigms.

Wikipedia's Programming_paradigm cites 60+ paradigms!

But, are "programming paradigms" really paradigms by Kuhn's definition or are they just characteristics?

The imperative programming paradigm

The imperative paradigm has its roots in programming at the machine level.

Machine-level programming:

- Instructions change memory locations or registers
- Instructions alter the flow of control

Programming with an imperative language:

- Expressions compute values based on memory contents
- Assignments alter memory contents
- Control structures guide the flow of control, perhaps iterating to accumulate a result.

The imperative programming paradigm

Solutions using the procedural or object-oriented paradigms typically make use of the imperative programming paradigm, too.

Two fundamental characteristics of languages that support the imperative paradigm:

- "Variables"—data objects whose values typically change as execution proceeds.
- Support for iteration—a “while” control structure, for example.

Imperative programming, continued

Here's an imperative solution in Java to sum the integers in an array:

```
int sum(int a[])
{
    int sum = 0;
    for (int i = 0; i < a.length; i++)
        sum += a[i];

    return sum;
}
```

The **for** loop causes **i** to vary over the indices of the array, as the variable **sum** accumulates the result.

How can the above solution be improved?

Imperative programming, continued

With Java's "enhanced **for**", also known as a for-each loop, we can avoid array indexing.

```
int sum(int a[])
{
   int sum = 0;
   for (int val: a)
      sum += val;

   return sum;
}
```

Is this an improvement? If so, why?

Can we write **sum** in a non-imperative way?

Imperative programming, continued

We can use recursion to get rid of loops and assignments, but...ouch!

```
int sum(int a[]) { return sum(a, 0); }

int sum(int a[], int i)
{
    if (i == a.length)
        return 0;
    else
        return a[i] + sum(a, i+1);
}
```

Wrt. correctness, which of the three versions would you bet your job on?

The level of a paradigm

Programming paradigms can apply at different levels:

- Making a choice between procedural and object-oriented programming fundamentally determines the high-level structure of a program.

- The imperative paradigm is focused more on the small aspects of programming—how code looks at the line-by-line level.

Java combines the object-oriented and imperative paradigms.

The procedural and object-oriented paradigms apply to *programming in the large*.

The imperative paradigm applies to *programming in the small*.

Background:
Value, type, side effect

Value, type, and side effect

An *expression* is a sequence of symbols that can be evaluated to produce a value.

Here are some Java expressions:

```
'x'
i + j * k
f(args.length * 2) + n
```

There are three questions that are commonly considered when looking at an expression in conventional languages like Java and C:

- What value does the expression produce?
- What's the type of that value?
- Does the expression have any side effects?

Mnemonic aid: Imagine you're wearing a vest that's reversed.
"vest" reversed is "t-se-v": type/side-effect/value.

Value, type, and side effect, continued

What is the value of the following Java expressions?

```
3 + 4
     7

1 < 2
     true

"abc".charAt(1)
     'b'

s = "3" + 4
     "34"

"a,bb,c3".split(",")
     An array with three elements: "a", "bb" and "c3"

"a,bb,c3".split(",")[2]
     "c3"

"a,bb,c3".split(",")[2].charAt(0) == 'X'
     false
```

Value, type, and side effect, continued

What is the type of each of the following Java expressions?

```
3 + 4
    int

1 < 2
    boolean

"abc".charAt(1)
    char

s = "3" + 4
    String

"a,bb,c3".split(",")
    String []

"a,bb,c3".split(",")[2]
    String

"a,bb,c3".split(",")[2].charAt(0) == 'X'
    boolean
```

When we ask,

"What's the type of this expression?"

we're actually asking this:

"What's the type of the value produced by this expression?"

Value, type, and side effect, continued

A "side effect" is a change to the program's observable data or to the state of the environment in which the program runs.

Which of these Java expressions have a side effect?

```
x + 3 * y
```

No side effect. A computation was done but no evidence of it remains.

```
x += 3 * y
```

Side effect: **3 * y** *is added to* **x**.

```
s.length() > 2 || s.charAt(1) == '#'
```

No side effect. A computation was done but no evidence of it remains.

Value, type, and side effect, continued

More expressions to consider wrt. side effects:

"testing".toUpperCase()

A string "TESTING" was created somewhere but we can't get to it. No side effect.

L.add("x"), where L is an ArrayList

An element was added to L. Definitely a side-effect!

System.out.println("Hello!")

Side effect: "Hello!" went somewhere.

window.checkSize()

*We can't tell without looking at **window.checkSize()**!*

The hallmark of imperative programming

Side effects are the hallmark of imperative programing.

Programs written in an imperative style are essentially an orchestration of side effects.

Recall:

```
int sum = 0;
for (int i = 0; i < a.length; i++)
   sum += a[i];
```

Can we program without side effects?

The Functional Paradigm

The functional programming paradigm

A key characteristic of the functional paradigm is writing functions that are like pure mathematical functions.

Pure mathematical functions:

- Always produce the same value for given input(s)
- Have no side effects
- Can be easily combined to produce more powerful functions

Ideally, functions are specified with notation that's similar to what you see in math books—cases and expressions.

Functional programming, continued

Other characteristics of the functional paradigm:

- Values are never changed but lots of new values are created.
- Recursion is used in place of iteration.
- Functions are values. Functions are put into data structures, passed to functions, and returned from functions. Lots of temporary functions are created.

Based on the above, how well would Java support functional programming? How about C?

Haskell basics

What is Haskell?

Haskell is a pure functional programming language; it has no imperative features.

Was designed by a committee with the goal of creating a standard language for research into functional programming.

First version appeared in 1990. Latest version is known as Haskell 2010.

Is said to be *non-strict*—it supports *lazy evaluation*.

It is not object-oriented in any way.

Haskell resources

Website: **haskell.org**
All sorts of resources!

Books: (on Safari, too)

Learn You a Haskell for Great Good!, by Miran Lipovača
http://learnyouahaskell.com (Known as LYAH.)

Programming in Haskell, by Hutton
Note: See appendix B for mapping of non-ASCII chars!

Real World Haskell, by O'Sullivan, Stewart, and Goerzen
http://realworldhaskell.org (I'll call it RWH.)

Haskell 2010 Report (I'll call it H10.)
http://haskell.org/definition/haskell2010.pdf

Interacting with Haskell

On lectura we can interact with Haskell by running **ghci**:

```
% ghci
GHCi, version 7.4.1: ...more...  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
>
```

With no arguments, **ghci** starts a read-eval-print loop (REPL)—expressions that we type at the prompt (**>**) are evaluated and the result is printed.

Note: the standard prompt is **Prelude>** but I've got

```
:set prompt "> "
```

in my **~/.ghci** file.

Interacting with Haskell, continued

Let's try some expressions with **ghci**:

Go *live!*

```
> 3+4
7

> 3 * 4.5
13.5

> (3 > 4) || (5 < 7)
True

> 2 ^ 200
160693804425899027554196209234116260252220299378
2792835301376

> "abc" ++ "xyz"
"abcxyz"
```

Interacting with Haskell, continued

We can use **:help** to see available commands:

```
> :help
 Commands available from the prompt:
  <statement>                  evaluate/run <statement>
  :                            repeat last command
  :{\n ..lines.. \n:}\n        multiline command
```

...lots more...

Interacting with Haskell, continued

The command **:set +t** causes types to be shown:

```
> :set +t
> 3+4
7
it :: Integer

> 3 == 4
False
it :: Bool
```

"**::**" is read as "has type". The value of the expression is "bound" to the name **it**.

Note that **:set +t** is not a Haskell expression—it's a command recognized by **ghci**.

Interacting with Haskell, continued

We can use **it** in subsequent computations:

```
> 3+4
7
it :: Integer

> it + it * it
56
it :: Integer

> it /= it
False
it :: Bool
```

Extra Credit Assignment 1

For two assignment points of extra credit:

1. Run **ghci** (or WinGHCi) somewhere and try ten Haskell expressions with some degree of variety. (Not just ten additions, for example!) Do a **:set +t** at the start.

2. Capture the output and put it in a plain text file, eca1.txt. No need for your name, NetID, etc. in the file. No need to edit out errors.

3. On lectura, turn in eca1.txt with the following command:

```
% turnin 372-eca1 eca1.txt
```

Due: At the start of the next lecture after we hit this slide.

Needless to say, feel free to read ahead in the slides and show experimentation with the following material, too.

Haskell version issues

lectura has version 2012.1.0.0 of the Haskell Platform, which has version 7.4.1 of **ghci** but the latest version is 7.10.3.

In 7.10.x a number of functions that operate on lists were switched to operating on "Foldable"s instead. IMO, this extra level of abstraction makes the language harder to learn, so I plan to avoid 7.10.x this semester.

If you want to install Haskell on your own machine, I recommend that you get the Haskell Platform 2014.2.0.0, which has version 7.8.3 of **ghci**. (URLs on next slide.)

As far as I know, there are no significant compatibility issues between 7.8.3 and lectura's 7.4.1 that will impact our usage.

Haskell downloads

https://www.haskell.org/platform/prior.html has prior versions of the Haskell Platform.

Under **2014**, on the line **2014.2.0.0, August 2014 ⟹ ...**

For OS X, get "**Mac OS X, 64bit**".

For Windows, "**Windows, 32bit**" should be fine, but if you have trouble, (1) let us know and (2) go ahead and try the 64-bit version.

The ~/.ghci file

When ghci starts up on Linux or OS X it looks for the file ~/.ghci – a .ghci file in the user's home directory.

I have these two lines in my ~/.ghci file on both my Mac and on lectura:

```
:set prompt "> "
:m +Text.Show.Functions
```

The first line simply sets the prompt to something I like.

The second line is very important:

It loads a module that allows functions to be printed as values, although just showing <function> for function values. Without it, lots of examples in these slides won't work!

~/.ghci, continued

Goofy fact: **~/.ghci** must not be group- or world-writable!

If you see something like this,

```
*** WARNING: /p1/hw/whm/.ghci is writable by
someone else, IGNORING!
```

Fix it at the shell prompt with this:

```
% chmod og-w  ~/.ghci
```

Details on **.ghci** and lots more can be found in
downloads.haskell.org/~ghc/latest/docs/users_guide.pdf

~/.ghci, continued

On Windows, ghci and WinGHCi use a different initialization file:

```
%APPDATA%\ghc\ghci.conf
```

(Note: the file is named ghci.conf, not .ghci!)

%APPDATA% represents the location of your Application Data directory. You can find that path by typing set appdata in a command window, like this:

```
C:\>set appdata
APPDATA=C:\Users\whm\Application Data
```

Combing the two, the full path to the file <u>for me</u> would be

```
C:\Users\whm\Application Data\ghc\ghci.conf
```

Functions and function types

Calling functions

In Haskell, *juxtaposition* indicates a function call:

```
> negate 3
-3
it :: Integer

> even 5
False
it :: Bool

> pred 'C'
'B'
it :: Char

> signum 2
1
it :: Integer
```

Note: These functions and many more are defined in the Haskell "Prelude", which is loaded by default when `ghci` starts up.

Calling functions, continued

Function call with juxtaposition is left-associative.

signum negate 2 means (signum negate) 2

```
> signum negate 2
<interactive>:40:1:    -- It's an error!
   No instance for (Num (a0 -> a0)) arising from a
use of `signum'
...
```

We add parentheses to call negate 2 first:

```
> signum (negate 2)
-1
it :: Integer
```

Calling functions, continued

Function call with juxtaposition has higher precedence than any operator.

```
> negate 3+4
1
it :: Integer
```

`negate 3 + 4` means `(negate 3) + 4.` Use parens to force + first:

```
> negate (3 + 4)
-7
it :: Integer

> signum (negate (3 + 4))
-1
it :: Integer
```

Function types

Haskell's Data.Char module has a number of functions for working with characters. We'll use it to start learning about function types.

```
> :m +Data.Char          (:m(odule) loads a module)

> isLower 'b'
True
it :: Bool

> toUpper 'a'
'A'
it :: Char

> ord 'A'
65
it :: Int

> chr 66
'B'
it :: Char
```

We can also reference a function in a module with a *qualified name*:

```
% ghci
...
> Data.Char.ord 'G'
71
```

Function types, continued

We can use **ghci**'s **:type** command to see what the type of a function is:

```
> :type isLower
isLower :: Char -> Bool
```
(read -> as "to")

The type **Char -> Bool** means that **isLower** is a function that takes an argument of type **Char** and produces a result of type **Bool.**

Using **ghci**, what are the types of **toUpper**, **ord**, and **chr**?

We can use **:browse Data.Char** to see everything in the module.

Type consistency

Like most languages, Haskell requires that expressions be *type-consistent* (or *well-typed*).

Here is an example of an inconsistency:

```
> chr 'x'
<interactive>:32:5:
  Couldn't match expected type Int with actual type Char
  In the first argument of `chr', namely 'x'

> :type chr
chr :: Int -> Char

> :type 'x'
'x' :: Char
```

chr requires its argument to be an **Int** but we gave it a **Char**. We can say that **chr 'x'** is *ill-typed*.

Type consistency, continued

State whether each expression is well-typed and if so, its type.

```
'a'
isUpper
isUpper 'a'
not (isUpper 'a')
not not (isUpper 'a')
toUpper (ord 97)
isUpper (toUpper (chr 'a'))
isUpper (intToDigit 100)
```

```
'a' :: Char
chr :: Int -> Char
digitToInt :: Char -> Int
intToDigit :: Int -> Char
isUpper :: Char -> Bool
not :: Bool -> Bool
ord :: Char -> Int
toUpper :: Char -> Char
```

Sidebar: Key bindings in **ghci**

ghci uses the **haskeline** package to provide line-editing.

A few handy bindings:

TAB	completes identifiers
^A	Start of line
^E	End of line
^R	Incremental search backwards

More:

http://trac.haskell.org/haskeline/wiki/KeyBindings

Sidebar: Using a REPL to help learn a language

As we've seen, **ghci** provides a REPL (read-eval-print loop) for Haskell.

What are some other languages that have a REPL available?

How does a REPL help us learn a language?

Is there a REPL for Java?

javarepl.com

What characteristics does a language need to support a REPL?

If there's no REPL for a language, how hard is it to write one?

Type classes

What's the type of **negate**?

Recall the **negate** function:

```
> negate 5
-5
it :: Integer

> negate 5.0
-5.0
it :: Double
```

What's the type of **negate**? (Is it both **Integer -> Integer** and **Double -> Double**??)

Type classes

Bool, **Char**, and **Integer** are examples of Haskell types.

Haskell also has *type classes*. A type class specifies the operations must be supported on a type in order for that type to be a member of that type class.

Num is one of the many type classes defined in the Prelude.

:info Num shows that for a type to be a **Num,** it must support addition, subtraction, multiplication and four functions: **negate**, **abs**, **signNum**, and **fromInteger**. (The **Num** club!)

The Prelude defines four *instances* of the **Num** type class: **Int** (word-size), **Integer** (unlimited size), **Float** and **Double**.

Type classes, continued

Here's the type of **negate:**

```
> :type negate
negate :: Num a => a -> a
```

The type of **negate** is specified using a *type variable*, **a**.

The portion **a -> a** specifies that **negate** returns a value having the same type as its argument.

"If you give me an ***Int****, I'll give you back an* ***Int****."*

The portion **Num a =>** is a *class constraint*. It specifies that the type **a** must be an instance of the type class **Num**.

How can we state the type of **negate** in English?

negate *accepts any value whose type is an instance of* ***Num****.*
It returns a value of the same type.

Type classes, continued

What type do integer literals have?

```
> :type 3
3 :: Num a => a

> :type (-27)                -- Note: Parens needed!
(-27) :: Num a => a
```

Literals are typed with a class constraint of **Num**, so they can be used by any function that accepts **Num a => a**.

Type classes, continued

Let's check the type of a decimal fraction:

```
> :type 3.4
3.4 :: Fractional a => a
```

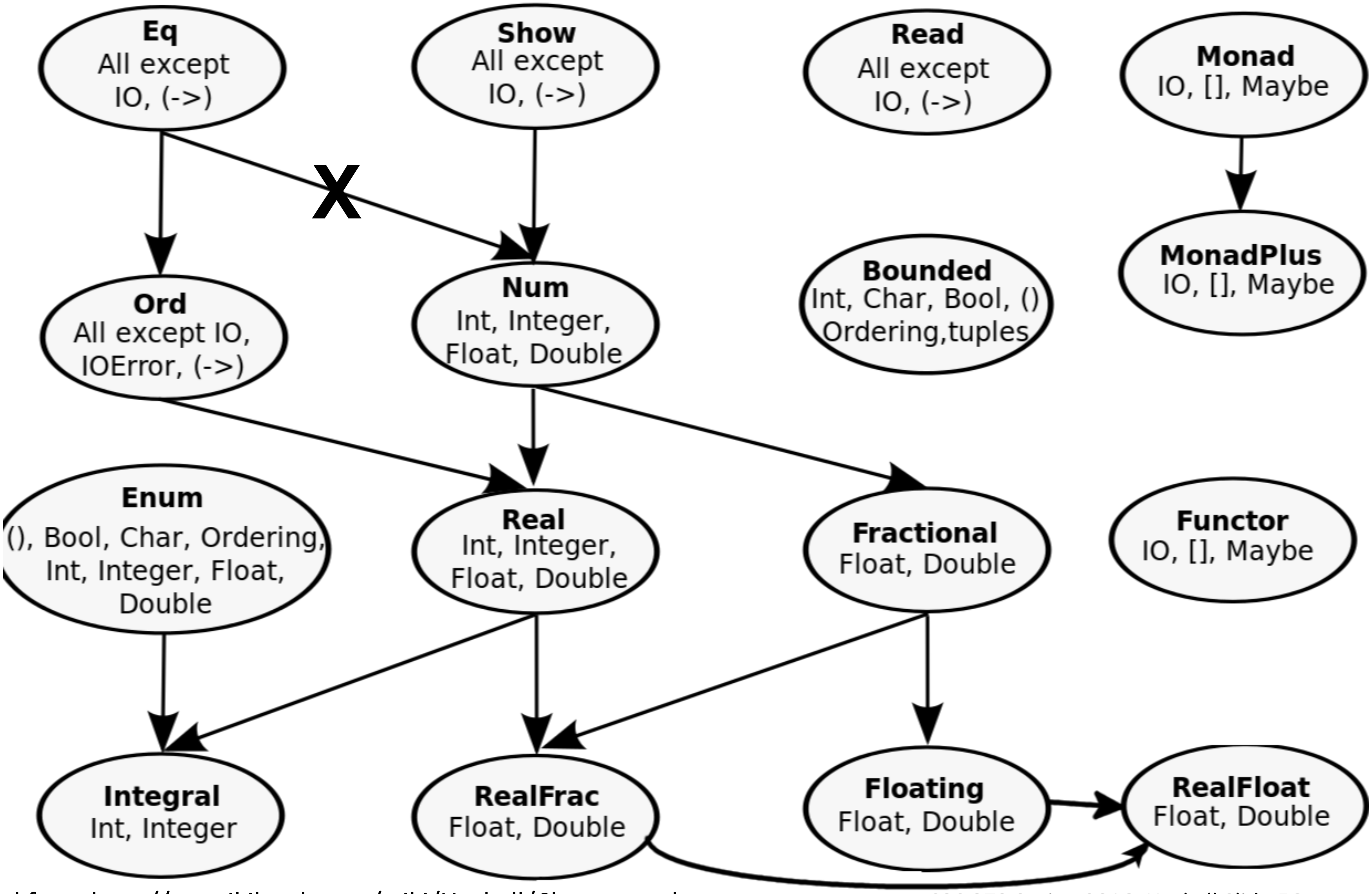
Will negate 3.4 work?

```
> :type negate
negate :: Num a => a -> a

> negate 3.4
-3.4
```

Speculate: Why does it work?

Type classes, continued

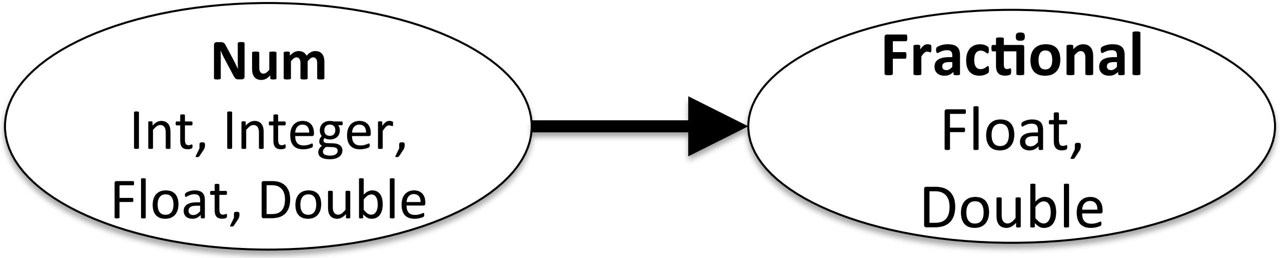
Haskell type classes form a hierarchy. The Prelude has these:



Adapted from http://en.wikibooks.org/wiki/Haskell/Classes_and_types

Type classes, continued

Excerpt:



The arrow from **Num** to **Fractional** means that a **Fractional** can be used as a **Num**. (What does that remind you of?)

Given

```
negate :: Num a => a -> a
```

and

```
5.0 :: Fractional a => a
```

then

negate 5.0 is valid.

Note that the bubbles also show the types that are instances of the type class. (Do **:info Num** again, and **:info Fractional**, too.)

Type classes, continued

What's meant by the type of **truncate**?

```
truncate :: (Integral b, RealFrac a) => a -> b
```

truncate accepts a type whose type class is an instance of **RealFrac** but produces a type whose type class is an instance of **Integral**.

LYAH pp. 27-33 has a good description of the Prelude's type classes. ("Type Classes 101")

Note that type classes are not required for functional programming but because Haskell makes extensive use of them, we must learn about them.

negate is *polymorphic*

In essence, **negate :: Num a => a -> a** describes many functions:

```
negate :: Integer -> Integer
negate :: Int -> Int
negate :: Float -> Float
negate :: Double -> Double
...and more...
```

negate is a *polymorphic function*. It handles values of many forms.

If a function's type has any type variables, it's a polymorphic function.

How does Java handle this problem? How about C? C++?

Sidebar: LHtLaL—introspective tools

:set +t, **:type** and **:info** are three introspective tools that we can use to help learn Haskell.

When learning a language, look for such tools early on.

Some type-related tools in other languages:

Python: **type(*expr*)** and **repr(*expr*)**

JavaScript: **typeof(expr)**

PHP: **var_dump(*expr1*, *expr2*, ...)**

C: **sizeof(*expr*)**

Java: **getClass()**

What's a difference between **ghci**'s **:type** and Java's **getClass()**?

Sidebar, continued

Here's a Java program that makes use of the "boxing" mechanism to show the type of values, albeit with wrapper types for primitives.

```
public class exprtype {
   public static void main(String args[]) {
      int n = 1;
      showtype(n++, 3 + 'a');
      showtype(n++, 3 + 4.0);
      showtype(n++, "a,b,c".split(","));
      showtype(n++, new HashMap<String,Integer>());
   }
   private static void showtype(int num, Object o) {
      System.out.format("%d: %s\n", num, o.getClass());
   }
}
```

Output:

```
1: class java.lang.Integer
2: class java.lang.Double
3: class [Ljava.lang.String;
4: class java.util.HashMap    (Note: no String or Integer—type erasure!)
```

More on functions

Writing simple functions

A function can be defined in the REPL by using **let**. Example:

```
> let double x = x * 2
double :: Num a => a -> a

> double 5
10
it :: Integer

> double 2.7
5.4
it :: Double

> double (double (double 111111111111))
888888888888
it :: Integer
```

Simple functions, continued

More examples:

```
> let neg x = -x
neg :: Num a => a -> a

> let isPositive x = x > 0
isPositive :: (Num a, Ord a) => a -> Bool

> let toCelsius temp = (temp - 32) * 5/9
toCelsius :: Fractional a => a -> a
```

The determination of types based on the operations performed is known as *type inferencing*. (More on it later!)

Note: function and parameter names must begin with a lowercase letter or _. (If capitalized they're assumed to be *data constructors*.)

Simple functions, continued

We can use :: *type* to constrain a function's type:

```
> let neg x = -x :: Integer
neg :: Integer -> Integer

> let toCelsius temp = (temp - 32) * 5/9 :: Double
toCelsius :: Double -> Double
```

:: *type* has low precedence; parentheses are required for this:

```
> let isPositive x = x > (0::Integer)
isPositive :: Integer -> Bool
```

Note that :: *type* applies to an expression, not a function.

We'll use :: *type* to simplify some following examples.

Sidebar: loading functions from a file

We can put function definitions in a file. When we do, **we leave off the let**!

I've got four function definitions in the file **simple.hs**, as shown with the UNIX **cat** command:

```
% cat simple.hs
double x = x * 2 :: Integer   -- Note: no "let"!
neg x = -x :: Integer
isPositive x = x > (0::Integer)
toCelsius temp = (temp - 32) * 5/(9::Double)
```

The **.hs** suffix is required.

Sidebar, continued

Assuming **simple.hs** is in the current directory, we can load it with **:load** and see what we got with **:browse**.

```
% ghci
> :load simple
[1 of 1] Compiling Main             ( simple.hs, interpreted )
Ok, modules loaded: Main.

> :browse
double :: Integer -> Integer
neg :: Integer -> Integer
isPositive :: Integer -> Bool
toCelsius :: Double -> Double
```

Note the colon in **:load**, and that the suffix **.hs** is assumed.

We can use a path, like **:load ~/372/hs/simple**, too.

Sidebar: My usual edit-run cycle

ghci is clumsy to type! I've got an **hs** alias in my **~/.bashrc**:

```
alias hs=ghci
```

I specify the file I'm working with as an argument to **hs**.

```
% hs simple
GHCi, version 7.8.3 ...
[1 of 1] Compiling Main          ( simple.hs, interpreted )
Ok, modules loaded: Main.
> ... experiment ...
```

After editing in a different window, I use **:r** to reload the file.

```
> :r
[1 of 1] Compiling Main          ( simple.hs, interpreted )
Ok, modules loaded: Main.
> ...experiment some more...
```

Lather, rinse, repeat.

Functions with multiple arguments

Here's a function that produces the sum of its two arguments:

```
> let add x y = x + y :: Integer
```

Here's how we call it: (no commas or parentheses!)

```
> add 3 5
8
```

Here is its type:

```
> :type add
add :: Integer -> Integer -> Integer
```

The operator -> is right-associative, so the above means this:

```
add :: Integer -> (Integer -> Integer)
```

But what does that mean?

Multiple arguments, continued

Recall our negate function:

```
> let neg x = -x :: Integer
neg :: Integer -> Integer
```

Here's **add** again, with parentheses added to show precedence:

```
> let add x y = x + y :: Integer
add :: Integer -> (Integer -> Integer)
```

add is a function that takes an integer as an argument and produces a function as its result!

add 3 5 means **(add 3) 5**

Call **add** with the value 3, producing a nameless function.

Call that nameless function with the value 5.

Partial application

When we give a function fewer arguments than it requires, the resulting value is called a *partial application*. It is a function.

We can bind a name to a partial application like this:

```
> let plusThree = add 3
plusThree :: Integer -> Integer
```

The name **plusThree** now references a function that takes an **Integer** and returns an **Integer**.

What will **plusThree 5** produce?

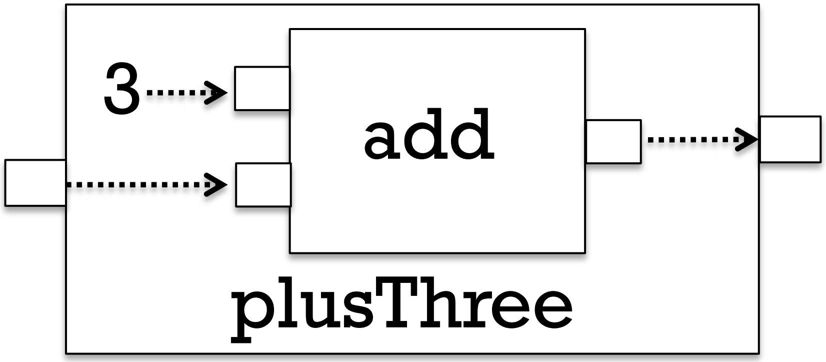
```
> plusThree 5
8
it :: Integer
```

Partial application, continued

At hand:

```
> let add x y = x + y :: Integer
add :: Integer -> (Integer -> Integer)  -- parens added

> let plusThree = add 3
plusThree :: Integer -> Integer
```

Let's picture **add** and **plusThree** as boxes with inputs and outputs:



An analogy: **plusThree** is like a calculator where you've clicked 3, then +, and handed it to somebody.

Partial application, continued

At hand:

```
> let add x y = x + y :: Integer
add :: Integer -> (Integer -> Integer)  -- parens added
```

Another: *(with parentheses added to type to aid understanding)*

```
> let add3 x y z = x + y + z :: Integer
add3 :: Integer -> (Integer -> (Integer -> Integer))
```

These functions are said to be defined in *curried* form, which allows partial application of arguments.

The idea of a partially applicable function was first described by Moses Schönfinkel. It was further developed by Haskell B. Curry. Both worked wtih David Hilbert in the 1920s.

$\boxed{log_2\, n}$

What prior use have you made of partially applied functions?

Note: Next set of slides!

REPLACEMENTS

Put a big "X" on slides 74-76 in the 1-76 set and continue with this set.

Some key points

- The *general form* of a function definition (for now):

  name param1 param2 ... paramN = expression

  —At the **ghci** prompt, use **let ...**

- A function with a type like **Integer -> Char -> Char** takes two arguments, an **Integer** and a **Char**. It produces a **Char**.

- Remember that **->** is a right-associative type operator.

  Integer -> Char -> Char means **Integer -> (Char -> Char)**

- A function call like

  f x y z

  means

  ((f x) y) z

  and (conceptually) causes two temporary, unnamed functions to be created.

Some key points, continued

- Calling a function with fewer arguments than it requires creates a *partial application*, a function value.

- There's really nothing special about a partial application—it's just another function.

Another view of partial application

Consider this function:

```
    let f x y z = x + y + y * z

    let f1 = f 3
```

is equivalent to

```
    let f1 y z = 3 + y + y * z

    let f2 = f1 5
```

is equivalent to

```
    let f2 z = 3 + 5 + 5 * z

    let val = f2 7
```

is equivalent to

```
    let val = f 3 5 7
```

and

```
    let val = f1 5 7
```

One way to think of partial application is that as each argument is provided, a parameter is dropped and the argument's value is "wired" into the expression for the function, producing a new function with one less parameter.

Exercise

Add parentheses to show the order of operations for the following expression:

```
    f g 3 4 + x f 3 g(5*x)
```

Note that the expression is function calls and an addition.

Recall that function call is the highest precedence operation and is left-associative.

Let's first note that the addition is lowest precedence, and last:

```
    (f g 3 4) + (x f 3 g (5*x))
```

Let's now reflect the left-associativity of function call:

```
    (((f g) 3) 4) + ((((x f) 3) g) (5*x))
```

Exercise

Problem: Define a function **min3** that computes the minimum of three values. The Prelude has a **min** function.

```
> min3 5 2 10
2
```

Solution:

```
> let min3 a b c = min a (min b c)
min3 :: Ord a => a -> a -> a -> a
```

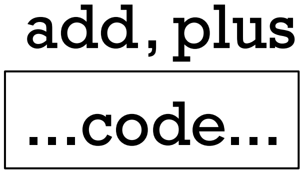
What are some types that **min3** can be used with?

Functions are values

A fundamental characteristic of a functional language: functions are values that can be used as flexibly as values of other types.

This **let** creates a function value and binds the name **add** to it.

```
> let add x y = x + y
```



This **let** binds the name **plus** to the value of **add**, whatever it is.

```
> let plus = add
```

(Diagram here merged w/ above)

Either name can be used to reference the function value:

```
> add 3 4
7
> plus 5 6
11
```

Functions as values, continued

What does the following suggest to you?

```
> :info add
add :: Num a => a -> a -> a

> :info +
class Num a where
  (+) :: a -> a -> a
  ...
infixl 6 +
```

Operators in Haskell are simply functions that have a symbolic name bound to them.

infixl 6 + shows that the symbol **+** can be used as a infix operator that is left associative and has precedence level 6.

Use **:info** to explore these operators: **==**, **>**, **+**, *****,**||**, **^**, **^^** and ******.

Function/operator equivalence

To use an operator like a function, enclose it in parentheses:

```
> (+) 3 4
7
```

Conversely, we can use a function like an operator by enclosing it in backquotes:

```
> 3 `add` 4
7

> 11 `rem` 3
2
```

Speculate: do **`add`** and **`rem`** have precedence and associativity?

Sidebar: Custom operators

Haskell lets us define custom operators.

Example: (loaded from a file)

```
(+%) x percentage = x + x * percentage / 100
infixl 6 +%
```

Usage:

```
> 100 +% 1
101.0
> 12 +% 25
15.0
```

The characters ! # $ % & * + . / < = > ? @ \ ^ | - ~ : and non-ASCII Unicode symbols can be used in custom operators.

Modules often define custom operators.

Reference: Operators from the Prelude

Precedence	Left associative operators	Non associative operators	Right associative operators
9	!!		.
8			^, ^^, **
7	*, /, `div`, `mod`, `rem`, `quot`		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, `elem`, `notElem`	
3			&&
2			||
1	>>, >>=		
0			$, $!, `seq`

Note: From page 51 in Haskell 2010 report

Type Inferencing

Type inferencing

It was briefly mentioned that Haskell performs type infererencing: the types of values are inferred based on the operations performed.

Example:

```
> let isCapital c = c >= 'A' && c <= 'Z'
isCapital :: Char -> Bool
```

Because c is being compared to 'A' and 'Z', both of which are type Char, c is inferred to be a Char.

Type inferencing, continued

Recall **ord** in the **Data.Char** module:

```
> :t ord
ord :: Char -> Int
```

What type will be inferred for the following function?

```
f x y = ord x == y
```

1. The argument of **ord** is a **Char**, so **x** must be a **Char**.

2. The result of **ord**, an **Int**, is compared to **y**, so **y** must be an **Int**.

Let's try it:

```
> let f x y = ord x == y
f :: Char -> Int -> Bool
```

Type inferencing, continued

Recall this example:

```
> let isPositive x = x > 0
isPositive :: (Num a, Ord a) => a -> Bool
```

:info shows that **>** operates on types that are instances of **Ord**:

```
> :info >
class Eq a => Ord a where
  (>) :: a -> a -> Bool
  ...
```

Because **x** is an operand of **>**, Haskell infers that the type of **x** must be a member of the **Ord** type class.

Because **x** is being compared to 0, Haskell also infers that the type of **x** must be a member of the **Num** type class.

Type inferencing, continued

If a contradiction is reached during type inferencing, it's an error.

The function below uses **x** as both a **Num** and a **Char**.

```
> let g x y = x > 0 && x > '0'

<interactive>:20:17:
    No instance for (Num Char) arising from the literal `0'
    Possible fix: add an instance declaration for (Num Char)
    In the second argument of `(>)', namely `0'
    In the first argument of `(&&)', namely `x > 0'
    In the expression: x > 0 && x > '0'
```

Note that Haskell's suggested fix, making **Char** be an instance of the **Num** type class, isn't very good.

Type Specifications

Type specifications for functions

It's a good practice to specify the type of a function along with its definition in a file.

Examples, using **cat** to make it clear that they're in a file:

```
% cat typespecs.hs
min3::Ord a => a -> a -> a -> a
min3 x y z = min x (min y z)

isCapital :: Char -> Bool
isCapital c = c >= 'A' && c <= 'Z'

isPositive :: (Num a, Ord a) => a -> Bool
isPositive x = x > 0
```

Type specifications, continued

Sometimes type specifications can backfire. What's the ramification of the difference in these two type specifications?

```
add1::Num a => a -> a -> a
add1 x y = x + y

add2::Integer -> Integer -> Integer
add2 x y = x + y
```

add1 can operate on Nums but a2 requires Integers.

Challenge: Without using ::*type*, show an expression that works with add1 but fails with add2.

Type specification for functions, continued

There are two pitfalls for Haskell novices related to type specifications for functions:

1. Specifying a type, such as **`Integer`**, rather than a type class, such as **`Num`**, may make a function's type needlessly specific, like **`add2`** on the previous slide.

2. In some cases the type can be plain wrong without the mistake being obvious, leading to a baffling problem. (An "Ishihara".)

Recommendation:

Try writing functions without a type specification and see what type gets inferred. If the type looks reasonable, and the function works as expected, add a specification for that type.

Type specifications can prevent Haskell's type inferencing mechanism from making a series of bad inferences that lead one far away from the actual source of an error.

Continuation with indentation

A Haskell source file is a series of *declarations*. Here's a file with two declarations:

```
% cat indent1.hs
add::Integer -> Integer -> Integer
add x y = x + y
```

A declaration can be continued across multiple lines by indenting subsequent lines more than the first line of the declaration. These weaving declarations are poor style but are valid:

```
add
   ::
 Integer-> Integer-> Integer
add x y
  =
 x
   + y
```

Indentation, continued

A line that starts in the same column as the previous declaration ends that previous declaration and starts a new one.

```
% cat indent2.hs
add::Integer -> Integer -> Integer
add x y =
x + y

% ghci indent2
...
indent2.hs:3:1:
   parse error (possibly incorrect indentation or
mismatched brackets)
Failed, modules loaded: none.
```

Note that 3:1 indicates line 3, column 1.

Guards

Guards

Recall this characteristic of functional programming:

> "Ideally, functions are specified with notation that's similar to what you see in math books—cases and expressions."

This function definition uses *guards* to specify three cases:

```
sign x | x < 0 = -1
       | x == 0 = 0
       | otherwise = 1
```

Notes:

- No let—this definition is loaded from a file with :load
- sign x appears just once. First guard might be on next line.
- The *guard* appears between | and =, and produces a Bool
- What is otherwise?

Guards, continued

Problem: Using guards, define a function **smaller**, like **min**:

```
> smaller 7 10
7

> smaller 'z' 'a'
'a'
```

Solution:

```
smaller x y
    | x <= y = x
    | otherwise = y
```

Guards, continued

Problem: Write a function **weather** that classifies a given temperature as hot if 80+, else nice if 70+, and cold otherwise.

```
> weather 95
"Hot!"
> weather 32
"Cold!"
> weather 75
"Nice"
```

A solution that takes advantage of the fact that <u>guards are tried in turn</u>:

```
weather temp | temp >= 80 = "Hot!"
             | temp >= 70 = "Nice"
             | otherwise = "Cold!"
```

if-else

Haskell's **if-else**

Here's an example of Haskell's **if-else**:

```
> if 1 < 2 then 3 else 4
3
```

How does this compare to the **if-else** in Java?

Sidebar: Java's **if-else**

Java's **if-else** is a statement. It cannot be used where a value is required.

Java's conditional operator is the analog to Haskell's **if-else**.

```
1 < 2 ? 3 : 4
```

(Java conditional, a.k.a ternary operator)

It's an expression that can be used when a value is required.

Java's if-else statement has an else-less form but Haskell's **if-else** does not. Why doesn't Haskell allow it?

Java's **if-else** vs. Java's conditional operator provides a good example of a *statement* vs. an *expression*.

Pythoners: Is there an **if-else** expression in Python?

```
3 if 1 < 2 else 4
```

Haskell's **if-else**, continued

What's the type of these expressions?

```
> :type if 1 < 2 then 3 else 4
if 1 < 2 then 3 else 4 :: Num a => a

> :type if 1 < 2 then '3' else '4'
if 1 < 2 then '3' else '4' :: Char

> if 1 < 2 then 3 else '4'
  <interactive>:12:15:
   No instance for (Num Char) arising from the literal `3'

> if 1 < 2 then 3
   <interactive>:13:16:
   parse error (possibly incorrect indentation or
   mismatched brackets)
```

Guards vs. **if-else**

Which of the versions of **sign** below is better?

```
sign x
    | x < 0 = -1
    | x == 0 = 0
    | otherwise = 1
```

```
sign x = if x < 0 then -1
                       else if x == 0 then 0
                                          else 1
```

We'll later see that *patterns* add a third possibility for expressing cases.

A Little Recursion

Recursion

A recursive function is a function that calls itself either directly or indirectly.

Computing the factorial of a integer (N!) is a classic example of recursion. Write it in Haskell (and don't peek below!) What is its type?

```
factorial n
    | n == 0 = 1          -- Base case, 0! is 1
    | otherwise = n * factorial (n - 1)

> :type factorial
factorial :: (Eq a, Num a) => a -> a

> factorial 40
815915283247897734345611269596115894272000000000
```

Recursion, continued

One way to manually trace through a recursive computation is to underline a call, then rewrite the call with a textual expansion.

```
factorial n
    | n == 0 = 1
    | otherwise = n * factorial (n – 1)
```

$\underline{\text{factorial } 4}$

$4 * \underline{\text{factorial } 3}$

$4 * 3 * \underline{\text{factorial } 2}$

$4 * 3 * 2 * \underline{\text{factorial } 1}$

$4 * 3 * 2 * 1 * \underline{\text{factorial } 0}$

$4 * 3 * 2 * 1 * 1$

Recursion, continued

Consider repeatedly dividing a number until the quotient is 1:

```
> 28 `quot` 3
9
> it `quot` 3
3
> it `quot` 3
1
```

(Note backquotes to use **quot** *as infix op.)*

(Remember that **it** *is previous result.)*

Problem: Write a recursive function **numDivs divisor x** that computes the number of times **x** must be divided by **divisor** to reach a quotient of 1.

```
> numDivs 3 28
3
> numDivs 2 7
2
```

Recursion, continued

A solution:

```
numDivs divisor x
    | (x `quot` divisor) < 1 = 0
    | otherwise =
                1 + numDivs divisor (x `quot` divisor)
```

Example:

```
> numDivs 3 28
3
```

What is its type?

```
numDivs :: (Integral a, Num a1) => a -> a -> a1
```

Will **numDivs 2 3.4** work?

```
> numDivs 2 3.4
<interactive>:93:1:
   No instance for (Integral a0) arising from a use of
`numDivs'
```

Sidebar: Fun with partial applications

Let's compute two partial applications of **numDivs**, using **let** to bind them to identifiers:

```
> let f = numDivs 2
> let g = numDivs 10
> f 9
3
> g 1001
3
```

What are more descriptive names than **f** and **g**?

```
> let floor_log2 = numDivs 2
> floor_log2 1000
9

> let floor_log10 = numDivs 10
> floor_log10 1000
3
```

Lists

List basics

In Haskell, a list is a sequence of values of the same type.

Here's one way to make a list. Note the type of **it** for each.

```
> [7, 3, 8]
[7,3,8]
it :: [Integer]

> [1.3, 10, 4, 9.7]  -- note mix of literals
[1.3,10.0,4.0,9.7]
it :: [Double]

> ['x', 10]
<interactive>:20:7:
   No instance for (Num Char) arising from the literal `10'
```

It is said that lists in Haskell are homogeneous.

List basics, continued

The function **length** returns the number of elements in a list:

```
> length [3,4,5]
3

> length []
0
```

What's the type of **length**?

```
> :type length
length :: [a] -> Int
```

With no class constraint specified, **[a]** indicates that **length** operates on lists containing elements of any type.

List basics, continued

The **head** function returns the first element of a list.

```
> head [3,4,5]
3
```

What's the type of **head**?

```
head :: [a] -> a
```

Here's what **tail** does. How would you describe it?

```
> tail [3,4,5]
[4,5]
```

What's the type of **tail**?

```
tail :: [a] -> [a]
```

Important: **head** and **tail** are good for learning about lists but we'll almost always use patterns to access list elements!

List basics, continued

The ++ operator concatenates two lists, producing a new list.

```
> [3,4] ++ [10,20,30]
[3,4,10,20,30]

> it ++ it
[3,4,10,20,30,3,4,10,20,30]

> let f = (++) [1,2,3]
> f [4,5]
[1,2,3,4,5]

> f [4,5] ++ reverse (f [4,5])
[1,2,3,4,5,5,4,3,2,1]
```

What are the types of ++ and reverse?

```
> :type (++)
(++) :: [a] -> [a] -> [a]

> :type reverse
reverse :: [a] -> [a]
```

List basics, continued

A range of values can be specified with a dot-dot notation:

```
> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
it :: [Integer]

> [-5,-3..20]
[-5,-3,-1,1,3,5,7,9,11,13,15,17,19]

> length [-1000..1000]
2001

> [10..5]
[]
it :: [Integer]
```

This is known as the *arithmetic sequence notation*, described in H10 3.10.

List basics, continued

The !! operator produces a list's Nth element, zero-based:

```
> :type (!!)
(!!) :: [a] -> Int -> a

> [10,20..100] !! 3
40
```

Sadly, we can't use a negative value to index from the right.

```
> [10,20..100] !! (-2)
*** Exception: Prelude.(!!): negative index
```

Should that be allowed?

Important: Extensive use of !! might indicate you're writing a Java program in Haskell!

Comparing lists

Haskell lists are values and can be compared as values:

```
> [3,4] == [1+2, 2*2]
True

> [3] ++ [] ++ [4] == [3,4]
True

> tail (tail [3,4,5,6]) == [last [4,5]] ++ [6]
True
```

Conceptually, how many lists are created by each of the above?

A programmer using a functional language writes complex expressions using lists (and more!) as freely as a Java programmer might write **`f(x) * a == g(a,b) + c`**.

Comparing lists, continued

Lists are compared *lexicographically*: Corresponding elements are compared until an inequality is found. The inequality determines the result of the comparison.

Example:

```
> [1,2,3] < [1,2,4]
True
```

Why: The first two elements are equal, and **3 < 4**.

More examples:

```
> [1,2,3] < [1,1,1,1]
False
> [1,2,3] > [1,2]
True
```

Lists of Lists

We can make lists of lists.

```
> let x = [[1], [2,3,4], [5,6]]
x :: [[Integer]]
```

Note the type: x is a list of Integer lists.

length counts elements at the top level.

```
> length x
3
```

Recall that length :: [a] -> Int Given that, what's the type of a for length x?

What's the value of length (x ++ x ++ [3])?

Lists of lists, continued

More examples:

```
> let x = [[1], [2,3,4], [5,6]]

> head x
[1]

 > tail x
[[2,3,4],[5,6]]

> x !! 1 !! 2
4

> head (head (tail (tail x)))
5
```

Strings are [Char]

Strings in Haskell are simply lists of characters.

```
> "testing"
"testing"
it :: [Char]

> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
it :: [Char]

> ["just", "a", "test"]
["just","a","test"]
it :: [[Char]]
```

What's the beauty of this?

Strings, continued

All list functions work on strings, too!

```
> let asciiLets = ['A'..'Z'] ++ ['a'..'z']
asciiLets :: [Char]

> length asciiLets
52

> reverse (drop 26 asciiLets)
"zyxwvutsrqponmlkjihgfedcba"

> :type elem
elem :: Eq a => a -> [a] -> Bool

> let isAsciiLet c = c `elem` asciiLets
isAsciiLet :: Char -> Bool
```

Strings, continued

The Prelude defines **String** as **[Char]** (a *type synonym*).

```
> :info String
type String = [Char]
```

A number of functions operate on **String**s. Here are two:

```
> :type words
words :: String -> [String]

> :type unwords
unwords :: [String] -> String
```

What's the following doing?

```
> unwords (tail (words "Just some words!"))
"some words!"
```

"cons" lists

Like most functional languages, Haskell's lists are "cons" lists.

A "cons" list has two parts:
 head: a value
 tail: a list of values (possibly empty)

The **:** ("cons") operator creates a list from a value and a list of values of that same type (or an empty list).

```
> 5 : [10, 20,30]
[5,10,20,30]
```

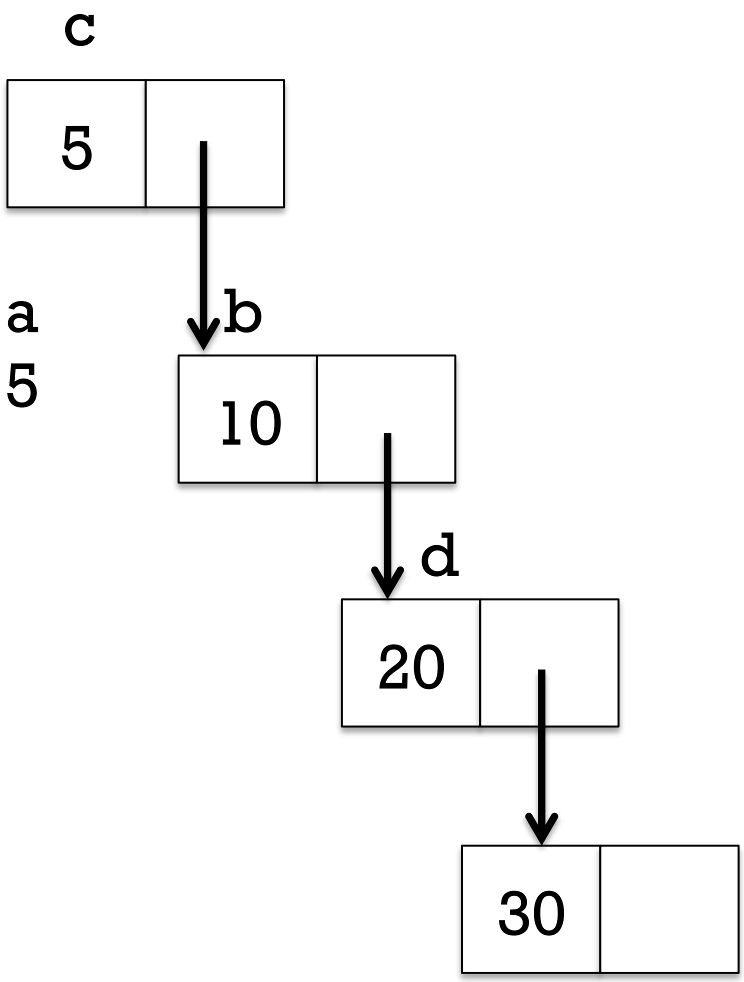
What's the type of the cons operator?

```
> :type (:)
(:) :: a -> [a] -> [a]
```

"cons" lists, continued

The cons (:) operation forms a new list from a value and a list.

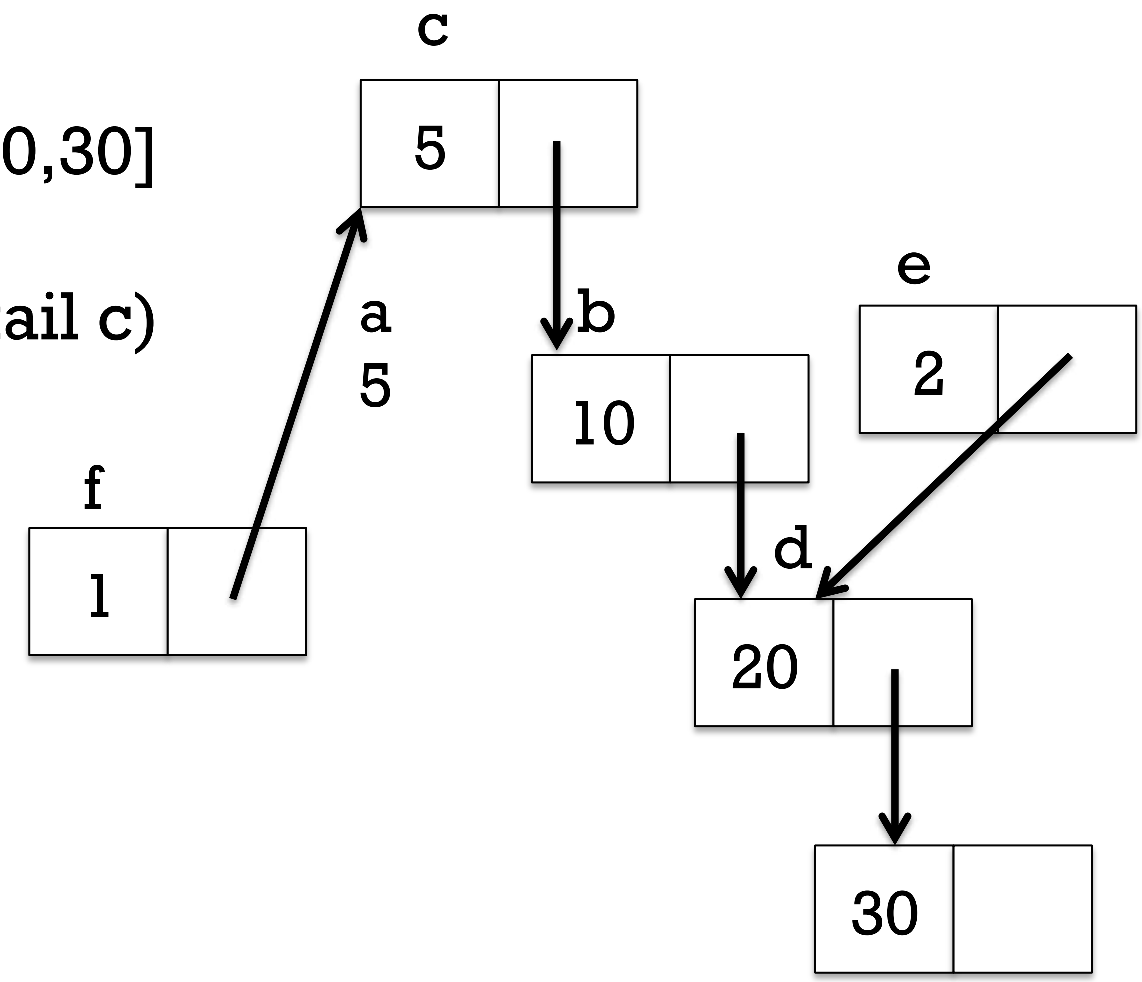
```
> let a = 5
> let b = [10,20,30]
> let c = a:b
[5,10,20,30]

> head c
5

> tail c
[10,20,30]

> let d = tail (tail c)
> d
[20,30]
```



"cons" lists, continued

A cons node can be referenced by multiple cons nodes.

```
> let a = 5
> let b = [10,20,30]
> let c = a:b
> let d = tail (tail c)
[20,30]

> let e=2:d
[2,20,30]

> let f=1:c
[1,5,10,20,30]
```



"cons" lists, continued

What are the values of the following expressions?

```
> 1:[2,3]
[1,2,3]

> 1:2
...error...

> chr 97:chr 98:chr 99:[]
"abc"

> []:[]
[[]]

> [1,2]:[]
[[1,2]]

> []:[1]
...error...
```

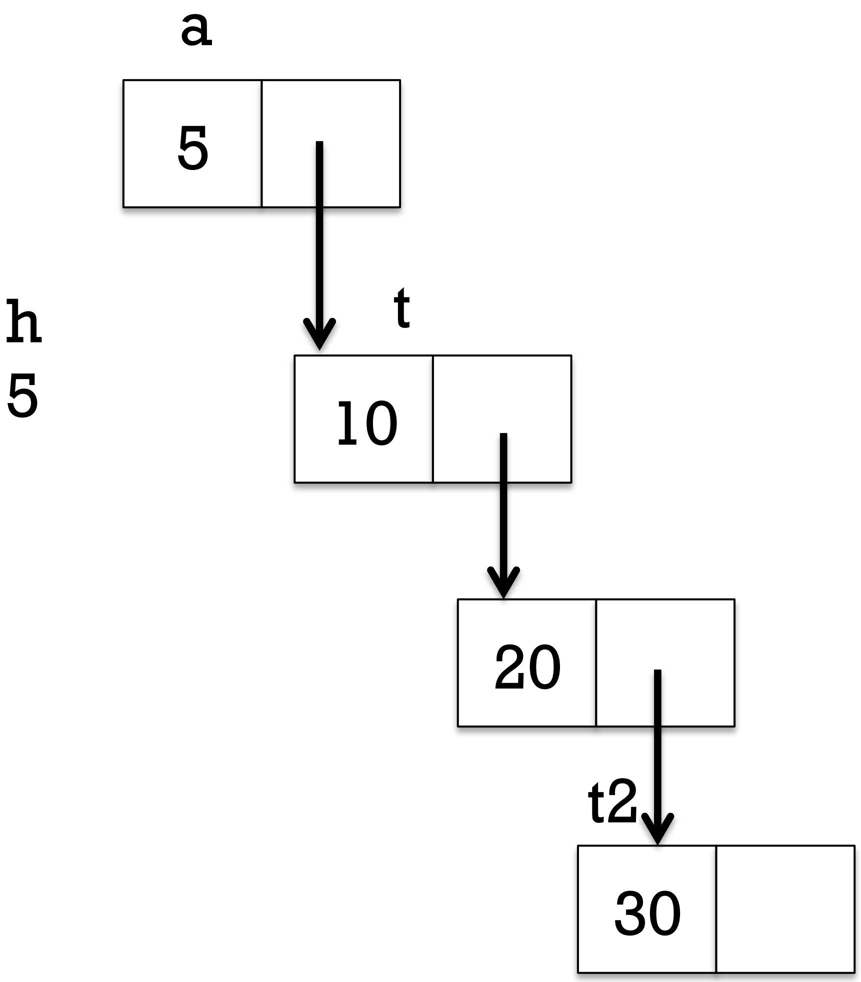
cons is right associative

```
chr 97:(chr 98:(chr 99:[]))
```

head and **tail** visually

It's important to understand that **tail** does not create a new list. Instead it simply returns an existing cons node.

```
> let a = [5,10,20,30]

> let h = head a
> h
5

> let t = tail a
> t
[10,20,30]

> let t2 = tail (tail t)
> t2
[30]
```



A little on performance

What operations are likely fast with cons lists?

- Get the head of a list
- Get the tail of a list
- Make a new list from a head and tail ("cons up a list")

What operations are likely slower?

- Get the Nth element of a list
- Get the length of a list

With cons lists, what does list concatenation involve?

```
> let m=[1..10000000]
> length (m++[0])
10000001
```

True or false?

The head of a list is a one-element list.

False, unless...

...it's the head of a list of lists that starts with a one-element list

The tail of a list is a list.

True

The tail of an empty list is an empty list.

It's an error!

`length (tail (tail x)) == (length x) – 2`

True (assuming what?)

A cons list is essentially a singly-linked list.

True

A doubly-linked list might help performance in some cases.

Hmm...what's the backlink for a multiply-referenced node?

Changing an element in a list might affect the value of many lists.

Trick question! We can't change a list element. We can only "cons up" new lists and reference existing lists.

fromTo

Here's a function that produces a list with a range of integers:

```
> let fromTo first last = [first..last]

> fromTo 10 15
[10,11,12,13,14,15]
```

Problem: Write a recursive version of **fromTo** that uses the cons operator to build up its result.

fromTo, continued

One solution:

```
fromTo first last
    | first > last = []
    | otherwise = first : fromTo (first+1) last
```

Evaluation of **fromTo 1 3** via substitution and rewriting:

$\underline{\texttt{fromTo 1 3}}$

$\texttt{1 : fromTo }\underline{\texttt{(1+1)}}\texttt{ 3}$

$\texttt{1 : }\underline{\texttt{fromTo 2 3}}$

$\texttt{1 : 2 : fromTo }\underline{\texttt{(2+1)}}\texttt{ 3}$

$\texttt{1 : 2 : }\underline{\texttt{fromTo 3 3}}$

$\texttt{1 : 2 : 3 : fromTo }\underline{\texttt{(3+1)}}\texttt{ 3}$

$\texttt{1 : 2 : 3 : }\underline{\texttt{fromTo 4 3}}$

$\texttt{1 : 2 : 3 : []}$

The **Enum** type class has **enumFromTo** and more.

fromTo, continued

Do **:set +s** to get timing and memory information, and make some lists. Try these:

```
fromTo 1 10
let f = fromTo        -- So we can type f instead of fromTo
f 1 1000
let f = fromTo 1      -- Note partial application
f 1000
let x = f 1000000
length x
take 5 (f 1000000)
```

List comprehensions

Here's a simple example of a *list comprehension*:

```
> [x^2 | x <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
```

This describes a list of the squares of **`x`** where **`x`** takes on each of the values from 1 through 10.

List comprehensions are very powerful but in the interest of time and staying focused on the core concepts of functional programming, we're not going to cover them.

Chapter 5 in Hutton has some very interesting examples of practical computations with list comprehensions.

A little output

A little output

The **putStr** function outputs a string:

```
> putStr "just\ntesting\n"
just
testing
```

Here's the type of **putStr**:

```
> :t putStr
putStr :: String -> IO ()
```

The return type of **putStr**, **IO ()**, is known as an *action*. It represents an interaction with the outside world, which is a side effect.

The construction **()** is read as "unit". The unit type has a single value, unit. Both the type and the value are written as **()**.

A little output, continued

For the time being, we'll use this approach for functions that produce output:

- A helper function will produce a ready-to-print string that contains newline characters as needed.

- The top-level function will call the helper function and then call **putStr** with the helper function's result.

A little output, continued

We can use **show** to produce a string representation of any value whose type is a member of the **Show** type class.

```
> :t show
show :: Show a => a -> String

> show 10
"10"

> show [10,20]
"[10,20]"

> show show
"<function>"
```

printN

Let's write a function to print the integers from 1 to N:

```
> printN 3
1
2
3
```

First, let's write a helper, **printN'**:

```
> printN' 3
"1\n2\n3\n"
```

Solution:

```
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"
```

printN, continued

At hand:

```
printN'::Integer -> String
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"
```

Usage:

```
> printN' 10
"1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n"
```

Let's write the top-level function:

```
printN::Integer -> IO ()
printN n = putStr (printN' n)
```

printN, continued

All together, as a file:

```
% cat printN.hs
printN::Integer -> IO ()
printN n = putStr (printN' n)

printN'::Integer -> String
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"

% ghci printN
...
> printN 3
1
2
3
```

printNunary

At hand:

```
printN::Integer -> IO ()
printN n = putStr (printN' n)

printN'::Integer -> String
printN' n
    | n == 0 = ""
    | otherwise = printN' (n-1) ++ show n ++ "\n"
```

Let's modify **printN** to print lines of characters:

```
> printN 3 '|'
|
||
|||
```

We can view it as printing *unary numbers* with a specified "digit", so we'll call the new version **printNunary**.

printNunary, continued

Useful: The Prelude has **replicate :: Int -> a -> [a]**

```
> replicate 3 7
[7,7,7]
> replicate 3 'a'
"aaa"
```

Let's add a parameter for the character to print and call **replicate** instead of **show**:

```
printNunary::Int -> Char -> IO ()
printNunary n c = putStr (printNunary' n c)

printNunary'::Int -> Char -> String
printNunary' n c
   | n == 0 = ""
   | otherwise = printNunary' (n-1) c ++
                        (replicate n c) ++ "\n"
```

charbox

Let's write **charbox**:

```
> charbox 5 3 '*'
*****
*****
*****

> :t charbox
charbox :: Int -> Int -> Char -> IO ()
```

How can we approach it?

charbox, continued

Let's work out a sequence of computations with ghci:

```
> replicate 5 '*'
"*****"


> it ++ "\n"
"*****\n"


> replicate 2 it
["*****\n","*****\n"]    -- the type of it is [[Char]]

> :t concat
concat :: [[a]] -> [a]

> concat it
"*****\n*****\n"

> putStr it
*****
*****
```

charbox, continued

Let's write charbox':

```
charbox'::Int -> Int -> Char -> String
charbox' w h c = concat (replicate h (replicate w c ++ "\n"))
```

Test:

```
> charbox' 3 2 '*'
"***\n***\n"
```

Now we're ready for the top-level function:

```
charbox::Int -> Int -> Char -> IO ()
charbox w h c = putStr (charbox' w h c)
```

How does this approach contrast with how we'd write it in Java?

Sidebar: Where's the code?

On the CS machines, selected Haskell code is in this directory:

/cs/www/classes/cs372/spring16/haskell

In these slides I'll refer to that directory as **spring16**.

spring16/slides.hs has the smaller functions, in rough chronological order. For functions that evolve, there may be multiple versions, with all but one version commented out.

Note: **{- ... -}** is a multi-line comment in Haskell.

Larger examples are in their own files, like **spring16/printN.hs** and **spring16/charbox.hs**.

spring16 is also accessible on the web:

http://cs.arizona.edu/classes/cs372/spring16

Patterns

Motivation: Summing list elements

Imagine a function that computes the sum of a list's elements.

```
> sumElems  [1..10]
55

> :type sumElems
sumElems :: Num a => [a] -> a
```

Implementation:

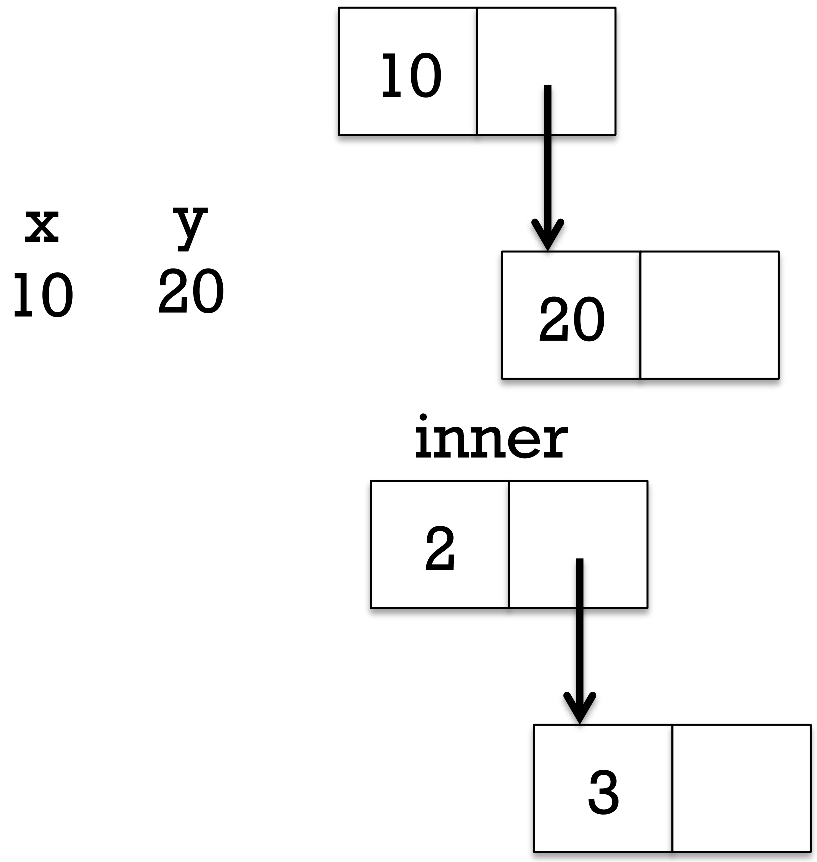
```
sumElems list
    | list == [] = 0
    | otherwise = head list + sumElems (tail list)
```

It works but it's not idiomatic Haskell. We should use *patterns* instead!

Patterns

In Haskell we can use *patterns* to bind names to elements of data structures.

```
> let [x,y] = [10,20]
> x
10
> y
20

> let [inner] = [[2,3]]
> inner
[2,3]
```

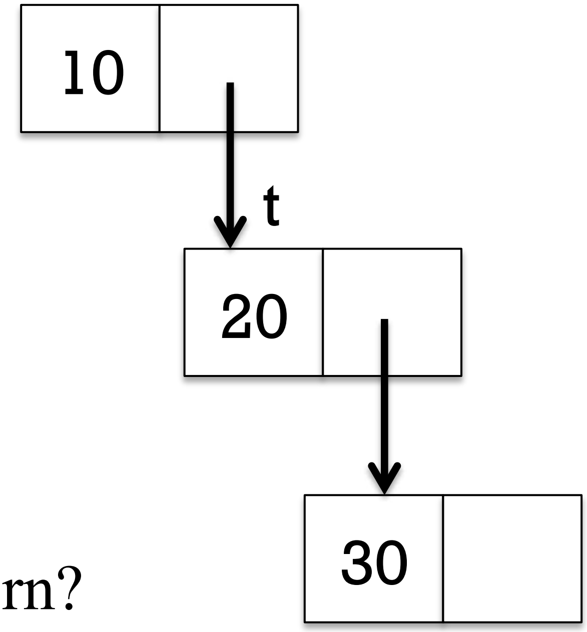


Speculate: Given a list like [10,20,30] how could we use a pattern to bind names to the head and tail of the list?

Patterns, continued

We can use the cons operator in a pattern.

```
> let h:t = [10,20,30]

> h
10

> t
[20,30]
```



What values get bound by the following pattern?

```
> let a:b:c:d = [10,20,30]
> [c,b,a]              -- in a list so I could show them w/ a one-liner
[30,20,10]

> d
[]                     -- Why didn't I do [d,c,b,a] above?
```

Patterns, continued

If some part of a structure is not of interest, we indicate that with an underscore, known as the *wildcard pattern*.

```
> let _:(a:[b]):c = [[1],[2,3],[4]]
> a
2
> b
3
> c
[[4]]
```

No binding is done for the wildcard pattern.

The pattern mechanism is completely general—patterns can be arbitrarily complex.

Patterns, continued

A name can only appear once in a pattern. This is invalid:

```
> let a:a:[] = [3,3]
<interactive>:25:5:
   Conflicting definitions for `a'
```

When using **let** as we are here, a failed pattern isn't manifested until we try to see what's bound to a name.

```
> let a:b:[] = [1]
> a
*** Exception: <interactive>:26:5-16: Irrefutable
pattern failed for pattern a : b : []
```

Practice

Describe in English what must be on the right hand side for a successful match.

`let (a:b:c) = ...`

A list containing at least two elements.

Does `[[1,2]]` match?

`[2,3]` ?

`"abc"` ?

`let [x:xs] = ...`

A list whose only element is a non-empty list.

Does `words "a test"` match?

`[words "a test"]` ?

`[[]]` ?

`[[[]]]` ?

Patterns in function definitions

Recall our non-idiomatic **sumElems**:

```
sumElems list
    | list == [] = 0
    | otherwise = head list + sumElems (tail list)
```

How could we redo it using patterns?

```
sumElems [] = 0
sumElems (h:t) = h + sumElems t
```

Note that **sumElems** appears on both lines and that there are no guards. **sumElems** has two *clauses*. (H10 4.4.3.1)

The parentheses in (h:t) are required!!

Do the types of the two versions differ?

```
(Eq a, Num a) => [a] -> a
        Num a => [a] -> a
```

Patterns in functions, continued

Here's a buggy version of **sumElems**:

```
buggySum [x] = x
buggySum (h:t) = h + buggySum t
```

What's the bug?

```
> buggySum [1..100]
5050
> buggySum []
*** Exception: slides.hs:(62,1)-(63,31): Non-
exhaustive patterns in function buggySum
```

Patterns in functions, continued

At hand:

```
buggySum [x] = x
buggySum (h:t) = h + buggySum t
```

If we use the **-fwarn-incomplete-patterns** option of **ghci**, we'll get a warning when loading:

```
% ghci -fwarn-incomplete-patterns buggySum.hs
buggySum.hs:1:1: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for ‘buggySum’: Patterns not matched: []
    >
```

Suggestion: add a bash alias! (See us if you don't know how to.)

```
alias ghci="ghci -fwarn-incomplete-patterns"
```

Patterns in functions, continued

What's a little silly about the following list-summing function?

```
sillySum [] = 0
sillySum [x] = x
sillySum (h:t) = h + sillySum t
```

The second clause isn't needed.

An "as pattern"

Consider a function that duplicates the head of a list:

```
> duphead [10,20,30]
[10,10,20,30]
```

Here's one way to write it, but it's repetitious:

```
duphead (x:xs) = x:x:xs
```

We can use an "as pattern" to bind a name to the list as a whole:

```
duphead all@(x:xs) = x:all
```

Can it be improved?

```
duphead all@(x:_) = x:all
```

The term "as pattern" perhaps comes from Standard ML, which uses an "as" keyword for the same purpose.

Patterns, then guards, then **if-else**

Good coding style in Haskell:
    Prefer patterns over guards
    Prefer guards over **if-else**

Patterns—first choice!

```
sumElems [] = 0
sumElems (h:t) = h + sumElems t
```

Guards—second choice...

```
sumElems list
    | list == [] = 0
    | otherwise = head list + sumElems (tail list)
```

if-else—third choice...

```
sumElems list =
   if list == [] then 0
   else head list + sumElems (tail list)
```

And, these comparisons imply that **list**'s type must in **Eq**!

Patterns, then guards, then if-else

Recall this example of guards:

```
weather temp | temp >= 80 = "Hot!"
             | temp >= 70 = "Nice"
             | otherwise = "Cold!"
```

Can we rewrite **weather** to have three clauses with patterns?

No.

The pattern mechanism doesn't provide a way to test ranges.

Design question: should patterns and guards be unified?

Revision: the general form of a function

We first saw this *general form* of a function definition:

name param1 param2 ... paramN = expression

Revision: A function may have one or more clauses, of this form:

function-name pattern1 pattern2 ... patternN

{ | *guard-expression1* } = *result-expression1*

...

{ | *guard-expressionN* } = *result-expressionN*

The set of clauses for a given name is the *binding* for that name. (See 4.4.3 in H10.)

If values in a call match the pattern(s) for a clause and a guard is true, the corresponding expression is evaluated.

Revision, continued

At hand, a more general form for functions:

```
function-name pattern1 pattern2 ... patternN
        { | guard-expression1 } = result-expression1
        ...
        { | guard-expressionN } = result-expressionN
```

How does

```
add x y = x + y
```

conform to the above specification?

- **x** and **y** are trivial patterns
- **add** has one clause, which has no guard

Pattern/guard interaction

If the patterns of a clause match but all guards fail, the next clause is tried. Here's a contrived example:

```
f (h:_) | h < 0 = "negative head"
f list | length list > 3 = "too long"
f (_:_) = "ok"
f [] = "empty"
```

Usage:

```
> f [-1,2,3]
"negative head"

> f []
"empty"

> f [1..10]
"too long"
```

How many clauses does **f** have?

4

What if 2nd and 3rd clauses swapped?

3rd clause would never be matched!

What if 4th clause is removed?

Warning with **-fwarn-incomplete-patterns**; "non-exhaustive patterns" exception on **f []**.

Recursive functions on lists

Simple recursive list processing functions

Problem: Write **`len x`**, which returns the length of list **`x`**.

```
> len []
0

> len "testing"
7
```

Solution:

```
len [] = 0
len (_:t) = 1 + len t  -- since head isn't needed, use _
```

Simple list functions, continued

Problem: Write **odds x**, which returns a list having only the odd numbers from the list **x**.

```
> odds [1..10]
[1,3,5,7,9]

> take 10 (odds [1,4..100])
[1,7,13,19,25,31,37,43,49,55]
```

Handy: **odd :: Integral a => a -> Bool**

Solution:

```
odds [] = []
odds (h:t)
    | odd h = h:odds t
    | otherwise = odds t
```

Simple list functions, continued

Problem: write **isElem x vals**, like **elem** in the Prelude.

```
> isElem 5 [4,3,7]
False

> isElem 'n' "Bingo!"
True

> "quiz" `isElem` words "No quiz today!"
True
```

Solution:

```
isElem _ [] = False     -- Why a wildcard?
isElem x (h:t)
    | x == h = True
    | otherwise = x `isElem` t
```

Simple list functions, continued

Problem: write a function that returns a list's maximum value.

```
> maxVal "maximum"
'x'

> maxVal [3,7,2]
7

> maxVal (words "i luv this stuff")
"this"
```

Note that the Prelude has **max :: Ord a => a -> a -> a**

One solution:

```
maxVal [x] = x
maxVal (x:xs) = max x (maxVal xs)
maxVal [] = error "empty list"
```

Sidebar: C and Python challenges

C programmers: Write **strlen** in C in a functional style. Do **strcmp** and **strchr**, too!

Python programmers: In a functional style write **size(x)**, which returns the number of elements in the string or list **x**. Restriction: You may not use **type()**.

Tuples

Tuples

A Haskell *tuple* is an ordered aggregation of two or more values of possibly differing types.

```
> (1, "two", 3.0)
(1,"two",3.0)
it :: (Integer, [Char], Double)

> (3 < 4, it)
(True,(1,"two",3.0))
it :: (Bool, (Integer, [Char], Double))
```

What's something we can represent with a tuple that we can't represent with a list?

We can't create analogous lists for the above tuples, due to the mix of types. Lists must be homogeneous.

Tuples, continued

A function can return a tuple:

```
> let pair x y = (x,y)
```

What's the type of **pair**?

```
pair :: t -> t1 -> (t, t1)
```

-- why not a -> b -> (a,b)?

Let's play...

```
> pair 3 4
(3,4)

> pair (3,4)
<function>

> it 5
((3,4),5)
```

Tuples, continued

The Prelude has two functions that operate on 2-tuples.

```
> let p = pair 30 "forty"
p :: (Integer, [Char])

> p
(30,"forty")

> fst p
30

> snd p
"forty"
```

Tuples, continued

Recall: patterns used to bind names to list elements have the same syntax as expressions to create lists.

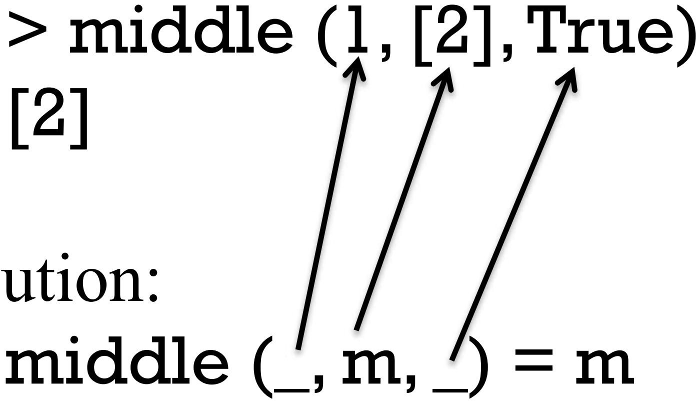
Patterns for tuples are like that, too.

Problem: Write **middle**, to extract a 3-tuple's second element.

```
> middle ("372", "BIOW 208", "Mitchell")
"BIOW 208"

> middle (1, [2], True)
[2]
```

Tuples, continued

At hand:



```
> middle (1, [2], True)
[2]
```

Solution:

```
middle (_, m, _) = m
```

What's the type of **middle?**

```
middle :: (t, t1, t2) -> t1
```

Does the following call work?

```
> middle(1,[(2,3)],4)
[(2,3)]
```

Tuples, continued

Here's the type of **zip** from the Prelude:

```
zip :: [a] -> [b] -> [(a, b)]
```

Speculate: What does **zip** do?

```
> zip ["one","two","three"] [10,20,30]
[("one",10),("two",20),("three",30)]

> zip ['a'..'z'] [1..]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6),('g',7),('h',8),('i',
9),('j',10), ...more..., ('x',24),('y',25),('z',26)]
```

What's especially interesting about the second example?

`[1..]` is an infinite list! **zip** stops when either list runs out.

Tuples, continued

Problem: Write **elemPos**, which returns the zero-based position of a value in a list, or -1 if not found.

```
> elemPos 'm' ['a'..'z']
12
```

Hint: Have a helper function do most of the work.

Solution:

```
elemPos x vals = elemPos' x (zip vals [0..])

elemPos' _ [] = -1
elemPos' x ((val,pos):vps)
    | x == val = pos
    | otherwise = elemPos' x vps
```

Sidebar: To curry or not to curry?

Consider these two functions:

```
> let add_c x y = x + y      -- _c for curried arguments
add_c :: Num a => a -> a -> a

> let add_t (x,y) = x + y    -- _t for tuple argument
add_t :: Num a => (a, a) -> a
```

Important: Note the difference in types!

Usage:

```
> add_c 3 4
7

> add_t (3,4)
7
```

Which is better, **add_c** or **add_t**?

The **Eq** type class and tuples

:info Eq shows many lines like this:

```
    ...
    instance (Eq a, Eq b, Eq c, Eq d, Eq e) => Eq (a, b, c, d, e)
    instance (Eq a, Eq b, Eq c, Eq d) => Eq (a, b, c, d)
    instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
    instance (Eq a, Eq b) => Eq (a, b)
```

We haven't talked about **instance** declarations but let's speculate: What's being specified by the above?

```
instance (Eq a, Eq b, Eq c) => Eq (a, b, c)
```

If values of each of the three types **a**, **b**, and **c** can be tested for equality then 3-tuples of type **(a, b, c)** can be tested for equality.

The **Ord** and **Bounded** type classes have similar instance declarations.

Lists vs. tuples

Type-wise, lists are homogeneous; tuples are heterogeneous.

We can write a function that handles a list of any length but a function that operates on a tuple specifies the arity of that tuple.

> Example: we can't write an analog for **`head`**, to return the first element of an arbitrary tuple.

Even if values are homogeneous, using a tuple lets static type-checking ensure that an exact number of values is being aggregated.

> Example: A 3D point could be represented with a 3-element list but using a 3-tuple <u>guarantees</u> points have three coordinates.

If there were *Head First Haskell* it would no doubt have an interview with List and Tuple, each arguing their own merit.

More on patterns and functions

Literals in patterns

Literal values can be part or all of a pattern. Here's a 3-clause binding for **f**:

```
f 1 = 10
f 2 = 20
f n = n
```

For contrast, with guards:

```
f n
  | n == 1 = 10
  | n == 2 = 20
  | otherwise = n
```

Usage:

```
> f 1
10

> f 3
3
```

Remember: Patterns are tried in the order specified.

Literals in patterns, continued

Here's a function that classifies characters as parentheses (or not):

```
parens c
    | c == '(' = "left"
    | c == ')' = "right"
    | otherwise = "neither"
```

Could we improve it by using patterns instead of guards?

```
parens '(' = "left"
parens ')' = "right"
parens _  = "neither"
```

Which is better?

Remember: Patterns, then guards, then **if-else**.

Literals in patterns, continued

not is a function:

```
> :type not
not :: Bool -> Bool

> not True
False
```

Problem: Using literals in patterns, define **not**.

Solution:

```
not True = False
not _ = True            -- Using wildcard avoids comparison
```

Pattern construction

A pattern can be:

- A literal value such as 1, '**x**', or **True**
- An identifier (bound to a value if there's a match)
- An underscore (the wildcard pattern)
- A tuple composed of patterns
- A list of patterns in square brackets (fixed size list)
- A list of patterns constructed with **:** operators
- Other things we haven't seen yet

Note the recursion.

Patterns can be arbitrarily complex.

3.17.1 in H10 shows the full syntax for patterns.

The **where** clause for functions

Intermediate values and/or helper functions can be defined using an optional **where** clause for a function.

Here's an example to show the syntax; the computation is not meaningful.

```
f x
   | g x < 0 = g a + g b
   | a > b = g b
   | otherwise = g a * g b
  where {
      a = x * 5;
      b = a * 2 + x;
      g t = log t + a
      }
```

The names **a** and **b** are bound to expressions; **g** is a function binding.

The bindings in the **where** clause are done first (!), then the guards are evaluated in turn.

Like variables defined in a method or block in Java, **a**, **b**, and **g** are not visible outside the the function **f**.

The *layout rule* for **where** (and more)

This is a valid declaration with a **where** clause:

```
f x = a + b + g a where { a = 1; b = 2; g x = -x }
```

The **where** clause has three declarations enclosed in braces and separated by semicolons.

We can take advantage of the *layout rule* and write it like this instead:

```
f x = a + b + g a
    where
        a = 1
        b = 2
        g x =
              -x
```

Besides whitespace what's different about the second version?

The layout rule, continued

At hand:

```
f x = a + b + g a
   where
      a = 1
      b = 2
      g x =
          -x
```

Another example:

```
f x = a + b + g a where a = 1
                        b = 2
                        g x =
                           -x
```

The <u>absence of a brace</u> after **where** activates the layout rule.

The column position of the <u>first token after **where**</u> establishes the column in which declarations of the **where** must start.

Note that the declaration of **g** is continued onto a second line; if the minus sign were at or left of the line, it would be an error.

The layout rule, continued

Don't confuse the layout rule with indentation-based continuation of declarations! (See slides 94-95.)

The layout rule allows omission of braces and semicolons in **where**, **do**, **let**, and **of** blocks. (We'll see **do** and **let** later.)

Indentation-based continuation applies

1. outside of **where/do/let/of** blocks
2. inside **where/do/let/of** blocks when the layout rule is triggered by the absence of an opening brace.

The layout rule is also called the "off-side rule".

TAB characters are assumed to have a width of 8.

What other languages have rules of a similar nature?

countEO

Imagine a function that counts occurrences of even and odd numbers in a list.

```
> countEO [3,4,5]
(1,2)                       -- one even, two odds
```

Code:

```
countEO [] = (0,0)          -- no odds or evens in []
countEO (x:xs)
    | odd x = (evens, odds+1)
    | otherwise = (evens+1, odds)
  where
    (evens, odds) = countEO xs   -- do counts for tail first!
```

Would it be awkward to write it without using **where**?

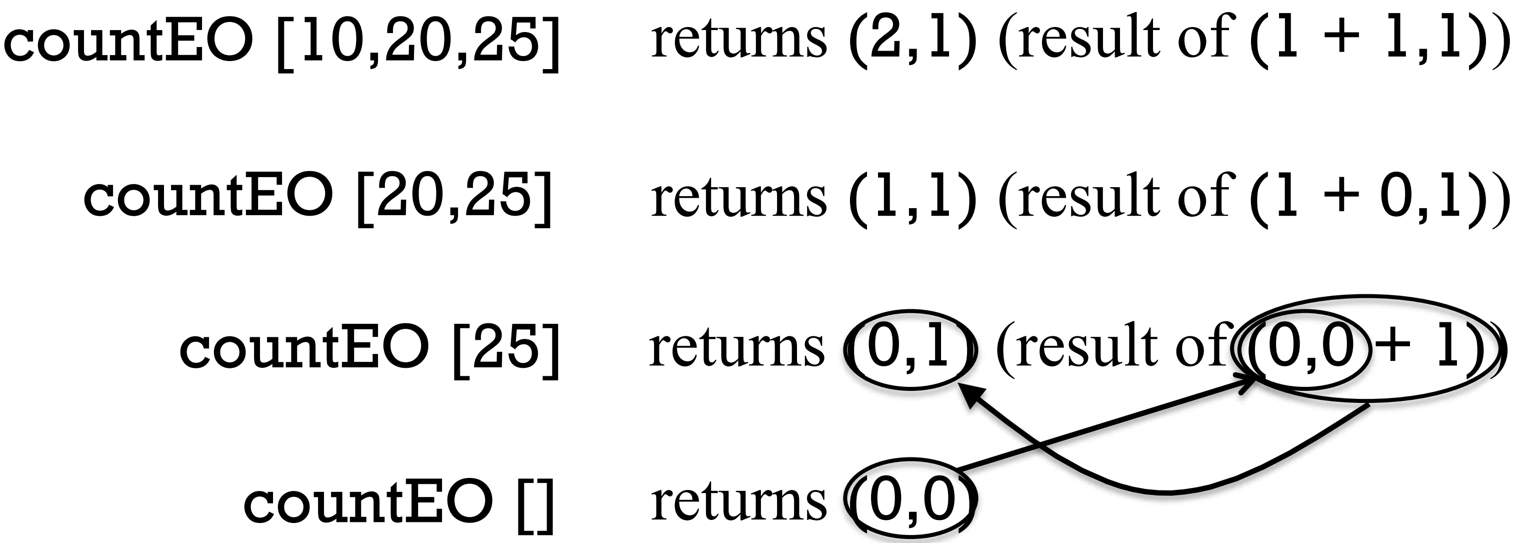
Try it!

countEO, continued

At hand:

```
countEO [] = (0,0)
countEO (x:xs)
    | odd x = (evens, odds + 1)
    | otherwise = (1+ evens, odds)
  where (evens, odds) = countEO xs
```

Here's one way to picture this recursion:



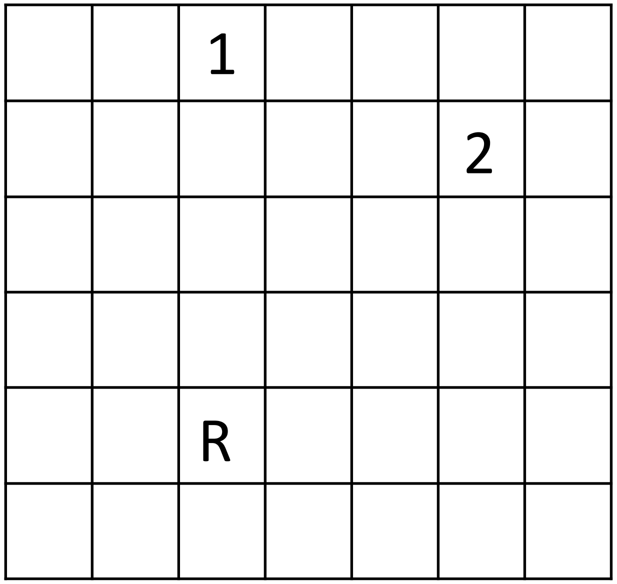
```
countEO [10,20,25]     returns (2,1) (result of (1 + 1,1))

    countEO [20,25]     returns (1,1) (result of (1 + 0,1))

        countEO [25]     returns (0,1) (result of (0,0 + 1))

            countEO []     returns (0,0)
```

Larger examples

travel

Imagine a robot that travels on an infinite grid of cells. Movement is directed by a series of one character commands: **n**, **e**, **s**, and **w**.

Let's write a function **travel** that moves the robot about the grid and determines if the robot ends up where it started (i.e., it got home) or elsewhere (it got lost).



If the robot starts in square R the command string **nnnn** leaves the robot in the square marked 1.

The string **nenene** leaves the robot in the square marked 2.

nnessw and **news** move the robot in a round-trip that returns it to square R.

travel, continued

Usage:

```
> travel "nnnn"         -- ends at 1
"Got lost; 4 from home"

> travel "nenene"      -- ends at 2
"Got lost; 6 from home"

> travel "nnessw"
"Got home"
```

		1				
					2	
		R				

How can we approach this problem?

travel, continued

One approach:

1. Map letters into integer 2-tuples representing X and Y displacements on a Cartesian plane.
2. Sum the X and Y displacements to yield a net displacement.

Example:

Argument value: "`nnee`"

Mapped to tuples: `(0,1) (0,1) (1,0) (1,0)`

Sum of tuples: `(2,2)`

Another:

Argument value: "`nnessw`"

Mapped to tuples: `(0,1) (0,1) (1,0) (0,-1) (0,-1) (-1,0)`

Sum of tuples: `(0,0)`

travel, continued

First, let's write a helper function to turn a direction into an **(x,y)** displacement:

```
mapMove :: Char -> (Int, Int)
mapMove 'n' = (0,1)
mapMove 's' = (0,-1)
mapMove 'e' = (1,0)
mapMove 'w' = (-1,0)
mapMove c = error ("Unknown direction: " ++ [c])
```

Missing case found with **ghci -fwarn-incomplete-patterns**

Usage:

```
> mapMove 'n'
(0,1)

> mapMove 'w'
(-1,0)
```

travel, continued

Next, a function to sum **x** and **y** displacements in a list of tuples:

```
> sumTuples [(0,1),(1,0)]
(1,1)

> sumTuples [mapMove 'n', mapMove 'w']
(-1,1)
```

Implementation:

```
sumTuples :: [(Int,Int)] -> (Int,Int)
sumTuples [] = (0,0)
sumTuples ((x,y):ts) = (x + sumX, y + sumY)
   where
      (sumX, sumY) = sumTuples ts
```

travel, continued

travel itself:

```
travel :: [Char] -> [Char]
travel s
   | disp == (0,0) = "Got home"
   | otherwise = "Got lost; " ++ show (abs x + abs y) ++
                    " from home"
   where
      tuples = makeTuples s
      disp@(x,y) = sumTuples tuples -- note "as pattern"

      makeTuples :: [Char] -> [(Int, Int)]
      makeTuples [] = []
      makeTuples (c:cs) = mapMove c : makeTuples cs
```

As is, **mapMove** and **sumTuples** are at the top level but **makeTuples** is hidden inside **travel**. How should they be arranged?

Sidebar: top-level vs. hidden functions

```
travel s
  | disp == (0,0) = "Got home"
  | otherwise = "Got lost; " ...
  where
    tuples = makeTuples s
    disp = sumTuples tuples

    makeTuples [] = []
    makeTuples (c:cs) =
        mapMove c:makeTuples cs

    mapMove 'n' = (0,1)
    mapMove 's' = (0,-1)
    mapMove 'e' = (1,0)
    mapMove 'w' = (-1,0)
    mapMove c = error ...

    sumTuples [] = (0,0)
    sumTuples ((x,y):ts) = (x + sumX, y + sumY)
       where
          (sumX, sumY) = sumTuples ts
```

Top-level functions can be tested after code is loaded but functions inside a **where** block are not visible.

The functions at left are hidden in the **where** block but they can easily be changed to top-level using a shift or two with an editor.

Note: Types are not shown, to save space.

tally

Consider a function **tally** that counts character occurrences in a string:

```
> tally "a bean bag"
a 3
b 2
  2
g 1
n 1
e 1
```

Note that the characters are shown in order of decreasing frequency.

How can this problem be approached?

In a nutshell: `[('a',3),('b',2),(' ',2),('g',1),('n',1),('e',1)]`

tally, continued

Let's start by writing **incEntry c tuples**, which takes a list of (*character, count*) tuples and produces a new list of tuples that reflects the addition of the character **c**.

```
incEntry :: Char -> [(Char, Int)] -> [(Char, Int)]
```

Calls to **incEntry** with **'t', 'o', 'o'**:

```
> incEntry 't' []
[('t',1)]

> incEntry 'o' it
[('t',1),('o',1)]

> incEntry 'o' it
[('t',1),('o',2)]
```

```
{-  incEntry c tups

    tups is a list of (Char, Int) tuples that indicate how many
    times a character has been seen.  A possible value for tups:
        [('b',1),('a',2)]

    incEntry produces a copy of tups with the count in the tuple
    containing the character c incremented by one.

    If no tuple with c exists, one is created with a count of 1.
-}

incEntry::Char -> [(Char,Int)] -> [(Char,Int)]
incEntry c [ ] = [(c, 1)]
incEntry c ((char, count):entries)
  | c == char = (char, count+1) :  entries
  | otherwise = (char, count) :  incEntry c entries
```

Next, let's write **mkentries s**. It calls **incEntry** for each character in the string **s** in turn and produces a list of (*char, count*) tuples.

```
mkentries :: [Char] -> [(Char, Int)]
```

Usage:

```
> mkentries "tupple"
[('t',1),('u',1),('p',2),('l',1),('e',1)]

> mkentries "cocoon"
[('c',2),('o',3),('n',1)]
```

Code:

```
mkentries :: [Char] -> [(Char, Int)]
mkentries s = mkentries' s []
   where
      mkentries' [ ] entries = entries
      mkentries' (c:cs) entries =
         mkentries' cs (incEntry c entries)
```

```
{-  insert, isOrdered, and sort  provide an insertion sort -}
insert v [ ] = [v]
insert v (x:xs)
    | isOrdered (v,x) = v:x:xs
    | otherwise =  x:insert v xs

isOrdered ((_, v1), (_, v2)) = v1 > v2

sort [] = []
sort (x:xs) = insert x (sort xs)

> mkentries "cocoon"
[('c',2),('o',3),('n',1)]

> sort it
[('o',3),('c',2),('n',1)]
```

tally, continued

```
{- fmtEntries prints (char,count) tuples one per line -}
fmtEntries [] = ""
fmtEntries ((c, count):es) =
      [c] ++ " " ++ (show count) ++ "\n" ++ fmtEntries es

{- top-level function -}
tally s = putStr (fmtEntries (sort (mkentries s)))

> tally "cocoon"
o 3
c 2
n 1
```

- How does this solution exemplify functional programming? (slide 24)
- How is it like procedural programming (slide 5)

Running **tally** from the command line

Let's run it on lectura...

```
% code=/cs/www/classes/cs372/spring16/haskell

% cat $code/tally.hs
... everything we've seen before and now a main:
main = do
   bytes <- getContents  -- reads all of standard input
   tally bytes

% echo -n cocoon | runghc $code/tally.hs
o 3
c 2
n 1
```

tally from the command line, continued

$code/genchars N generates N random letters:

```
% $code/genchars 20
KVQaVPEmClHRbgdkmMsQ
```

Lets tally a million letters:

```
 % $code/genchars 1000000 |
         time runghc $code/tally.hs >out
21.79user 0.24system 0:22.06elapsed
% head -3 out
s  19553
V 19448
 J  19437
```

tally from the command line, continued

Let's try a compiled executable.

```
% cd $code
% ghc --make -rtsopts tally.hs
% ls -l tally
-rwxrwxr-x 1 whm whm 1118828 Jan 26 00:54 tally

% ./genchars 1000000 > 1m
% time ./tally +RTS -K40000000 -RTS < 1m > out
real   0m7.367s
user   0m7.260s
sys    0m0.076s
```

tally performance in other languages

Here are user CPU times for implementations of **tally** in several languages. The same one-million letter file was used for all timings.

Language	Time (seconds)
Haskell	7.260
Ruby	0.548
Icon	0.432
Python 2	0.256
C w/ `gcc -O3`	0.016

However, our **tally** implementation is very simplistic. An implementation of **tally** by an expert Haskell programmer, Chris van Horne, ran in 0.008 seconds. (See **spring16/haskell/tally-cwvh[12].hs**.)

Then I revisited the C version (**tally2.c**) and got to 3x faster than Chris' version with a one-billion character file.

Real world problem: "How many lectures?"

Here's an early question when planning a course for a semester:

"How many lectures will there be?"

How should <u>we</u> answer that question?

Google for a course planning app?

No! Let's write a Haskell program! ☺

classdays

One approach:

```
> classdays  ...arguments...
#1 H 1/15        (for 2015...)
#2 T 1/20
#3 H 1/22
#4 T 1/27
#5 H 1/29
...
```

What information do the arguments need to specify?

First and last day

Pattern, like M-W-F or T-H

How about holidays?

Arguments for **classdays**

Let's start with something simple:

```
> classdays  (1,15)  (5,6)  [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
#3 H 1/22
#4 T 1/27
...
#32 T 5/5
>
```

The first and last days are represented with (*month*,*day*) tuples.

The third argument shows the pattern of class days: the first is a Thursday, and it's five days to the next class. The next is a Tuesday, and it's two days to the next class. Repeat!

Date handling

There's a **Data.Time.Calendar** module but writing two minimal date handling functions provides good practice.

```
> toOrdinal (12,31)
365    -- 12/31 is the last day of the year

> fromOrdinal 32
(2,1)  -- The 32nd day of the year is February 1.
```

What's a minimal data structure that could help us?

```
[(0,0),(1,31),(2,59),(3,90),(4,120),(5,151),(6,181),
(7,212),(8,243),(9,273),(10,304),(11,334),(12,365)]
```

(1,31) *The last day in January is the 31st day of the year*

(7,212) *The last day in July is the 212th day of the year*

toOrdinal and fromOrdinal

```
offsets = [(0,0),(1,31),(2,59),(3,90),(4,120),(5,151),(6,181),
(7,212),(8,243),(9,273),(10,304),(11,334),(12,365)]

toOrdinal (month, day) = days + day
   where
      (_,days) = offsets!!(month-1)

fromOrdinal ordDay =
      fromOrdinal' (reverse offsets) ordDay
   where
      fromOrdinal' ((month,lastDay):t) ordDay
         | ordDay > lastDay = (month + 1, ordDay - lastDay)
         | otherwise = fromOrdinal' t ordDay
      fromOrdinal' [] _ = error "invalid month?"
```

```
> toOrdinal (12,31)
365
```

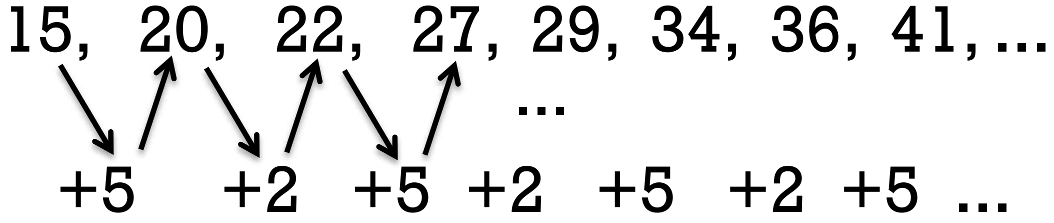
```
> fromOrdinal 32
(2,1)
```

Recall:

```
> classdays  (1,15)  (5,6)  [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
...
```

Ordinal dates for (1,15) and (5,6) are 15 and 126, respectively.

With the Thursday-Tuesday pattern we'd see the ordinal dates progressing like this:



15, 20, 22, 27, 29, 34, 36, 41, ...

...

+5 +2 +5 +2 +5 +2 +5 ...

Imagine this series of calls to a helper, **showLecture**:

```
showLecture 1 15  'H'
showLecture 2  20 'T'
showLecture 3  22 'H'
showLecture 4  27  'T'
...
showLecture 32 125 'T'
```

Desired output:

```
#1 H 1/15
#2 T 1/20
#3 H 1/22
#4 T 1/27
...
#32 T 5/5
```

What computations do we need to transform

```
showLecture 1 15  'H'
```

into

```
"#1 H 1/15\n"
```

?

We have: **showLecture 1 15 'H'**

We want: **"#1 H 1/15"**

1 is lecture #1; 15 is 15th day of year

Let's write **showOrdinal :: Integer -> [Char]**

```
> showOrdinal 15
"1/15"
```

```
showOrdinal ordDay = show month ++ "/" ++ show day
   where
      (month,day) = fromOrdinal ordDay
```

Now we can write **showLecture**:

```
showLecture lecNum ordDay dayOfWeek =
      "#" ++ show lecNum ++ " " ++ [dayOfWeek] ++
       " " ++ showOrdinal ordDay ++ "\n"
```

Recall:

```
showLecture 1 15  'H'
showLecture 2  20 'T'
...
showLecture 32 125 'T'
```

Desired output:

```
#1 H 1/15
#2 T 1/20
...
#32 T 5/5
```

Let's "cons up" a list out of the results of those calls...

```
> showLecture 1 15 'H' :
   showLecture 2 20 'T' :
   "...more..." :  -- I literally typed "...more..."
   showLecture 32 125 'T' : []
["#1 H 1/15\n","#2 T 1/20\n", "...more...","#32 T
5/5\n"]
```

How close are the contents of that list to what we need?

Now lets imagine a recursive function **showLectures** that builds up a list of results from **showLecture** calls:

```
    showLectures 1 15 126 [('H',5),('T',2)]             "#1 H 1/15\n"
       showLectures 2 20 126 [('T',2),('H',5)]          "#2 T 1/20\n"
            ...
          showLectures 32 125 126 [('T',2),('H',5)]  "#32 T 5/5\n"
            showLectures 33 127 126 [('H',5),('T',2)]
```

Result:

```
    ["#1 H 1/15\n","#2 T 1/20\n", ... ,"#33 H 5/5\n"]
```

Now let's write **showLectures:**

```
  showLectures  lecNum thisDay lastDay
                    (pair@(dayOfWeek, daysToNext):pairs)
     | thisDay > lastDay = []
     | otherwise =  showLecture lecNum thisDay dayOfWeek
       :  showLectures (lecNum+1) (thisDay + daysToNext)
                     lastDay (pairs ++ [pair])
```

classdays—top-level

Finally, a top-level function to get the ball rolling:

```
classdays first last pattern = putStr (concat result)
   where
      result =
         showLectures 1 (toOrdinal first) (toOrdinal last) pattern
```

Usage:

```
> classdays (1,15) (5,6) [('H',5),('T',2)]
#1 H 1/15
#2 T 1/20
#3 H 1/22
...
#31 H 4/30
#32 T 5/5
```

Full source is in **spring16/haskell/classdays.hs**

Note: next set of slides!

Errors

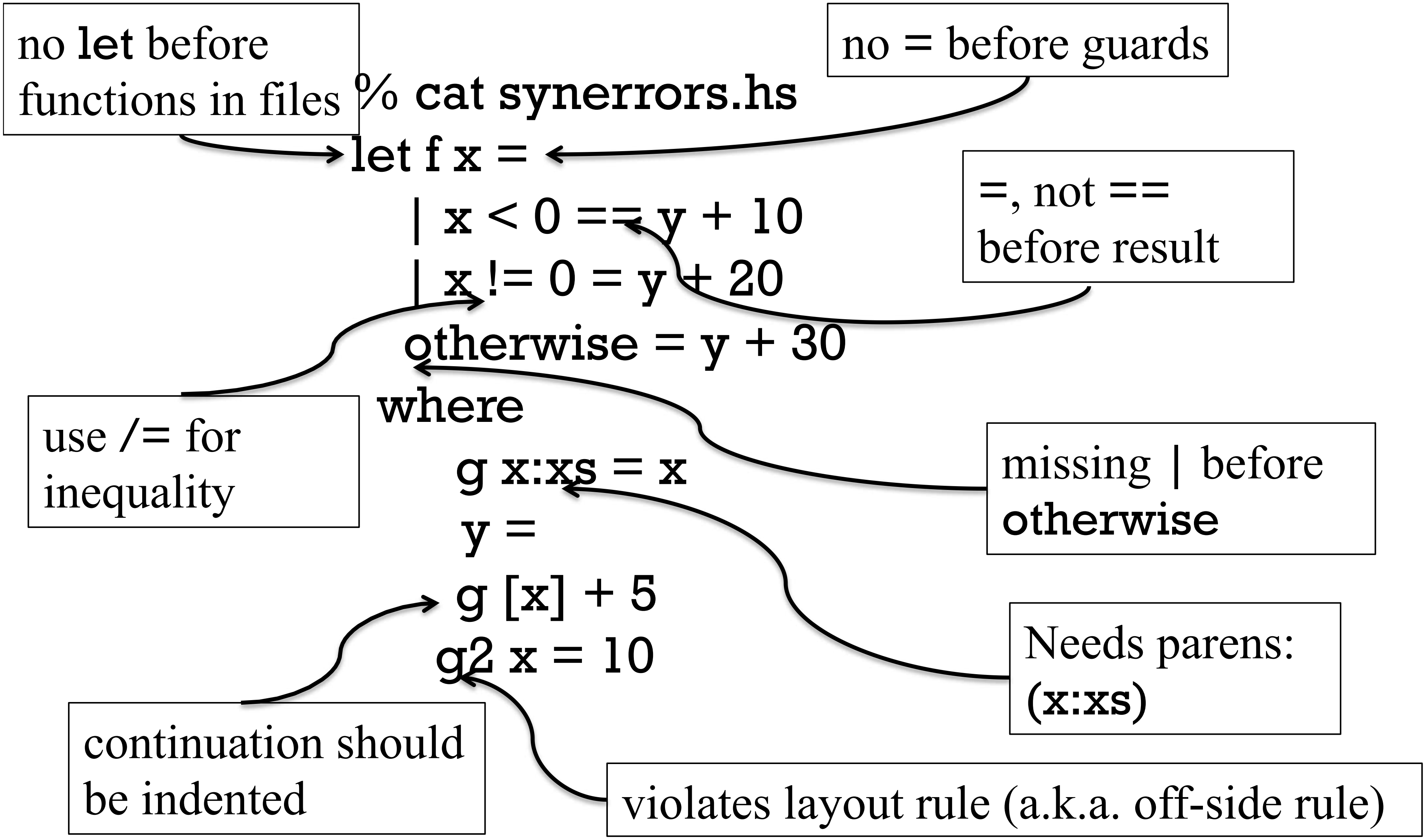
Syntax errors

What syntax errors do you see in the following file?

```
% cat synerrors.hs
let f x =
    | x < 0 == y + 10
    | x != 0 = y + 20
    otherwise = y + 30
  where
       g x:xs = x
        y =
       g [x] + 5
      g2 x = 10
```

Syntax errors, continued

What syntax errors do you see in the following file?



```
% cat synerrors.hs
let f x =
   | x < 0 == y + 10
   | x != 0 = y + 20
   otherwise = y + 30
 where
     g x:xs = x
     y =
     g [x] + 5
    g2 x = 10
```

no **let** before functions in files

no = before guards

=, not == before result

use /= for inequality

missing | before **otherwise**

Needs parens: **(x:xs)**

continuation should be indented

violates layout rule (a.k.a. off-side rule)

Type errors

In my opinion, producing understandable messages for type errors is what ghci is worst at.

If only concrete types are involved, type errors are typically easy to understand.

```
> chr 'x'
   Couldn't match expected type `Int' with actual
     type `Char'
    In the first argument of `chr', namely 'x'
    In the expression: chr 'x'
    In an equation for `it': it = chr 'x'

> :type chr
chr :: Int -> Char
```

Type errors, continued

Code and error:

```
f x y
    | x == 0 = []
    | otherwise = f x
```

```
 Couldn't match expected type `[al]' with actual type
`t0 -> [al]'
   In the return type of a call of `f'
   Probable cause: `f' is applied to too few arguments
   In the expression: f x
```

The error message is perfect in this case.

The first clause implies that **f** returns a list but the second clause returns a partial application, of type `t0 -> [al]`, a contradiction.

Type errors, continued

Code:

```
countEO (x:xs)
    | odd x = (evens, odds+1)
    | otherwise = (evens+1, odds)
   where (evens,odds) = countEO
```

Error:

```
Couldn't match expected type `(t3, t4)'
            with actual type `[t0] -> (t1, t2)'
   In the expression: countEO
   In a pattern binding: (evens, odds) = countEO
```

What's the problem?

It's expecting a tuple, (t3,t4) but it's getting a function, [t0] -> (t1,t2)

Typically, instead of getting errors about too few (or too many) function arguments, you get function types popping up in unexpected places.

Type errors, continued

Is there an error in the following?

```
f [] = []
f [x] = x
f (x:xs) = x : f xs
```

Another way to produce an infinite type:

```
let x = head x
```

```
Occurs check: cannot construct the infinite type:
   a0 = [a0]      ("a0 is a list of a0s"--whm)
   In the first argument of `(:)', namely `x'
   In the expression: x : f xs
   In an equation for `f': f (x : xs) = x : f xs
```

The second and third clauses are fine by themselves but together they create a contradiction.

Technique: Comment out clauses (and/or guards) to find the troublemaker, or incompatibilities between them.

Type errors, continued

Recall **ord :: Char -> Int.**

Note these two errors:

```
> ord 5
  No instance for (Num Char) arising from the literal `5'
    Possible fix: add an instance declaration for (Num Char)

> length 3
  No instance for (Num [a0]) arising from the literal `3'
    Possible fix: add an instance declaration for (Num [a0])
```

The error "**No instance for (A B)**" means I want a **B** but got an **A**.

The suggested fix, adding an instance declaration, is always wrong in our simple Haskell world.

Debugging

Debugging in general

My advice in a nutshell for debugging in Haskell:

Don't need to do any debugging!

My usual development process in Haskell:

1. Work out expressions at the **ghci** prompt.
2. Write a function using those expressions and put it in a file.
3. Test that function at the **ghci** prompt.
4. Repeat with the next function.

With conventional languages I might write dozens of lines of code before trying them out.

With Haskell I might write a half-dozen lines of code before trying them out.

Tracing

The **Debug.Trace** module provides a **trace** function that sneakily does output without getting embroiled with the I/O machinery.

Consider a trivial function:

```
f 1 = 10
f n = n * 5 + 7
```

Let's augment it with tracing:

```
import Debug.Trace
f 1 = trace "f: first case" 10
f n = trace "f: default case" n * 5 + 7
```

Execution:

```
> f 1
f: first case
10

> f 3
f: default case
22
```

Tracing, continued

Here's **countEO** with tracing:

```
import Debug.Trace
countEO [] = (0,0)
countEO list@(x:xs)
   | odd x = (evens, odds+1)
   | otherwise = (evens+1, odds)
 where
  result = countEO xs
  (evens,odds) =
    trace ("countEO " ++ show xs ++ " --> " ++ show result) result
```

Execution:

```
> countEO [3,2,4]
(countEO [] --> (0,0)
countEO [4] --> (1,0)
countEO [2,4] --> (2,0)
2,1)
```

Before tracing the **where** was:

```
(evens,odds) = countEO xs
```

ghci's debugger

ghci does have some debugging support but debugging is *expression-based*. Here's some simple interaction with it on **countEO**:

```
> :step countEO [3,2,4]
Stopped at countEO.hs:(1,1)-(6,29)
_result :: (t, t1) = _
> :step
Stopped at countEO.hs:3:7-11
_result :: Bool = _
x :: Integer = 3
> :step
Stopped at countEO.hs:3:15-29
_result :: (t, t1) = _
evens :: t = _
odds :: t1 = _
> :step
(Stopped at countEO.hs:6:20-29
_result :: (t, t1) = _
xs :: [Integer] = [2,4]
```

```
countEO [] = (0,0)
countEO (x:xs)
    | odd x = (evens, odds+1)
    | otherwise = (evens+1, odds)
  where
    (evens,odds) = countEO xs
```

_result shows type of current expression

Arbitrary expressions can be evaluated at the > prompt (as always).

More on debugging

There's lots more to the debugging support in **gchi**.

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci-debugger.html

http://www.youtube.com/watch?v=1OYljb_3Cdg
GHCi's Debugger - Haskell from Scratch #2

In 352, I promote **gdb** heavily but this is the first time in 372 that I've ever mentioned tracing and debugging for Haskell.

Again, my advice in a nutshell for debugging in Haskell:
Don't need to do any debugging!

Excursion:
A little bit with infinite lists
and lazy evaluation

Infinite lists

Here's a way we've seen to make an infinite list:

```
> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,2
2,23,24,25,26,27,28,29,30,31,32,^C
```

What does the following `let` create?

```
> let f = (!!) [1,3..]
f :: Int -> Integer
```

A function that produces the Nth odd number, zero-based.

Yes, we could say `nthOdd n = (n*2)+1` but that wouldn't be nearly as much fun! (This *is* functional programming!)

I want you to be cognizant of performance but don't let concerns about performance stifle creativity!

Lazy evaluation

Consider the following **let**. Why does it complete?

```
> let fives=[5,10..]
fives :: [Integer]
```

A simple answer we'll later refine:

Haskell uses *lazy evaluation*. Values aren't computed until needed.

How will the following expression behave?

```
> take (head fives) fives
[5,10,15,20,25]
```

Haskell computes the first element of **fives**, and then four more elements of **fives**.

Lazy evaluation, continued

Here is an expression that is said to be *non-terminating:*

```
> length fives
...when tired of waiting...^C Interrupted.
```

The value of **length fives** is said to be ⊥, read "bottom".

But, we can bind a name to **length fives**:

```
> let numFives = length fives
numFives :: Int
```

That completes because Haskell hasn't yet needed to compute a value for **length fives**.

We can get another coffee break by asking Haskell to print the value of **numFives:**

```
> numFives
...after a while...^CInterrupted.
```

Lazy evaluation, continued

We can use **:print** to explore lazy evaluation:

```
> let fives = [5,10..]

> :print fives
fives = (_t2::[Integer])

> take 3 fives
[5,10,15]
```

What do you think **:print fives** will now show?

```
> :print fives
fives = 5 : 10 : 15 : (_t3::[Integer])
```

Lazy evaluation, continued

Consider this function:

```
f x y z = if x < y then y else z
```

Will the following expressions terminate?

```
> f 2 3 (length [1..])
3

> f 3 2 (length [1..])
^CInterrupted.

> f 3 (length [1..]) 2
^CInterrupted.
```

Sidebar: Lazy vs. non-strict

In fact, Haskell doesn't fully meet the requirements of lazy evaluation.
The word "lazy" appears only once in the Haskell 2010 Report.

What Haskell does provide is *non-strict evaluation*:
Function arguments are not evaluated until a value is needed.

From the previous slide:

```
f x y z = if x < y then y else z
```

Reconsider the following wrt. non-strict evaluation:

```
> f 2 3 (length [1..]) -- Third argument is not used
3

> f 3 2 (length [1..]) -- Third argument is used
^CInterrupted.
```

See **wiki.haskell.org/Lazy_vs._non-strict** for the fine points of lazy evaluation vs. non-strict evaluation. Google for more, too.

More with infinite lists

Speculate: Can infinite lists be concatenated?

```
> let values = [1..] ++ [5,10..] ++ [1,2,3]
> :t values
values :: [Integer]
```

What will the following do?

```
> let nums = [1..]
> nums > [1,2,3,5]
False
```

False due to lexicographic comparison—4 < 5

How far did evaluation of **nums** progress?

```
> :print nums
nums = 1 : 2 : 3 : 4 : (_t2::[Integer])
```

Infinite expressions

What does the following expression mean?

```
> let threes = 3 : threes
```

threes is a list whose head is 3 and whose tail is threes!

```
> take 5 threes
[3,3,3,3,3]
```

How about the following?

```
> let xys = ['x','y'] ++ xys

> take 5 xys
"xyxyx"

> xys !! 100000000
'x'
```

One more:

```
> let x = 1 + x
> x
^CInterrupted.
```

intsFrom

Problem: write a function **intsFrom** that produces the integers from a starting value.

```
> intsFrom 1
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...

> intsFrom 1000
[1000,1001,1002,1003,1004,1005,1006,1007,1008,...

> take 5 (intsFrom 1000000)
[1000000,1000001,1000002,1000003,1000004]
```

Solution:

```
intsFrom n = n : intsFrom (n + 1)
```

Does **length (intsFrom (minBound::Int))** terminate?

repblock

The **cycle** function returns an infinite number of repetitions of its argument, a list:

```
> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
```

Using **cycle**, write **repblock**:

```
> repblock "abc" 4 3 -- string width height
abca
bcab
cabc

> repblock "+-" 3 2
+-+
-+-
```

repblock, continued

At hand: **repblock s width height**

Approach: Create an infinite repetition of **s** and take **width**-sized lines **height** times.

Solution:

```
repblock :: String -> Int -> Int -> IO ()
repblock s width height =
   putStr (repblock' (cycle s) width height)

repblock' :: String -> Int -> Int -> String
repblock' s width height
    | height == 0 = ""
    | otherwise = take width s ++ "\n" ++
        repblock' (drop width s) width (height - 1)
```

Higher-order functions

Remember: Functions are values

Remember:

A fundamental characteristic of a functional language: functions are values that can be used as flexibly as values of other types.

Here are some more examples of that. What do the following do?

```
> let nums = [1..10]

> (if 3 < 4 then head else last) nums
1

> let funcs = (tail, (:) 100)

> fst funcs nums
[2,3,4,5,6,7,8,9,10]

> snd funcs nums
[100,1,2,3,4,5,6,7,8,9,10]
```

Lists of functions

We can work with lists of functions:

```
> let funcs = [head, last]

> funcs
[<function>,<function>]

> let nums = [1..10]

> head funcs nums
1

> (funcs!!1) nums
10

> last [last]
<function>
```

Lists of functions, continued

Is the following valid?

```
> [take, tail, init]
Couldn't match type `[a2]' with `Int'
   Expected type: Int -> [a0] -> [a0]
        Actual type: [a2] -> [a2]
   In the expression: init
```

What's the problem?

take does not have the same type as **tail** and **init**.

Puzzle: Make **[take, tail, init]** valid by adding two characters.

```
> [take 5, tail, init]
[<function>,<function>,<function>]
```

Comparing functions

Can functions be compared?

```
> add == plus
No instance for (Eq (Integer -> Integer -> Integer))
    arising from a use of `=='
In the expression: add == plus
```

You might see a proof based on this in 473:

If we could determine if two arbitrary functions perform the same computation, we could solve the halting problem, which is considered to be unsolvable.

Because functions can't be compared, this version of **length** won't work for lists of functions: (Its type: **(Num a, Eq t) => [t] -> a**)

```
len list@(_h:t)
    | list == [] = 0
    | otherwise = 1 + len t
```

A simple *higher-order function*

Definition: A *higher-order function* is a function that has one or more arguments that are functions.

twice is a higher-order function with <u>two</u> arguments: **f** and **x**

```
    twice f x = f (f x)
```

What does it do?

```
    > twice tail [1,2,3,4,5]
    [3,4,5]

    > tail (tail [1,2,3,4,5])
    [3,4,5]
```

.

twice, continued

At hand:

```
> let twice f x = f (f x)
> twice tail [1,2,3,4,5]
[3,4,5]
```

Let's make the precedence explicit:

```
> ((twice tail) [1,2,3,4,5])
[3,4,5]
```

Consider a partial application...

```
> let t2 = twice tail    -- like let t2 x = tail (tail x)
> t2
<function>
it :: [a] -> [a]
```

twice, continued

At hand:

```
> let twice f x = f (f x)
> twice tail [1,2,3,4,5]
[3,4,5]
```

Let's **give** twice a partial application!

```
> twice (drop 2) [1..5]
[5]
```

Let's make a partial application with a partial application!

```
> twice (drop 5)
<function>
> it ['a'..'z']
"klmnopqrstuvwxyz"
```

Try these!

```
twice (twice (drop 3)) [1..20]
twice (twice (take 3)) [1..20]
```

twice, continued

A *higher-order function* is a function that has one or more arguments that are functions.

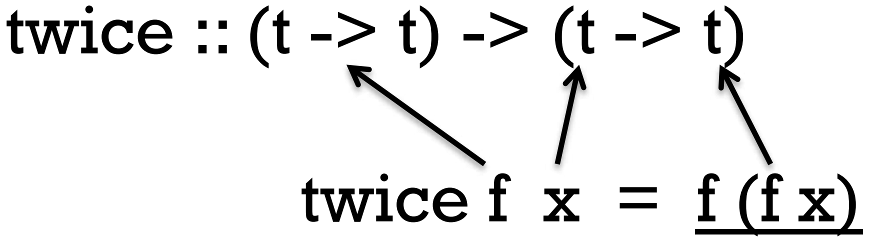
At hand:

```
    twice f x = f (f x)
```

What's the the type of **twice**?

```
    > :t twice
    twice :: (t -> t) -> t -> t
```

Parentheses added to show precedence:



```
    twice :: (t -> t) -> (t -> t)

              twice f  x  =  f (f x)
```

What's the correspondence between the elements of the clause and the elements of the type?

The **map** function

The Prelude's **map** function

Recall **double x = x * 2**

map is a Prelude function that applies a function to each element of a list, producing a new list:

```
> map double [1..5]
[2,4,6,8,10]

> map length (words "a few words")
[1,3,5]

> map head (words "a few words")
"afw"
```

Is **map** a higher order function?

Yes! It's first argument is a function.

map, continued

At hand:

```
> map double [1..5]
[2,4,6,8,10]
```

Write it!

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

What is its type?

```
map :: (a -> b) -> [a] -> [b]
```

What's the relationship between the length of the input and output lists?

The lengths are always the same.

map, continued

Mapping (via map) is applying a transformation (a function) to each of the values in a list, always producing a new list of the same length.

```
> map chr [97,32,98,105,103,32,99,97,116]
"a big cat"

> map isLetter it
[True,False,True,True,True,False,True,True,True]

> map not it
[False,True,False,False,False,True,False,False,False]

> map head (map show it) -- Note: show True is "True"
"FTFFFTFFF"
```

Sidebar: map can go parallel

Here's another map:

```
> map weather [85,55,75]
["Hot!","Cold!","Nice"]
```

This is equivalent:

```
> [weather 85, weather 55, weather 75]
["Hot!","Cold!","Nice"]
```

Because functions have no side effects, we can immediately turn a mapping into a parallel computation. We might start each function call on a separate processor and combine the values when all are done.

map and partial applications

What's the result of these?

```
> map (add 5) [1..10]
[6,7,8,9,10,11,12,13,14,15]

> map (drop 1) (words "the knot was cold")
["he","not","as","old"]

> map (replicate 5) "abc"
["aaaaa","bbbbb","ccccc"]
```

map and partial applications, cont.

What's going on here?

```
> let f = map double
> f [1..5]
[2,4,6,8,10]

> map f [[1..3],[10..15]]
[[2,4,6],[20,22,24,26,28,30]]
```

Here's the above in one step:

```
> map (map double) [[1..3],[10..15]]
[[2,4,6],[20,22,24,26,28,30]]
```

Here's one way to think about it:

```
[(map double) [1..3], (map double) [10..15]]
```

Sections

Instead of using **map (add 5)** to add 5 to the values in a list, we should use a *section* instead: (it's the idiomatic way!)

```
> map (5+) [1,2,3]
[6,7,8]   -- [5+ 1, 5+ 2, 5+ 3]
```

More sections:

```
> map (10*) [1,2,3]
[10,20,30]

> map (++"*") (words "a few words")
["a*","few*","words*"]

> map ("*"++) (words "a few words")
["*a","*few","*words"]
```

Sections, continued

Sections have one of two forms:

(*infix-operator value*) Examples: (+5), (/10)

(*value infix-operator*) Examples: (5*), ("x"++)

<u>Iff</u> the operator is commutative, the two forms are equivalent.

```
> map (3<=) [1..4]
[False,False,True,True]
```

[3 <= 1, 3 <= 2, 3 <= 3, 3 <= 4]

```
> map (<=3) [1..4]
[True,True,True,False]
```

[1 <= 3, 2 <= 3, 3 <= 3, 4 <= 4]

Sections aren't just for **map**; they're a general mechanism.

```
> twice (+5) 3
13
```

travel, revisited

Now that we're good at recursion...

Some of the problems on the next assignment will encourage working with higher-order functions by prohibiting you from writing any recursive functions!

Think of it as isolating muscle groups when weight training.

Here's a simple way to avoid what's prohibited:

Pretend that you no longer understand recursion!

What's a base case? Is it related to baseball?
Why would a function call itself? How's it stop?
Is a recursive plunge refreshing?

If you were UNIX machines, I'd do `chmod 0` on an appropriate section of your brains.

travel revisited

Recall our traveling robot: (slide 195)

```
> travel "nnee"
"Got lost"

> travel "nnss"
"Got home"
```

Recall our approach:

Argument value: "nnee"

Mapped to tuples: (0,1) (0,1) (1,0) (1,0)

Sum of tuples: (2,2)

How can we solve it non-recursively?

travel, continued

Recall:

```
> :t mapMove
mapMove :: Char -> (Int, Int)

> mapMove 'n'
(0,1)
```

Now what?

```
> map mapMove "nneen"
[(0,1),(0,1),(1,0),(1,0),(0,1)]
```

Can we sum them with **map**?

travel, continued

We have:

```
> let disps= map mapMove "nneen"
[(0,1),(0,1),(1,0),(1,0),(0,1)]
```

We want: (2,3)

Any ideas?

```
> :t fst
fst :: (a, b) -> a

> map fst disps
[0,0,1,1,0]

> map snd disps
[1,1,0,0,1]
```

travel, revisited

We have:

```
> let disps= map mapMove "nneen"
[(0,1),(0,1),(1,0),(1,0),(0,1)]
> map fst disps
[0,0,1,1,0]
> map snd disps
[1,1,0,0,1]
```

We want: (2,3)

Ideas?

```
> :t sum
sum :: Num a => [a] -> a

> (sum (map fst disps), sum (map snd disps))
(2,3)
```

travel—Final answer

```
travel :: [Char] -> [Char]
travel s
  | totalDisp == (0,0) = "Got home"
  | otherwise = "Got lost"
  where
    disps = map mapMove s
    totalDisp = (sum (map fst disps),
                 sum (map snd disps))
```

Did we have to understand recursion to write this?

No.

Did we <u>write</u> any recursive functions?

No.

Did we <u>use</u> any recursive functions?

Maybe so, but using recursive functions doesn't violate the prohibition at hand.

Filtering

Filtering

Another higher order function in the Prelude is **filter**:

```
> filter odd [1..10]
[1,3,5,7,9]

> filter isDigit "(800) 555-1212"
"8005551212"
```

What's **filter** doing?

What is the type of **filter**?

```
filter :: (a -> Bool) -> [a] -> [a]
```

Think of **filter** as filtering *in*, not filtering *out*.

filter, continued

More...

```
>  filter (<= 5) (filter odd [1..10])
[1,3,5]

> map (filter isDigit) ["br549", "24/7"]
["549","247"]

> filter (`elem` "aeiou") "some words here"
"oeoee"
```

Note that (`elem` ...) is a section!

```
elem :: Eq a => a -> [a] -> Bool
```

filter, continued

At hand:

```
> filter odd [1..10]
[1,3,5,7,9]

> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

Let's write **filter**!

```
filter _ [] = []
filter f (x:xs)
    | f x = x : filteredTail
    | otherwise = filteredTail
  where
    filteredTail = filter f xs
```

filter uses a *predicate*

filter's first argument (a function) is called a *predicate* because inclusion of each value is predicated on the result of calling that function with that value.

Several Prelude functions use predicates. Here are two:

```
all :: (a -> Bool) -> [a] -> Bool
> all even [2,4,6,8]
True
> all even [2,4,6,7]
False

dropWhile :: (a -> Bool) -> [a] -> [a]
> dropWhile isSpace "  testing  "
"testing  "
> dropWhile isLetter it
"  "
```

map vs. **filter**

For reference:

```
> map double [1..10]
[2,4,6,8,10,12,14,16,18,20]

> filter odd [1..10]
[1,3,5,7,9]
```

map:

transforms a list of values

length *input* == length *output*

filter:

selects values from a list

0 <= length *output* <= length *input*

map and **filter** are in Python and JavaScript, to name two of many languages having them. (And, they're trivial to write!)

Put a big "X" on 281-282 and go to slide 305!

~~Anonymous functions~~

Anonymous functions

We can map a section to double the numbers in a list:

```
> map (*2) [1..5]
[2,4,6,8,10]
```

Alternatively we could use an *anonymous function*:

```
> map (\x -> x * 2) [1..5]
[2,4,6,8,10]
```

What are things we can do with an anonymous function that we can't do with a section?

```
> map (\n -> n * 3 + 7) [1..5]
[10,13,16,19,22]

> filter (\x -> head x == last x) (words "pop top suds")
["pop","suds"]
```

Anonymous functions, continued

The general form:

\ *pattern1 ... patternN* -> *expression*

Simple syntax suggestion: enclose the whole works in parentheses.

```
map (\x -> x * 2) [1..5]
```

The typical use case for an anonymous function is a single instance of supplying a higher order function with a computation that can't be expressed with a section or partial application.

Anonymous functions are also called *lambdas*, *lambda expressions*, and *lambda abstractions*.

The \ character was chosen due to its similarity to λ, used in Lambda calculus, another system for expressing computation.

Larger example: longest

Example: longest line(s) in a file

Imagine a program to print the longest line(s) in a file, along with their line numbers:

```
% runghc longest.hs /usr/share/dict/web2
72632:formaldehydesulphoxylate
140339:pathologicopsychological
175108:scientificophilosophical
200796:tetraiodophenolphthalein
203042:thyroparathyroidectomize
```

What are some ways in which we could approach it?

longest, continued

Let's work with a shorter file for development testing:

```
% cat longest.1
data
to
test
```

readFile in the Prelude lazily returns the full contents of a file as a string:

```
> readFile "longest.1"
"data\nto\ntest\n"
```

To avoid wading into I/O yet, let's focus on a function that operates on a string of characters (the full contents of a file):

```
> longest "data\nto\ntest\n"
"1:data\n3:test\n"
```

longest, continued

Let's work through a series of transformations of the data:

```
> let bytes = "data\nto\ntest\n"

> let lns = lines bytes
["data","to","test"]
```

Note: To save space, values of **let** bindings are being shown immediately after each **let**. E.g., **> lns** is not shown above.

Let's use **zip3** and **map length** to create (length, line-number, line) triples:

```
> let triples = zip3 (map length lns) [1..] lns
[(4,1,"data"),(2,2,"to"),(4,3,"test")]
```

longest, continued

We have (length, line-number, line) triples at hand:

```
> triples
[(4,1,"data"),(2,2,"to"),(4,3,"test")]
```

Let's use sort :: Ord a => [a] -> [a] on them:

```
> let sortedTriples = reverse (sort triples)
[(4,3,"test"),(4,1,"data"),(2,2,"to")]
```

Note that by having the line length first, triples are sorted first by line length, with ties resolved by line number.

We use reverse to get a descending order.

If line length weren't first, we'd instead use

```
Data.List.sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

and supply a function that returns an Ordering.

longest, continued

At hand:

```
> sortedTriples
[(4,3,"test"),(4,1,"data"),(2,2,"to")]
```

We'll handle ties by using **takeWhile** to get all the triples with lines of the maximum length.

Let's use a helper function to get the first element of a 3-tuple:

```
> let first (len, _, _) = len
> let maxLength = first (head sortedTriples)
4
```

first will be used in another place but were it not for that we might have used a pattern:

```
let (maxLength,_,_) = head sortedTriples
```

longest, continued

At hand:

```
> sortedTriples
[(4,3,"test"),(4,1,"data"),(2,2,"to")]

> maxLength
4
```

Let's use **takeWhile :: (a -> Bool) -> [a] -> [a]** to get the triples having the maximum length:

```
> let maxTriples = takeWhile
    (\triple -> first triple  == maxLength) sortedTriples
[(4,3,"test"),(4,1,"data")]
```

anonymous function for takeWhile

longest, continued

At hand:

```
> maxTriples
[(4,3,"test"),(4,1,"data")]
```

Let's map an anonymous function to turn the triples into lines prefixed with their line number:

```
> let linesWithNums =
      map (\(_,num,line) -> show num ++ ":" ++ line)
             maxTriples
["3:test","1:data"]
```

We can now produce a ready-to-print result:

```
> let result = unlines (reverse linesWithNums)
> result
"1:data\n3:test\n"
```

longest, continued

Let's package up our work into a function:

```
longest bytes = result
    where
        lns = lines bytes
        triples = zip3 (map length lns) [1..] lns
        sortedTriples = reverse (sort triples)
        maxLength = first (head sortedTriples)
        maxTriples = takeWhile
            (\triple -> first triple  == maxLength) sortedTriples
        linesWithNums =
            map (\(_,num,line) -> show num ++ ":" ++ line)
            maxTriples
        result = unlines (reverse linesWithNums)

        first (x,_,_) = x
```

longest, continued

At hand:

```
> longest "data\nto\ntest\n"
"1:data\n3:test\n"
```

Let's add a **main** that handles command-line args and does I/O:

```
% cat longest.hs
import System.Environment (getArgs)
import Data.List (sort)

longest bytes = ...from previous slide...

main = do
   args <- getArgs  -- Get command line args as list
   bytes <- readFile (head args)
   putStr (longest bytes)
```

Execution:

```
$ runghc longest.hs /usr/share/dict/words
39886:electroencephalograph's
```

Composition

Function composition

Given two functions f and g, the *composition* of f and g is a function c that for all values of x, (c x) equals (f (g x))

Here is a primitive compose function that applies two functions in turn:

```
> let compose f g x = f (g x)
```

How many arguments does compose have?

Its type:

```
(b -> c) -> (a -> b) -> a -> c

> compose init tail [1..5]
[2,3,4]

> compose signum negate 3
-1
```

Composition, continued

Haskell has a function composition operator. It is a dot (.)

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Its two operands are functions, and its result is a function.

```
> let numwords = length  .  words

> numwords "just testing this"
3

> map numwords ["a test", "up & down", "done"]
[2,3,1]
```

Composition, continued

Problem: Using composition create a function that returns the next-to-last element in a list:

```
> ntl [1..5]
4

> ntl "abc"
'b'
```

Two solutions:

```
ntl = head  .  tail  .  reverse
ntl = last  .  init
```

Problem: Recall **twice f x = f (f x)**. Define **twice** as a composition.

```
twice f = f  .  f
```

Composition, continued

Problem: Create a function to remove the digits from a string:

```
> rmdigits "Thu Feb  6 19:13:34 MST 2014"
"Thu Feb   :: MST "
```

Solution:

```
> let rmdigits = filter (not  .  isDigit)
```

Given the following, describe **f**:

```
> let f = (*2) . (+3)

> map f [1..5]
[8,10,12,14,16]
```

Would an anonymous function be a better choice?

Composition, continued

Given the following, what's the type of **numwords**?

```
> :type words
words :: String -> [String]

> :type length
length :: [a] -> Int

> let numwords = length . words
```

Type:

```
numwords :: String -> Int
```

Assuming a composition is valid, the type is based only on the input of the rightmost function and the output of the leftmost function.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Note: Next set of slides!

REPLACEMENTS

Put a big "X" on slides 299-300 in the 223-300 set and continue with this set.

Composition, continued

Consider the following:

```
> let s = "It's on!"
> map head (map show (map not (map isLetter s)))
"FFTFFTFFT"
```

Can we use composition to simplify it?

```
> map (head . show . not . isLetter) s
"FFTFFTFFT"
```

In general, because there are no side-effects,

```
map f (map g x)
```

is equivalent to

```
map (f . g) x
```

If f and g did output, how would the output of the two cases differ?

Point-free style

Recall **rmdigits**:

```
> rmdigits "Thu Feb  6 19:13:34 MST 2014"
"Thu Feb   :: MST "
```

What the difference between these two bindings for **rmdigits**?

```
rmdigits s = filter (not  .  isDigit) s

rmdigits = filter (not  .  isDigit)
```

The latter version is said to be written in *point-free style*.

A point-free binding of a function **f** has NO parameters!

Point-free style, continued

I think of point-free style as a natural result of fully grasping partial application and operations like composition.

Although it was nameless, we've already seen examples of point-free style, such as these:

```
nthOdd = (!!) [1,3..]
t2 = twice tail
numwords = length . words
ntl = head . tail . reverse
```

There's nothing too special about point-free style but it does save some visual clutter. It is commonly used.

The term "point-free" comes from topology, where a point-free function operates on points that are not specifically cited.

Point-free style, continued

Problem: Using point-free style, bind **len** to a function that works like the Prelude's **length**.

Handy:

```
> :t const
const :: a -> b -> a

> const 10 20
10

> const [1] "foo"
[1]
```

Solution:

```
len = sum . map (const 1)
```

See also: *Tacit programming* on Wikipedia

Go to slide 312

Anonymous functions (second attempt at 281-283)

Anonymous functions

Imagine that for every number in a list we'd like to double it and then subtract 5.

Here's one way to do it:

```
> let f n = n * 2 - 5
> map f [1..5]
[-3,-1,1,3,5]
```

We could instead use an *anonymous function* to do the same thing:

```
> map (\n -> n * 2 - 5) [1..5]
[-3,-1,1,3,5]
```

Which do you like better, and why?

Anonymous functions, continued

At hand:

```
    let f n = n * 2 - 5
    map f [1..5]
```

vs.

```
    map (\n -> n * 2 - 5) [1..5]
```

If a computation is only used in one place, using an anonymous function lets us specify it on the spot, directly associating its definition with its only use.

We also don't need to think up a name for the function! ☺

Anonymous functions, continued

The general form of an anonymous function:

```
    \ pattern1 ... patternN -> expression
```

Simple syntax suggestion: enclose the whole works in parentheses.

```
    map (\n -> n * 2 - 5) [1..5]
```

Anonymous functions are also called *lambda abstractions* (H10*),* *lambda expressions*, and just *lambdas* (LYAH).

The \ character was chosen due to its similarity to λ, used in the *lambda calculus*, another system for expressing computation.

The typical use case for an anonymous function is a single instance of supplying a higher order function with a computation that can't be expressed with a section or partial application.

Anonymous functions, continued

Speculate: What will **ghci** respond with?

```
> \x y -> x + y * 2
<function>
> it 3 4
11
```

The expression **\x y -> x + y * 2** produces a function value.

Here are three ways to bind the name **double** to a function that doubles a number:

```
double x = x * 2

double = \x -> x * 2

double = (*2)
```

Anonymous functions, continued

Anonymous functions are commonly used with higher order functions such as **map** and **filter**.

```
> map (\w -> (length w, w)) (words "a test now")
[(1,"a"),(4,"test"),(3,"now")]

> map (\c -> "{" ++ [c] ++ "}") "anon."
["{a}","{n}","{o}","{n}","{.}"]

> filter (\x -> head x == last x) (words "pop top suds")
["pop","suds"]
```

In the above examples, the anonymous functions are somewhat like the bodies of loops in imperative languages.

Go to slide 283

Hocus pocus with higher-order functions

Mystery function

What's this function doing?

```
f  a = g
   where
      g  b = a + b
```

Type?

```
f :: Num a => a -> a -> a
```

Interaction:

```
> let f ' = f 10
> f ' 20
30

> f 3 4
7
```

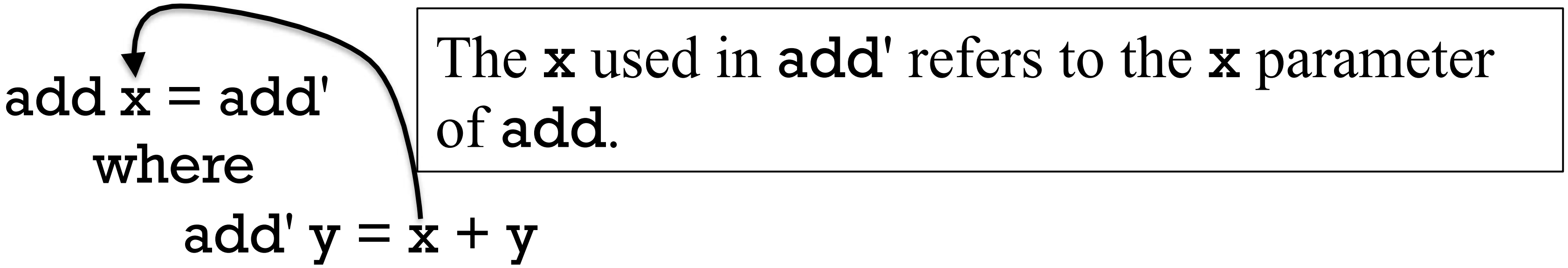
DIY Currying

Fact:

Curried function definitions are really just *syntactic sugar*—they just save some clutter. The don't provide something we can't do without.

Compare these two <u>completely equivalent</u> declarations for **add**:

```
add x y = x + y
```



```
add x = add'
    where
        add' y = x + y
```

The **x** used in **add**' refers to the **x** parameter of **add**.

The result of the call **add 5** is essentially this function:

```
add' y = 5 + y
```

The combination of the code for **add**' and the binding for **x** is known as a *closure*. It contains what's needed for execution.

Sidebar: Syntactic sugar

In 1964 Peter Landin coined the term "syntactic sugar".

A language construct that makes something easier to express but doesn't add a new capability is called *syntactic sugar*. It simply makes the language "sweeter" for human use.

Two examples from C:

- **"abc"** is equivalent to a **char** array initialized with **{'a', 'b', 'c', '\0'}**

- **a[i]** is equivalent to ***(a + i)**

What's an example of syntactic sugar in Java?

The "enhanced for": **for (int i: a) { ... }**

Syntactic sugar, continued

In Haskell a list like `[5, 2, 7]` can be expressed as `5:2:7:[]`.

Is that square-bracket list literal notation syntactic sugar?

What about `[1..]`, `[1,3..]`, `['a'..'z']`?

The `Enum` type class has `enumFrom`, `enumFromTo`, etc.

Recall these equivalent bindings for `double`:

```
double x = x * 2
double = \x -> x * 2
```

Is the first form just syntactic sugar?

What if a function has multiple clauses?

Are anonymous functions syntactic sugar?

Syntactic sugar, continued

"Syntactic sugar causes cancer of the semicolon."
—Alan J. Perlis.

Another Perlis quote:

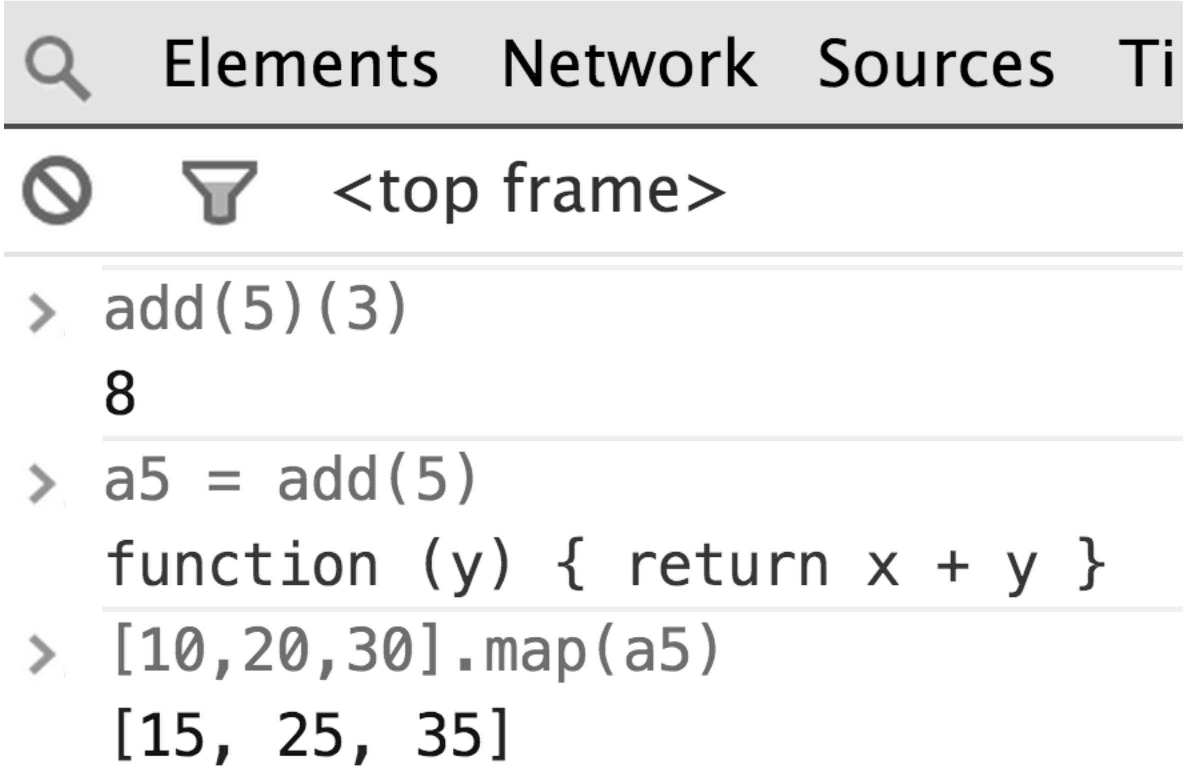
"A language that doesn't affect the way you think about programming is not worth knowing."

Perlis was the first recipient of the ACM's Turing Award.

DIY currying in JavaScript

JavaScript doesn't provide the syntactic sugar of curried function definitions but we can do this:

```
function add(x) {
    return function (y) { return x + y }
   }
```



```
> add(5)(3)
  8
> a5 = add(5)
  function (y) { return x + y }
> [10,20,30].map(a5)
  [15, 25, 35]
```

Try it in Chrome!

View>Developer> JavaScript Console brings up a console.

Type in the code for **add** on one line.

DIY currying in Python

```
>>> def add(x):
...         return lambda y: x + y
...

>>> f = add(5)

>>> type(f)
<type 'function'>

>>> map(f, [10,20,30])
[15, 25, 35]
```

Another mystery function

Here's another mystery function:

```
> let m f x y = f y x

> :type m
m :: (t1 -> t2 -> t) -> t2 -> t1 -> t
```

Can you devise a call to `m`?

```
> m add 3 4
7

> m (++) "a" "b"
"ba"
```

What is `m` doing? What could `m` be useful for?

flip

At hand:

```
m f x y = f y x
```

m is actually a Prelude function named **flip**:

```
> :t flip
flip :: (a -> b -> c) -> b -> a -> c

> flip take [1..10] 3
[1,2,3]

> let ftake = flip take
> ftake [1..10] 3
[1,2,3]
```

Any ideas on how to use it?

flip, continued

At hand:

```
flip f x y = f y x

> map (flip take "Haskell") [1..7]
["H","Ha","Has","Hask","Haske","Haskel","Haskell"]
```

Problem: write a function that behaves like this:

```
> f 'a'
["a","aa","aaa","aaaa","aaaaa",...
```

Solution:

```
f x = map (flip replicate x) [1..]
```

flip, continued

From assignment 3:

```
> splits "abcd"
[("a","bcd"),("ab","cd"),("abc","d")]
```

Some students have noticed the Prelude's **splitAt**:

```
> splitAt 2 [10,20,30,40]
([10,20],[30,40])
```

Problem: Write **splits** using higher order functions but no explicit recursion.

Solution:

```
splits list = map (flip splitAt list) [1..(length list - 1)]
```

The **$** operator

$ is the "application operator". Note what **:info** shows:

```
> :info ($)
($) :: (a -> b) -> a -> b
infixr 0 $      -- right associative infix operator with very
                -- low precedence
```

The following binding of **$** uses an infix syntax:

```
f $ x  =  f x        -- Equivalent: ($) f x = f x
```

Usage:

```
> negate $ 3 + 4
-7
```

What's the point of it?

The $ operator, continued

$ is a low precedence, right associative operator that applies a function to a value:

```
f $ x  =  f x
```

Because + has higher precedence than $, the expression

```
negate $ 3 + 4
```

groups like this:

```
negate $ (3 + 4)
```

How does the following expression group?

```
filter (>3) $ map length $ words "up and down"

filter (>3) (map length (words "up and down"))
```

Don't confuse $ with . (composition)!

Currying the uncurried

Problem: We're given a function whose argument is a 2-tuple but we wish it were curried so we could use a partial application of it.

```
g :: (Int, Int) -> Int
g (x,y) = x^2 + 3*x*y + 2*y^2

> g (3,4)
77
```

Solution: Curry it with **curry** from the Prelude!

```
> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

Your problem: Write **curry**!

Currying the uncurried, continued

At hand:

```
> g (3,4)
77
> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

Here's **curry**, and use of it:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x,y)

> let cg = curry g
> :type cg
cg :: Int -> Int -> Int

> cg 3 4
77
```

Currying the uncurried, continued

At hand:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

The key: **(curry g 3)** is a partial application of **curry**!

```
Call: curry g 3
            ↓ ↓
Dcl:  curry f x y = f  (x,  y)
                  = g (3, y)
```

Currying the uncurried, continued

At hand:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
```
(parentheses added)

```
curry f x y = f (x, y)

> map (curry g 3) [1..10]
[20,35,54,77,104,135,170,209,252,299]
```

Let's get **flip** into the game!

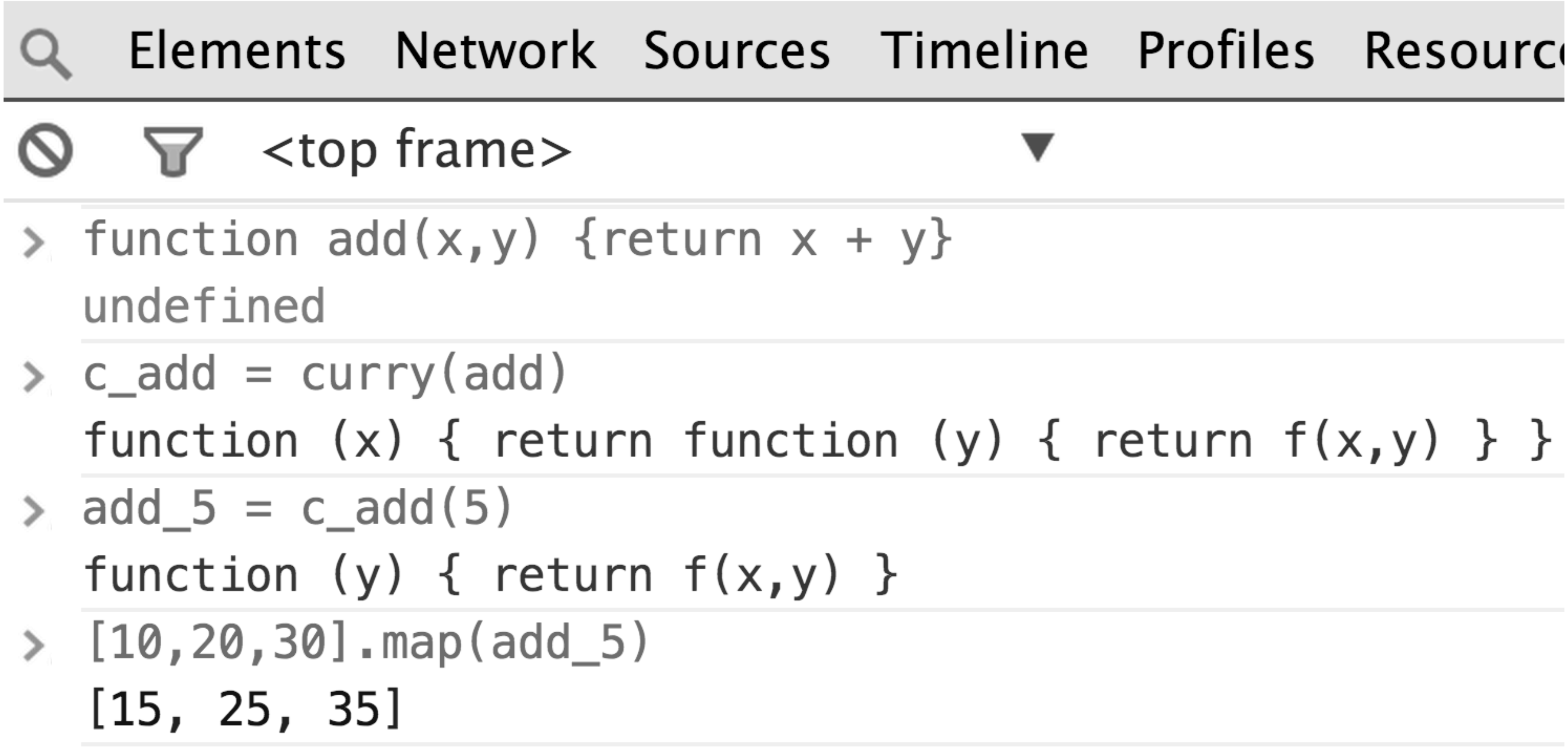
```
> map (flip (curry g) 4) [1..10]
[45,60,77,96,117,140,165,192,221,252]
```

The counterpart of **curry** is **uncurry**:

```
> uncurry (+) (3,4)
7
```

A **curry** function for JavaScript

```
function curry(f) {
    return function(x) {
        return function (y) { return f(x,y) }
        }
    }
```



```
> function add(x,y) {return x + y}
  undefined
> c_add = curry(add)
  function (x) { return function (y) { return f(x,y) } }
> add_5 = c_add(5)
  function (y) { return f(x,y) }
> [10,20,30].map(add_5)
  [15, 25, 35]
```

Folding

Reduction

We can *reduce* a list by a binary operator by inserting that operator between the elements in the list:

[1,2,3,4] reduced by + is 1 + 2 + 3 + 4

["a","bc", "def"] reduced by ++ is "a" ++ "bc" ++ "def"

Imagine a function **reduce** that does reduction by an operator.

```
> reduce (+) [1,2,3,4]
10

> reduce (++) ["a","bc","def"]
"abcdef"

> reduce max [10,2,4]
10                                  -- think of 10 `max` 2 `max` 4
```

Reduction, continued

At hand:

```
> reduce (+) [1,2,3,4]
10
```

An implementation of `reduce`:

```
reduce _ [] = undefined
reduce _ [x] = x
reduce op (x:xs) = x `op` reduce op xs
```

Does `reduce + [1,2,3,4]` do

```
((1 + 2) + 3) + 4
```

or

```
1 + (2 + (3 + 4))
```

?

In general, when would the grouping matter?

If the operation is non-associative, like division.

foldl1 and foldr1

In the Prelude there's no reduce but there is foldl1 and foldr1.

```
> foldl1 (+) [1..4]
10

> foldl1 max "maximum"
'x'

> foldl1 (/) [1,2,3]
0.16666666666666666     -- left associative: (1 / 2) / 3

> foldr1 (/) [1,2,3]             -- right associative: 1 / (2 / 3)
1.5
```

The types of both foldl1 and foldr1 are (a -> a -> a) -> [a] -> a.

foldl1 vs. foldl

Another folding function is foldl (no 1). Let's compare the types of the two:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl :: (a -> b -> a) -> a -> [b] -> a
```

What's different between them?

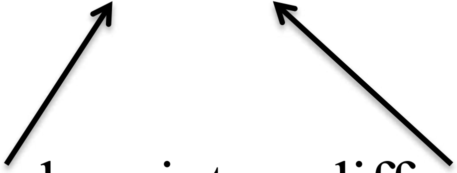
First difference: foldl requires one more argument:

```
> foldl (+) 0 [1..10]
55

> foldl (+) 100 []
100

> foldl1 (+) []
*** Exception: Prelude.foldl1: empty list
```

foldl1 vs. foldl, continued

Again, the types:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl :: (a -> b -> a) -> a -> [b] -> a
```



Second difference:

foldl can fold a list of values into a different type! (This is BIG!)

Examples:

```
> foldl f1 0 ["just","a","test"]
3                              -- folded strings into a number

> foldl f2 "stars: " [3,1,2]
"stars: ******"                -- folded numbers into a string

> foldl f3 0 [(1,1),(2,3),(5,10)]
57                             -- folded two-tuples into a sum of products
```

foldl

For reference:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Here's another view of the type: (acm_t stands for accumulator type)

```
foldl :: (acm_t -> elem_t -> acm_t) -> acm_t -> [elem_t] -> acm_t
```

foldl takes three arguments:

1. A function that takes an accumulated value and an element value and produces a new accumulated value
2. An initial accumulated value
3. A list of elements

Recall:

```
> foldl f1 0 ["just","a","test"]
3

> foldl f2 "stars: " [3,1,2]
"stars: ******"
```

Recall:

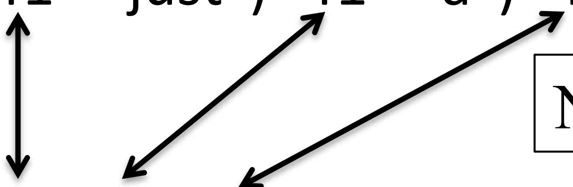
```
> foldl f1 0 ["just","a","test"]
3
```

Here are the computations that foldl did to produce that result

```
> f1 0 "just"
1
> f1 it "a"
2
> f1 it "test"
3
```

Let's do it in one shot, and use backquotes to infix f1:

```
> ((0  `f1`  "just")  `f1`  "a")  `f1`  "test"
3
```



Note the parallels between these two.

1 + 2 + 3 + 4 is the reduction we started this section with.

foldl, continued

At hand:

```
> f1 0 "just"
1
> f1 it "a"
2
> f1 it "test"
3
```

For reference:

```
> foldl f1 0 ["just","a","test"]
3
```

Problem: Write a function f1 that behaves like above.

Starter:

```
f1 :: acm_t -> elem_t -> acm_t
f1 acm elem =  acm + 1
```

Congratulations! You just wrote a *folding function*!

Recall:

```
> foldl f2 "stars: " [3,1,2]
"stars: ******"
```

Here's what foldl does with f2 and the initial value, "stars: ":

```
> f2 "stars: " 3
"stars: ***"
> f2 it 1
"stars: ****"
> f2 it 2
"stars: ******"
```

Write f2, with this starter:

```
f2 :: acm_t -> elem_t -> acm_t
f2 acm elem = acm ++ replicate elem '*'
```

Look! Another folding function!

Folding abstracts a common pattern of computation: a series of values contribute one-by-one to a result that accumulates.

The challenge of folding is to envision a function that takes nothing but an accumulated value (acm) and a single list element (elem) and produces a result that reflects the contribution of elem to acm.

```
f2 acm elem = acm ++ replicate elem '*'
```

It's important to recognize that the folding function never sees the full list!

We then call foldl with that folding function, an appropriate initial value and a list of values.

```
foldl f2 "stars: " [3,1,2]
```

foldl orchestrates the computation by making the appropriate series of calls to the folding function.

```
> (("stars: " `f2` 3) `f2` 1) `f2` 2
"stars: ******"
```

foldl, continued

Recall:

```
> foldl f3 0 [(1,1),(2,3),(5,10)]
57
```

Here are the calls that foldl will make:

```
> f3 0 (1,1)
1
> f3 it (2,3)
7
> f3 it (5,10)
57
```

Problem: write f3!

```
f3 acm (a,b) = acm + a * b
```

foldl, continued

Remember that

```
    foldl f 0 [10,20,30]
```

is like

```
    ((0 `f` 10) `f` 20) `f` 30
```

Here's an implementation of foldl:

```
 foldl f acm [] = acm
 foldl f acm (elem:elems) =  foldl f (acm `f` elem) elems
```

We can implement foldl1 in terms of foldl:

```
 foldl1 f (x1:xs) = foldl f x1 xs
 foldl1 _ [] = error "empty list"
```

A non-recursive countEO

Let's use folding to implement our even/odd counter non-recursively.

```
> countEO [3,4,7,9]
(1,3)
```

Often, a good place to start on a folding is to figure out what the initial accumulator value should be. What should it be for countEO?

```
(0,0)
```

What will be the calls to the folding function?

```
> f (0,0) 3
(0,1)
> f it 4
(1,1)
> f it 7
(1,2)
> f it 9
(1,3)
```

Now we're ready to write countEO:

```
countEO nums = foldl f (0,0) nums
    where
        f (evens, odds) elem
            | even elem = (evens + 1, odds)
            | otherwise = (evens, odds + 1)
```

Folds with anonymous functions

If a folding function is simple, an anonymous function is typically used.

Let's redo our three earlier folds with anonymous functions:

```
> foldl (\acm _ -> acm + 1) 0 ["just","a","test"]
3

> foldl (\acm elem -> acm ++ replicate elem '*') "stars: " [3,1,2]
"stars: ******"

> foldl (\acm (a,b) -> acm + a * b) 0 [(1,1),(2,3),(5,10)]
57
```

foldr

The counterpart of foldl is foldr. Compare their meanings:

```
foldl f zero [e1, e2, ..., eN] == (...((zero `f` e1) `f` e2) `f`...)`f` eN

foldr f zero [e1, e2, ..., eN] == e1 `f` (e2 `f` ... (eN `f` zero)...)
```

"zero" represents a computation-specific initial value. Note that with foldl, zero is leftmost; but with foldr, zero is rightmost.

Their types, with long type variables:

```
foldl :: (acm -> val -> acm) -> acm -> [val] -> acm
foldr :: (val -> acm -> acm) -> acm -> [val] -> acm
```

Mnemonic aid:

foldl's folding function has the accumulator on the left
foldr's folding function has the accumulator on the right

foldr, continued

Because cons (:) is right-associative, folds that produce lists are often done with foldr.

Imagine a function that keeps the odd numbers in a list:

```
> keepOdds [5,4,2,3]
[5,3]
```

Implementation, with foldr:

```
keepOdds list = foldr f [] list
    where
        f elem acm
            | odd elem = elem : acm
            | otherwise = acm
```

Here are calls to the folding function:

```
> f 3 [] -- rightmost first!
[3]
> f 2 it
[3]
> f 4 it
[3]
> f 5 it
[5,3]
```

filter and map with folds?

keepOdds could have been written using filter:

```
keepOdds = filter odd
```

Can we implement filter as a fold?

```
filter predicate list = foldr f [] list
    where
        f elem acm
            | predicate elem = elem : acm
            | otherwise = acm
```

How about implmenting map as a fold?

```
map f = foldr (\elem acm -> f elem : acm) []
```

Is folding One Operation to Rule Them All?

paired with a fold

Recall **paired** from assignment 3:

```
> paired "((())())"
True
```

Can we implement **paired** with a fold?

```
counter (-1) _ = -1
counter total '(' = total + 1
counter total ')' = total - 1
counter total _ = total

paired s = foldl counter 0 s == 0
```

paired is a fold with a simple wrapper, to test the result of the fold.

Point-free:

```
paired = (0==) . foldl counter 0
```

Folding, continued

Data.List.partition partitions a list based on a predicate:

```
> partition isLetter "Thu Feb 13 16:59:03 MST 2014"
("ThuFebMST","  13 16:59:03  2014")

> partition odd [1..10]
([1,3,5,7,9],[2,4,6,8,10])
```

Problem: Write **partition** using a fold.

```
sorter f val (pass, fail)
    | f val = (val:pass, fail)
    | otherwise = (pass, val:fail)

partition f = foldr (sorter f) ([],[])
```

A progression of folds

Let's do a progression of folds related to finding vowels in a string.

First, let's write a fold that counts vowels in a string:

```
> foldr (\val acm ->
        acm + if  val `elem` "aeiou" then 1 else 0)  0  "ate"
2
```

Now let's produce both a count and the vowels themselves:

```
> foldr (\val acm@(n, vows) ->
          if  val `elem` "aeiou" then (n+1, val:vows)
                                   else acm) (0,[]) "ate"
(2,"ae")
```

A progression of folds, continued

Finally, let's write a function that produces a list of vowels and their positions:

```
> vowelPositions "Down to Rubyville!"
[('o',1),('o',6),('u',9),('i',13),('e',16)]
```

Solution:

```
vowelPositions s = reverse result
  where (_, result, _) =
    foldl (\acm@(n, vows,pos) val ->
      if  val `elem` "aeiou" then (n, (val,pos):vows,pos+1)
                                 else (n,vows,pos+1)) (0,[],0) s
```

Note that **vowelPositions** uses **fold<u>l</u>** to produce a 3-tuple whose middle element is the result, in reverse order. (This is another function that's a fold with a wrapper, like **paired** on 348).

map vs. **filter** vs. folding

map:

transforms a list of values

length *input* == length *output*

filter:

selects values from a list

0 <= length *output* <= length *input*

folding

Input: A list of values and an initial value for accumulator

Output: <u>A value of any type and complexity</u>

True or false?

Any operation that processes a list can be expressed in a terms of a fold, perhaps with a simple wrapper.

We can fold a list of anythings into anything!

Far-fetched folding:

Refrigerators in Gould-Simpson to
((grams fat, grams protein, grams carbs), calories)

Keyboards in Gould-Simpson to
[("a", #), ("b", #), ..., ("@2", #), ("CMD", #)]

[Backpack] to
(# pens, pounds of paper,
[(title, author, [page #s with the word "computer")])

[Furniture]
to a structure of 3D vertices representing a *convex hull* that could hold any single piece of furniture.

User-defined types

A **Shape** type

A new type can be created with a **data** declaration.

Here's a simple **Shape** type whose instances represent circles or rectangles:

```
data Shape =
   Circle Double |          -- just a radius
   Rect Double Double       -- width and height
          deriving Show
```

The shapes have dimensions but no position.

Circle and **Rect** are *data constructors*.

"**deriving Show**" declares **Shape** to be an instance of the **Show** type class, so that values can be shown using some simple, default rules.

Shape is called an *algebraic type* because instances of **Shape** are built using other types.

Shape, continued

Instances of Shape are created by calling the data constructors:

```
> let r1 = Rect 3 4
> r1
Rect 3.0 4.0

> let r2 = Rect 5 3

> let c1 = Circle 2

> let shapes = [r1, r2, c1]

> shapes
[Rect 3.0 4.0,Rect 5.0 3.0,Circle 2.0]
```

```
data Shape =
   Circle Double |
   Rect Double Double
      deriving Show
```

Lists must be homogeneous—why are both Rects and Circles allowed in the same list?

Shape, continued

The data constructors are just functions—we can use all our function-fu with them!

```
data Shape =
   Circle Double |
   Rect Double Double
      deriving Show
```

```
> :t Circle
Circle :: Double -> Shape

> :t Rect
Rect :: Double -> Double -> Shape

> map Circle [2,3] ++ map (Rect 3) [10,20]
[Circle 2.0,Circle 3.0,Rect 3.0 10.0,Rect 3.0 20.0]
```

Shape, continued

Functions that operate on algebraic types use patterns based on the type's data constructors.

```
area (Circle r) = r ** 2 * pi
area (Rect w h) = w * h
```

```
data Shape =
   Circle Double |
   Rect Double Double
      deriving Show
```

Usage:

```
> r1
Rect 3.0 4.0

> area r1
12.0

> shapes
[Rect 3.0 4.0,Rect 5.0 3.0,Circle 2.0]

> map area shapes
[12.0,15.0,12.566370614359172]

> sum $ map area shapes
39.56637061435917
```

Shape, continued

Let's make the **Shape** type an instance of the **Eq** type class.

What does **Eq** require?

```
> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Default definitions from **Eq**:

```
(==) a b = not $ a /= b
(/=)  a b = not $ a == b
```

We'll say that two shapes are equal if their areas are equal.

```
instance Eq Shape where
   (==) r1 r2 = area r1 == area r2
```

Usage:

```
> Rect 3 4 == Rect 6 2
True

> Rect 3 4 == Circle 2
False
```

Shape, continued

Let's see if we can find the biggest shape:

```
> maximum shapes
 No instance for (Ord Shape) arising from a use of
`maximum'
   Possible fix: add an instance declaration for (Ord
Shape)
```

What's in **Ord**?

```
> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

Eq a => Ord a requires would-be **Ord** classes to be instances of **Eq**. (Done!)

Like **==** and **/=** with **Eq**, the operators are implemented in terms of each other.

Shape, continued

Let's make Shape an instance of the Ord type class:

```
instance Ord Shape where
   (<) r1 r2 = area r1 < area r2         -- < and <= are sufficient
   (<=) r1 r2 = area r1 <= area r2
```

Usage:

```
> shapes
[Rect 3.0 4.0,Rect 5.0 3.0,Circle 2.0]

> map area shapes
[12.0,15.0,12.566370614359172]

> maximum shapes
Rect 5.0 3.0

> Data.List.sort shapes
[Rect 3.0 4.0,Circle 2.0,Rect 5.0 3.0]
```

Note that we didn't need to write functions like **sumOfAreas** or **largestShape**—we can express those in terms of existing operations

Shape all in one place

Here's all the **Shape** code: (in **shape.hs**)

```
data Shape =
    Circle Double |
    Rect Double Double deriving Show

area (Circle r) = r ** 2 * pi
area (Rect w h) = w * h

instance Eq Shape where
    (==) r1 r2 = area r1 == area r2

instance Ord Shape where
    (<) r1 r2 = area r1 < area r2
    (<=) r1 r2 = area r1 <= area r2
```

What would be needed to add a **Figure8** shape and a **perimeter** function?

How does this compare to a **Shape/Circle/Rect** hierarchy in Java?

The type Ordering

Let's look at the **compare** function:

```
> :t compare
compare :: Ord a => a -> a -> Ordering
```

Ordering is a simple algebraic type, with only three values:

```
> :info Ordering
data Ordering = LT | EQ | GT

> [r1,r2]
[Rect 3.0 4.0,Rect 5.0 3.0]

> compare r1 r2
LT

> compare r2 r1
GT
```

What is Bool?

What do you suppose Bool really is?

Bool is just an algebraic type with two values:

```
> :info Bool
data Bool = False | True
```

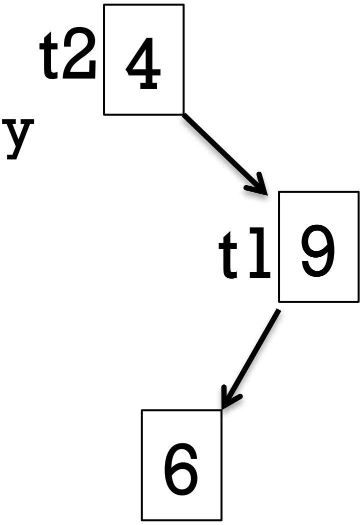
Bool is an example of Haskell's extensibility. Instead of being a primitive type, like boolean in Java, it's defined in terms of something more basic.

A binary tree

Here's an algebraic type for a binary tree: (in **tree.hs**)

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
              deriving Show
```

The **a** is a type variable. Our **Shape** type used **Double** values but Tree can hold values of any type!



```
> let t1 = Node 9 (Node 6 Empty Empty) Empty
> t1
Node 9 (Node 6 Empty Empty) Empty

> let t2 = Node 4 Empty t1
> t2
Node 4 Empty (Node 9 (Node 6 Empty Empty) Empty)
```

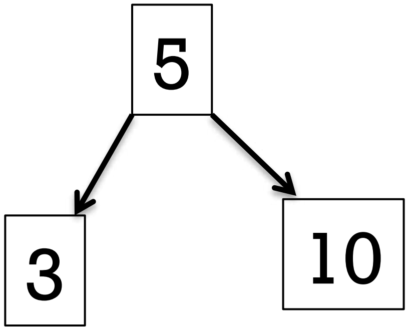
Tree, continued

Here's a function that inserts values, maintaining an ordered tree:

```
insert Empty v = Node v Empty Empty
insert (Node x left right) value
    | value <= x = (Node x (insert left value) right)
    | otherwise = (Node x left (insert right value))
```

Let's insert some values...

```
> let t = Empty
> insert t 5
Node 5 Empty Empty

> insert it 10
Node 5 Empty (Node 10 Empty Empty)

> insert it 3
Node 5 (Node 3 Empty Empty) (Node 10 Empty Empty)
```



Note that each insertion rebuilds some portion of the tree!

Tree, continued

Here's an in-order traversal that produces a list of values:

```
inOrder Empty = []
inOrder (Node val left right) =
    inOrder left ++ [val] ++ inOrder right
```

What's an easy way to insert a bunch of values?

```
> let t = foldl insert Empty [3,1,9,5,20,17,4,12]
> inOrder t
[1,3,4,5,9,12,17,20]

> inOrder $ foldl insert Empty "tim korb"
" bikmort"

> inOrder $ foldl insert Empty [Rect 3 4, Circle 1, Rect 1 2]
[Rect 1.0 2.0,Circle 1.0,Rect 3.0 4.0]
```

Maybe

Here's an interesting type:

```
> :info Maybe
data Maybe a = Nothing | Just a
```

Speculate: What's the point of it?

Here's a function that uses it:

```
> :t Data.List.find
Data.List.find :: (a -> Bool) -> [a] -> Maybe a
```

How could we use it?

```
> find even [3,5,6,8,9]
Just 6

> find even [3,5,9]
Nothing

> case (find even [3,5,9]) of { Just _ -> "got one"; _ -> "oops!"}
"oops!"
```

A little I/O

Sequencing

Consider this function declaration

```
f2 x = a + b + c
   where
      a = f x
      b = g x
      c = h x
```

```
a = f x
c = h x
b = g x
```

```
c = h x
b = g x
a = f x
```

Haskell guarantees that the order of the **where** clause bindings is inconsequential—those three lines can be in any order.

What enables that guarantee?

> (Pure) Haskell functions depend only on the argument value. For a given value of **x**, **f x** always produces the same result.

You can shuffle the bindings of any function's **where** clause without changing the function's behavior! (Try it with **longest**, slide 291.)

I/O and sequencing

Imagine a **getInt** function, which reads an integer from standard input (e.g., the keyboard).

Can the **where** clause bindings in the following function be done in any order?

```
f x = r
   where
      a = getInt
      b = getInt
      r = a * 2 + b + x
```

The following is not valid syntax but ignoring that, is it reorderable?

```
greet name = ""
   where
      putStr "Hello, "
      putStr name
      putStr "!\n"
```

I/O and sequencing, continued

One way we can specify that operations are to be performed in a specific sequence is to use a **do**:

```
% cat io2.hs
main = do
   putStrLn "Who goes there?"
   name <- getLine
   let greeting = "Hello, " ++ name ++ "!"
   putStrLn greeting
```

Interaction:

```
% runghc io2.hs
Who goes there?
whm (typed)
Hello, whm!
```

Actions

Here's the type of **putStrLn**:

```
    putStrLn :: String -> IO ()  ("unit", (), is the no-value value)
```

The type **IO x** represents an interaction with the outside world that produces a value of type **x**. Instances of **IO x** are called *actions*.

When an action is evaluated the corresponding outside-world activity is performed.

```
    > let hello = putStrLn "hello!"  (Note: no output here!)
    hello :: IO ()                   (Type of hello is an action.)

    > hello
    hello!      (Evaluating hello, an action, caused output.)
    it :: ()
```

Actions, continued

The value of **getLine** is an action that reads a line:

```
getLine :: IO String
```

We can evaluate the action, causing the line to be read, and bind a name to the string produced:

```
> s <- getLine
testing

> s
"testing"
```

Note that **getLine** is not a function!

Actions, continued

Recall **io2.hs**:

```
main = do
   putStrLn "Who goes there?"
   name <- getLine
   let greeting = "Hello, " ++ name ++ "!"
   putStrLn greeting
```

Note the type: **main :: IO ()**. We can say that **main** is an action. Evaluating **main** causes interaction with the outside world.

```
> main
Who goes there?
hello? (I typed)
Hello, hello?!
```

Is it pure?

A pure function (1) always produces the same result for a given argument value, and (2) has no side effects.

Is this a pure function?

```
twice :: String -> IO ()
twice s = do
   putStr s
   putStr s
```

twice "abc" will always produce the same value, an action that if evaluated will cause "**abcabc**" to be output.

The Haskell solution for I/O

We want to use pure functions whenever possible but we want to be able to do I/O, too.

In general, evaluating an action produces side effects.

Here's the Haskell solution for I/O in a nutshell:

Actions can evaluate other actions and pure functions but pure functions don't evaluate actions.

Recall **longest.hs**:

```
longest bytes = result where ...lots...
main = do
   args <- getArgs -- gets command line arguments
   bytes <- readFile (head args)
   putStr (longest bytes)
```

In conclusion...

If we had a whole semester...

If we had a whole semester to study functional programming, here's what might be next:

- More with infinite data structures (like `x = 1:x`)
- How lazy/non-strict evaluation works
- Implications and benefits of referential transparency (which means that the value of a given expression is always the same).
- Functors (structures that can be mapped over)
- Monoids (a set of things with a binary operation over them)
- Monads (for representing sequential computations)
- Zippers (a structure for traversing and updating another structure)
- And more!

Jeremiah Nelson and Jordan Danford are great local resources for Haskell!

Even if you never use Haskell again...

Recursion and techniques with higher-order functions can be used in most languages. Some examples:

JavaScript, Python, PHP, all flavors of Lisp, and lots of others:
Functions are "first-class" values; anonymous functions are supported.

C
Pass a function pointer to a recursive function that traverses a data structure.

C#
Excellent support for functional programming with the language itself, and LINQ, too.

Java 8
Lambda expressions are in!

OCaml
"an industrial strength programming language supporting functional, imperative and object-oriented styles" – **OCaml.org**
http://www.ffconsultancy.com/languages/ray_tracer/comparison.html

Killer Quiz!

Ruby

CSC 372, Spring 2016
The University of Arizona
William H. Mitchell
whm@cs

The Big Picture

Our topic sequence:

- Functional programming with Haskell (Done!)
- Imperative and object-oriented programming using dynamic typing with Ruby
- Logic programming with Prolog
- Whatever else in the realm of programming languages that we find interesting and have time for.

Introduction

From: Ralph Griswold <ralph@CS.Arizona.EDU>
Date: Mon, 18 Sep 2006 16:14:46 -0700

whm wrote:
> I ran into John Cropper in the mailroom a few minutes ago. He said
> he was out at your place today and that you're doing well. I
> understand you've got a meeting coming up regarding math in your
> weaving book -- sounds like fun!?

Hi, William

I'm doing well in the sense of surviving longer than expected. But
I'm still a sick person without much energy and with a lot of pain.
>
> My first lecture on Ruby is tomorrow. Ruby was cooked up by a
> Japanese fellow. Judging by the number of different ways to do the
> same thing, I wonder if Japanese has a word like "no".

Interesting. I know nothing about Ruby, but I've noticed it's
getting a lot of press, so there must be something to it.

Ralph's obituary: http://cs.arizona.edu/news/articles/200610-griswold.html

What is Ruby?

"A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write." — `ruby-lang.org`

Ruby is commonly described as an "object-oriented scripting language".

(I don't like the term "scripting language"!)

I describe Ruby as a dynamically typed object-oriented language.

Ruby was invented by Yukihiro Matsumoto ("Matz"), a "Japanese amateur language designer", in his own words.

Ruby on Rails, a web application framework, has largely driven Ruby's popularity.

Matz says...

Here is a second-hand excerpt of a posting by Matz:

> "Well, Ruby was born on February 24, 1993. I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising."

Another quote from Matz:

> "I believe that the purpose of life is, at least in part, to be happy. Based on this belief, Ruby is designed to make programming not only easy but also fun. It allows you to concentrate on the creative side of programming, with less stress. If you don’t believe me, read this book [the "pickaxe" book] and try Ruby. I’m sure you’ll find out for yourself."

Version issues

There is no written standard for Ruby. The language is effectively defined by MRI—Matz' Ruby Implementation.

The most recent stable version of MRI is 2.3.0.

The default version of Ruby on lectura is 1.8.7 but we'll use **`rvm`** (the Ruby Version Manager) to run version 2.2.4.

OS X, from Mavericks to El Capitan, has Ruby 2.0.0.

The last major upheaval in Ruby occurred between 1.8 and 1.9.

In general, there are few incompatibilities between 1.9.3 and the latest version.

The examples in these slides should work with with 1.9.3 through 2.3.0.

Resources

The Ruby Programming Language by David Flanagan and Matz

- Perhaps the best book on Safari that covers 1.9 (along with 1.8)
- I'll refer to it as "RPL".

Programming Ruby 1.9 & 2.0 (4th edition): *The Pragmatic Programmers' Guide* by Dave Thomas, with Chad Fowler and Andy Hunt

- Known as the "Pickaxe book"
- $28 for a DRM-free PDF at pragprog.com.
- I'll refer to it as "PA".
- First edition is here: http://ruby-doc.com/docs/ProgrammingRuby/

Safari has lots of pre-1.9 books, lots of books that teach just enough Ruby to get one into the water with Rails, and lots of "cookbooks".

Resources, continued

ruby-lang.org

- Ruby's home page

ruby-doc.org

- Documentation

- Here's a sample URL, for the **String** class in 2.2.4:
  http://ruby-doc.org/core-2.2.4/String.html

- Suggestion: Create a Chrome "search engine" named **rc** ("Ruby class") with this expansion:
  http://www.ruby-doc.org/core-2.2.4/%s.html
  (See http://www.cs.arizona.edu/~whm/o1nav.pdf)

Getting Ruby for OS X

Ruby 2.0.0, as supplied by Apple with recent versions of OS X, should be fine for our purposes.

I installed Ruby 2.2.0 on my Mac using MacPorts. The "port" is `ruby22`.

Lot of people install Ruby versions using the Homebrew package manager, too.

Getting Ruby for Windows

Go to **http://rubyinstaller.org/downloads/** and get
"**Ruby 2.2.4**" (not x64)

When installing, I recommend these selections:

Add Ruby executables to your PATH

Associate **.rb** and **.rbw** files with this Ruby installation

Running Ruby

rvm—Ruby Version Manager

rvm is the Ruby Version Manager. It lets one easily select a particular version of Ruby to work with.

On lectura, we can select Ruby 2.2.4 and then check the version like this:

```
% rvm 2.2.4
% ruby --version
ruby 2.2.4p230 (2015-12-16 revision 53155) [x86_64-linux]
```

Depending on your bash configuration, **rvm** may produce a message like "**Warning! PATH is not properly set up...**" but if **ruby --version** shows **2.2.4**, all is well.

Note: **rvm** does not work with **ksh**. If you're running **ksh**, let us know.

rvm, continued

IMPORTANT: you must either

1. Do **rvm 2.2.4** each time you login on lectura.

   —OR—

2. Add the command **rvm 2.2.4** to one of your bash start-up files.

There are a variety of ways in which bash start-up files can be configured.

- With the default configuration for CS accounts, add the line

  ```
  rvm 2.2.4 >& /dev/null
  ```

  at the end of your **~/.profile**.

- If you're using the configuration suggested in my Fall 2015 352 slides, put that line at the end of your **~/.bashrc**.

- Let us know if you have trouble with this.

irb—Interactive Ruby Shell

The **irb** command provides a REPL for Ruby.

irb can be run with no arguments but I usually start **irb** with a **bash** alias that specifies using a simple prompt and activates auto-completion:

```
alias irb="irb --prompt simple -r irb/completion"
```

When **irb** starts up, it first processes **~/.irbrc**, if present.

spring16/ruby/dotirbrc is a recommended starter **~/.irbrc** file.

```
% cp /cs/www/classes/cs372/spring16/ruby/dotirbrc  ~/.irbrc
```

Control-D terminates **irb**.

irb, continued

irb evaluates expressions as they are typed.

```
% irb
>> 1+2
=> 3

>> "testing" + "123"
=> "testing123"
```

Assuming you're using the ~/.irbrc suggested on the previous slide, you can use "**it**" to reference the last result:

```
>> it
=> "testing123"

>> it + it
=> "testing123testing123"
```

irb, continued

A couple more:

```
>> `ssh lec uptime`
=> " 18:00:58 up 10 days,  9:00, 99 users,  load average: 0.50,
0.32, 0.32\n"

>> it[-26,8]
=> "average:"
```

If an expression is definitely incomplete, irb displays an alternate prompt:

```
>> 1.23 +
?> 2e3
=> 2001.23
```

Note: To save space on the slides I'll typically not show the result line (=> ...) when it's uninteresting.

Ruby basics

Every value is an object

In Ruby every value is an object.

Methods can be invoked using *receiver.method*(*parameters...*)

```
>> "testing".count("t")     # How many "t"s are there?
 => 2

>> "testing".slice(1,3)
=> "est"

>> "testing".length()
=> 7
```

Repeat: In Ruby every value is an object.

What are some values in Java that are not objects?

Everything is an object, continued

Of course, "everything" includes numbers:

```
>> 1.2.class()
=> Float

>> (10-20).class()
=> Fixnum

>> 17**25
=> 5770627412348402378939569991057

>> it.succ()     # Remember: the custom .irbc is needed to use "it"
=> 5770627412348402378939569991058

>> it.class()
=> Bignum
```

Everything is an object, continued

The TAB key can be used to show completions:

```
>> 100.<TAB><TAB>
Display all 107 possibilities? (y or n)
100.__id__                      100.display
100.__send__                    100.div
100.abs                         100.divmod
100.abs2                        100.downto
100.angle                       100.dup
100.arg                         100.enum_for
100.between?                    100.eql?
100.ceil                        100.equal?
100.chr                         100.even?
100.class                       100.extend
100.clone                       100.fdiv
100.coerce                      100.floor
100.conj                        100.freeze
100.conjugate                   100.frozen?
100.define_singleton_method     100.gcd
100.denominator                 100.gcdlcm
```

Parentheses are optional, sometimes

Parentheses are often optional in method invocations:

```
>> 1.2.class
=> Float

>> "testing".count "aeiou"
=> 2
```

But, the following case fails. (Why?)

```
>> "testing".count "aeiou".class
TypeError: no implicit conversion of Class into String
        from (irb):17:in `count'
```

Solution:

```
>> "testing".count("aeiou").class
=> Fixnum
```

I usually omit parentheses in simple method invocations.

A post-Haskell hazard!

Don't let the optional parentheses make you have a Haskell moment and leave out a comma between arguments:

```
>> "testing".slice 2 3
SyntaxError: (irb):20: syntax error, unexpected tINTEGER,
expecting end-of-input
```

Commas are required between arguments!

```
>> "testing".slice 2,3
=> "sti"
```

Operators are methods, too

Ruby operators are methods with symbolic names.

In general,

expr1 op expr2

means

expr1.op(expr2)

Example:

```
>> 3 + 4
=> 7

>> 3.+(4)
=> 7

>> "abc".==(97.chr.+("bc"))
=> true
```

Kernel methods

The **Kernel** module has methods for I/O and more. Methods in **Kernel** can be invoked with only the method name.

```
>> puts "hello"
hello
=> nil

>> printf "sum = %d, product = %d\n", 3+4, 3 * 4
sum = 7, product = 12
=> nil

>> puts gets.inspect
testing
"testing\n"
=> nil
```

See **http://ruby-doc.org/core-2.2.4/Kernel.html**

Extra Credit Assignment 2

For two assignment points of extra credit:

1. Run `irb` somewhere and try ten Ruby expressions with some degree of variety.

2. Capture the output and put it in a plain text file, eca2.txt. No need for your name, NetID, etc. in the file. No need to edit out errors.

3. On lectura, turn in eca2.txt with the following command:

```
% turnin 372-eca2 eca2.txt
```

Due: At the start of the next lecture after we hit this slide.

Needless to say, feel free to read ahead in the slides and show experimentation with the following material, too.

Executing Ruby code in a file

The **ruby** command can be used to execute Ruby source code contained in a file.

By convention, Ruby files have the suffix **.rb**.

Here is "Hello" in Ruby:

```
% cat hello.rb
puts "Hello, world!"

% ruby hello.rb
Hello, world!
```

Note that the code does not need to be enclosed in a method—"top level" expressions are evaluated when encountered.

Executing Ruby code in a file, continued

Alternatively, code can be placed in a method that is invoked by an expression at the top level:

```
% cat hello2.rb
def say_hello
   puts "Hello, world!"
end

say_hello

% ruby hello2.rb
Hello, world!
```

The definition of **say_hello** must precede the call.

We'll see later that Ruby is somewhat sensitive to newlines.

A line-numbering program

Here's a program that reads lines from standard input and writes each, with a line number, to standard output:

```
line_num = 1        # numlines.rb

while line = gets
    printf("%3d: %s", line_num, line)
    line_num += 1    # Ruby does not have ++ and --
end
```

Execution:

```
% ruby numlines.rb < hello2.rb
  1: def say_hello
  2:     puts "Hello, world!"
  3: end
  4:
  5: say_hello
```

tac.rb

Problem: Write a program that reads lines from standard input and writes them in reverse order to standard output. Use only the Ruby you've seen.

For reference, here's the line-numbering program:

```
line_num = 1
while line = gets
   printf("%3d: %s", line_num, line)
   line_num += 1
end
```

Solution: (spring16/ruby/tac.rb)

```
reversed = ""
while line = gets
   reversed = line + reversed
end
puts reversed
```

Some basic types

The value **nil**

nil is Ruby's "no value" value. The name **nil** references the only instance of the class.

```
    >> nil
    => nil

    >> nil.class
    => NilClass

    >> nil.object_id
    => 4
```

TODO

```
    x = 1 if false
    p x # outputs nil
```

It seems like the presence of an assignment for x causes a reference to x to produce nil if no assignment is ever done.

We'll see that Ruby uses **nil** in a variety of ways.

Speculate: Do uninitialized variables have the value **nil**?

```
    >> x
    NameError: undefined local variable or method `x' for main
```

Strings and string literals

Instances of Ruby's **String** class represent character strings.

A variety of "escapes" are recognized in double-quoted string literals:

```
>> puts "newline >\n< and tab >\t<"
newline >
< and tab >    <

>> "\n\t\\".length
=> 3

>> "Newlines: octal \012, hex \xa, control-j \cj"
=> "Newlines: octal \n, hex \n, control-j \n"
```

Section 3.2, page 49 in RPL has the full list of escapes.

String literals, continued

In single-quoted literals only `\'` and `\\` are recognized as escapes:

```
>> puts '\n\t'
\n\t
=> nil

>> '\n\t'.length    # Four chars: backslash, n, backslash, t
=> 4

>> puts '\'\\'
'\
=> nil

>> '\'\\'.length # Two characters: apostrophe, backslash
=> 2
```

String has a lot of methods

The **public_methods** method shows the public methods that are available for an object. Here are some of the methods for **String**:

```
>> "abc".public_methods.sort
=> [:!, :!=, :!~, :%, :*, :+, :<, :<<, :<=, :<=>, :==, :===, :=~,
 :>, :>=, :[], :[]=, :__id__, :__send__, :ascii_only?,
 :between?, :bytes, :bytesize, :byteslice, :capitalize, :capitalize!
, :casecmp, :center, :chars, :chomp, :chomp!, :chop, :chop!, :chr
, :class, :clear, :clone, :codepoints, :concat, :count, :crypt, :defi
ne_singleton_method, :delete, :delete!, :display, :downcase, :d
owncase!, :dump, :dup, :each_byte, :each_char, :each_codepoi
nt, :each_line, :empty?, ...

>> "abc".public_methods.length
=> 169
```

Strings are mutable

Unlike Java, Haskell, and many other languages, strings in Ruby are mutable.

If two variables reference a string and the string is changed, the change is reflected by both variables:

```
>> x = "testing"

>> y = x          # x and y now reference the same instance of String

>> y << " this"          # the << operator appends a string
=> "testing this"

>> x
=> "testing this"
```

Is it a good idea to have mutable strings?

Strings are mutable, continued

The **dup** method produces a copy of a string.

```
>> x = "testing"
>> y = x.dup
=> "testing"

>> y << "...more"
=> "testing...more"

>> y
=> "testing...more"

>> x
=> "testing"
```

Some objects that hold strings **dup** the string when the string is added to the object.

Sidebar: applicative vs. imperative methods

Some methods have both an *applicative* and an *imperative* form.

String's **upcase** method is applicative—it produces a new **String** but doesn't change its *receiver*, the instance of **String** on which it's called:

```
>> s = "testing"
=> "testing"

>> s.upcase
=> "TESTING"

>> s
=> "testing"
```

applicative vs. imperative methods, contineud

In contrast, an imperative method potentially changes its receiver.

String's **upcase!** method is the imperative counterpart to **upcase**:

```
>> s.upcase!
=> "TESTING"

>> s
=> "TESTING"
```

A Ruby convention is that when methods have both an applicative and an imperative form, the imperative form ends with an exclamation mark.

String comparisons

Strings can be compared with a typical set of operators:

```
>> s1 = "apple"

>> s2 = "testing"

>> s1 == s2
=> false

>> s1 != s2
=> true

>> s1 < s2
=> true
```

We'll talk about details of **true** and **false** later.

String comparisons, continued

There is also a *comparison operator*: **<=>**

It produces -1, 0, or 1 depending on whether the first operand is less than, equal to, or greater than the second operand.

```
>> "apple" <=> "testing"
=> -1

>> "testing" <=> "apple"
=> 1

>> "x" <=> "x"
=> 0
```

This operator is sometimes read as "spaceship".

Substrings

Subscripting a string with a number produces a one-character string.

```
>> s="abcd"

>> s[0]          # Positions are zero-based
=> "a"

>> s[1]
=> "b"

>> s[-1]         # Negative positions are counted from the right
=> "d"

>> s[100]
=> nil           # An out-of-bounds reference produces nil
```

Historical note: With Ruby versions prior to 1.9, `"abc"[0]` is 97.

Why doesn't Java provide `s[n]` instead of `s.charAt(n)`?

Substrings, continued

A subscripted string can be the target of an assignment. A string of any length can be assigned.

```
>> s = "abc"
=> "abc"

>> s[0] = 65.chr
=> "A"

>> s[1] = "tomi"

>> s
=> "Atomic"

>> s[-3] = ""

>> s
=> "Atoic"
```

Substrings, continued

A substring can be referenced with

s[*start*, *length*]

```
r e p l a c e
0 1 2 3 4 5 6
7 6 5 4 3 2 1  (negative)
```

```
>> s = "replace"

>> s[2,3]
=> "pla"

>> s[3,100]        # Note too-long behavior!
=> "lace"

>> s[-4,3]
=> "lac"

>> s[10,10]
=> nil
```

Substrings with ranges

Instances of Ruby's **Range** class represent a range of values. A **Range** can be used to reference a substring.

```
>> r = 2..-2
=> 2..-2

>> r.class
=> Range

>> s = "replaced"

>> s[r]
=> "place"

>> s[r] = ""

>> s
=> "red"
```

Substrings with ranges, continued

It's more common to use literal ranges with strings:

```
>> s = "rebuilding"
>> s[2..-1]
=> "building"
```

r	**e**	**b**	**u**	**i**	**l**	**d**	**i**	**n**	**g**	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	
10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	***(negative)***

```
>> s[2..-4]
=> "build"

>> s[2...-4]     # three dots is "up to"
=> "buil"

>> s[-8..-4]
=> "build"

>> s[-4..-8]
=> ""
```

Changing substrings

A substring can be the target of an assignment:

```
>> s = "replace"

>> s[0,2] = ""
=> ""

>> s
=> "place"

>> s[3..-1] = "naria"
>> s
=> "planaria"

>> s["aria"] = "kton"  # If "aria" appears, replace it (error if not).
=> "kton"

>> s
=> "plankton"
```

```
r e p l a c e
0 1 2 3 4 5 6
7 6 5 4 3 2 1  (negative)
```

```
p l a c e
0 1 2 3 4
5 4 3 2 1  (negative)
```

Interpolation in string literals

In a string literal enclosed with double quotes the sequence #{*expr*} causes interpolation of *expr*, an arbitrary Ruby expression.

```
>> x = 10

>> y = "twenty"

>> s = "x = #{x}, y + y = #{y + y}"
=> "x = 10, y + y = twentytwenty"

>> puts "There are #{"".public_methods.length} string methods"
There are 169 string methods

>> "test #{"#{"abc".length*4}"}"     # Arbitrary nesting works
=> "test 12"
```

It's idiomatic to use interpolation rather than concatenation to build a string from multiple values.

Numbers

With 2.2.4 on lectura, integers in the range -2^{62} to $2^{62}-1$ are represented by instances of **Fixnum**. If an operation produces a number outside of that range, the value is represented with a **Bignum**.

```
>> x = 2**62-1
=> 4611686018427387903

>> x.class          => Fixnum

>> x += 1           => 4611686018427387904

>> x.class          => Bignum

>> x -= 1           => 4611686018427387903

>> x.class          => Fixnum
```

LHtLaL:
Explore boundaries!

Is this automatic transitioning between **Fixnum** and **Bignum** a good idea? How do other languages handle this?

Numbers, continued

The **Float** class represents floating point numbers that can be represented by a double-precision floating point number on the host architecture.

```
>> x = 123.456
=> 123.456

>> x.class
=> Float

>> x ** 0.5
=> 11.111075555498667

>> x = x / 0.0
=> Infinity

>> (0.0/0.0).nan?
=> true
```

Numbers, continued

Arithmetic on two **Fixnum**s produces a **Fixnum**.

```
>> 2/3
=> 0

>> it.class
=> Fixnum
```

Fixnums and **Float**s can be mixed. The result is a **Float**.

```
>> 10 / 5.1
=> 1.9607843137254903

>> 10 % 4.5
=> 1.0

>> it.class
=> Float
```

Numbers, continued

Ruby has a **Complex** type.

```
>> x = Complex(2,3)
=> (2+3i)

>> x * 2 + 7
=> (11+6i)

>> Complex 'i'
=> (0+1i)

>> it ** 2
=> (-1+0i)
```

Numbers, continued

There's **Rational**, too.

```
>> Rational(1,3)
=> (1/3)

>> it * 300
=> (100/1)

>> Rational 0.5
=> (1/2)

>> Rational 0.6
=> (5404319552844595/9007199254740992)

>> Rational 0.015625
=> (1/64)
```

Conversions

Unlike some languages, Ruby does not automatically convert strings to numbers and numbers to strings as needed.

```
>> 10 + "20"
TypeError: String can't be coerced into Fixnum
```

The methods **to_i**, **to_f**, and **to_s** are used to convert values to **Fixnum**s, **Float**s and **String**s, respectively.

```
>> 10.to_s + "20"
=> "1020"

>> 10 + "20".to_f
=> 30.0

>> 10 + 20.9.to_i
=> 30
```

```
>> 33.to_<TAB><TAB>
33.to_c        33.to_int
33.to_enum     33.to_r
33.to_f        33.to_s
33.to_i
```

Arrays

A sequence of values is typically represented in Ruby by an instance of `Array`.

An array can be created by enclosing a comma-separated sequence of values in square brackets:

```
>> a1 = [10, 20, 30]
=> [10, 20, 30]

>> a2 = ["ten", 20, 30.0, 2**40]
=> ["ten", 20, 30.0, 1099511627776]

>> a3 = [a1, a2, [[a1]]]
=> [[10, 20, 30], ["ten", 20, 30.0, 1099511627776], [[[10, 20, 30]]]]
```

What's a difference between Ruby arrays and Haskell lists?

Arrays, continued

Array elements and subarrays (sometimes called slices) are specified with a notation like that used for strings.

```
>> a = [1, "two", 3.0, %w{a b c d}]
=> [1, "two", 3.0, ["a", "b", "c", "d"]]

>> a[0]
=> 1

>> a[1,2]              # a[start, length]
=> ["two", 3.0]

>> a[-1]
=> ["a", "b", "c", "d"]

>> a[-1][-2]
=> "c"
```

Arrays, continued

Elements and subarrays can be assigned to. Ruby accommodates a variety of cases; here are some:

```
>> a = [10, 20, 30, 40, 50, 60]

>> a[1] = "twenty"; a
=> [10, "twenty", 30, 40, 50, 60]

>> a[2..4] = %w{a b c d e}; a
=> [10, "twenty", "a", "b", "c", "d", "e", 60]

>> a[1..-1] = []; a
=> [10]

>> a[0] = [1,2,3]; a                => [[1, 2, 3]]

>> a[4] = [5,6]; a                  => [[1, 2, 3], nil, nil, nil, [5, 6]]

>> a[0,2] = %w{a bb ccc}; a  => ["a", "bb", "ccc", nil, nil, [5, 6]]
```

Arrays, continued

A variety of operations are provided for arrays. Here's a sampling:

```
>> a = []

>> a << 1; a
=> [1]

>> a << [2,3,4];  a
=> [1, [2, 3, 4]]

>> a.reverse!;  a
=> [[2, 3, 4], 1]
```

Arrays, continued

A few more:

```
>> a
=> [[2, 3, 4], 1]

>> a[0].shift
=> 2

>> a
=> [[3, 4], 1]

>> a.unshift "a","b","c"
=> ["a", "b", "c", [3, 4], 1]

>> a.shuffle.shuffle
=> ["a", [3, 4], "b", "c", 1]
```

Arrays, continued

Even more!

```
>> a = [1,2,3,4]; b = [1,3,5]

>> a + b
=> [1, 2, 3, 4, 1, 3, 5]

>> a - b
=> [2, 4]

>> a & b
=> [1, 3]

>> a | b
=> [1, 2, 3, 4, 5]

>> ('a'..'zzz').to_a.size
=> 18278
```

Comparing arrays

We can compare arrays with **==** and **!=**. Elements are compared in turn, possibly recursively.

```
>> [1,2,3] == [1,2]
=> false

>> [1,2,[3,"bcd"]] == [1,2] + [[3, "abcde"]]
=> false

>> [1,2,[3,"bcd"]] == [1,2] + [[3, "abcde"[1..-2]]]
=> true
```

Comparing arrays

Comparison of arrays with **<=>** is lexicographic.

```
>> [1,2,3,4] <=> [1,2,10]
=> -1

>> [[10,20],[2,30], [5,"x"]].sort
=> [[2, 30], [5, "x"], [10, 20]]
```

Comparing arrays

Comparison with **<=>** produces **nil** if differing types are encountered.

```
>> [1,2,3,4] <=> [1,2,3,"four"]
=> nil
```

Tie!

```
>> [[10,20],[5,30], [5,"x"]].sort
ArgumentError: comparison of Array with Array failed
```

Here's a simpler failing case. Should it be allowed?

```
>> ["sixty",20,"two"].sort
ArgumentError: comparison of String with 20 failed
```

Comparing arrays, continued

At hand:

```
>> ["sixty",20,"two"].sort
ArgumentError: comparison of String with 20 failed
```

Contrast with Icon:

```
][ sort(["sixty",20,"two"])
  r := [20,"sixty","two"] (list)

][ sort([3.0, 7, 2, "a", "A", ":", [2], [1], -1.0])
  r := [2, 7, -1.0, 3.0, ":", "A", "a", [2], [1]]  (list)
```

What does Icon do better? What does Icon do worse?

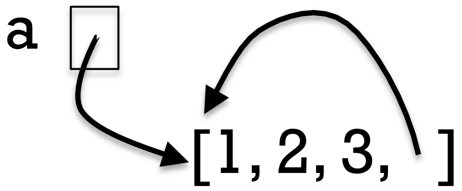
Here's Python 2:

```
>>> sorted([3.0, 7, 2, "a", "A", ":", [2], [1], -1.0])
[-1.0, 2, 3.0, 7, [1], [2], ':', 'A', 'a']
```

Arrays can be cyclic

An array can hold a reference to itself:

```
>> a = [1,2,3]

>> a.push a
=> [1, 2, 3, [...]]

>> a.size
=> 4

>> a[-1]
=> [1, 2, 3, [...]]

>> a[-1][-1][-1]
=> [1, 2, 3, [...]]
```



a

[1, 2, 3,]

```
>> a << 10
=> [1, 2, 3, [...], 10]

>> a[-2][-1]
=> 10
```

Type Checking

Static typing

"The Java programming language is a statically typed language, which means that every variable and every expression has a type that is known at compile time."

-- *The Java Language Specification, Java SE 7 Edition*

Assume the following:

```
int i = ...;   String s = ...;   Object o = ...;   static int f(int n);
```

What are the types of the following expressions?

```
i + 5
i + s
s + o
o + o
o.hashCode()
f(i.hashCode())
i = i + s
```

Did we need to know any values or execute any code to determine those types?

Static typing, continued

Java does type checking based on the declared types of variables and the intrinsic types of literals.

Haskell supports type declarations but also provides type inferencing.

What are the inferred types for **x**, **y**, and **z** in the following expression?

```
(isLetter $ head $ [x] ++ y) && z
```

```
> let f x y z = (isLetter $ head $ [x] ++ y) && z
f :: Char -> [Char] -> Bool -> Bool
```

Did we need to know any values or execute any code to determine those types?

Haskell is a statically typed language—the type of every expression can be determined by analyzing the code.

Static typing, continued

With a statically typed language, the type for all expressions is determined when a body of code is compiled/loaded/etc. Any type inconsistencies that exist are discoverable at that time.

Without having to run any code a statically typed language lets us guarantee that various types of errors don't exist. Examples:

Dividing a string by a float
Taking the "head" of an integer
Concatenating two numbers
Putting an integer in a list of strings

Static typing, continued

How often did your Haskell code run correctly as soon as the type errors were fixed?

How does that compare with your experience with Java?
With C?
With Python?

"The best news is that Haskell's type system will tell you if your program is well-typed before you run it. This is a big advantage because most programming errors are manifested as typing errors."—Paul Hudak, Yale

Do you agree with Hudak?

Variables in Ruby have no type

In Java, variables are declared to have a type.

Variables in Ruby do not have a type. Instead, type is associated with values.

```
>> x = 10
>> x.class              # What's the class of the object held in x?
=> Fixnum

>> x = "ten"
>> x.class
=> String

>> x = 2**100
>> x.class
=> Bignum
```

Dynamic typing

Ruby is a dynamically typed language. **There is no static analysis of the types involved in expressions**.

Consider this Ruby method:

```
def f x, y, z
   return x[y + z] * x.foo
end
```

For some combinations of types it will produce a value. For others it will produce a `TypeError`.

With dynamic typing such methods are allowed to exist.

Dynamic typing, continued

With dynamic typing, no type checking is done when code is compiled. Instead, types of values are checked during execution, as each operation is performed.

Consider this Ruby code:

```
while line = gets
   puts(f(line) + 3 + g(line)[-2])
end
```

What types must be checked each time through that loop?

Wrt. static typing, what are the implications of dynamic typing for...

Compilation speed?

Probably faster!

Execution speed?

Probably slower!

Reliability?

It depends...

Can testing compensate?

A long-standing question in industry:

Can a good test suite find type errors in dynamically typed code as effectively as static type checking?

What's a "good" test suite?

Full code coverage? (every line executed by some test)
Full path coverage? (all combinations of paths exercised)
How about functions whose return type varies?

But wouldn't we want a good test suite no matter what language we're using?

"Why have to write tests for things a compiler can catch?"
—Brendan Jennings, SigFig

What ultimately matters?

What does the end-user of software care about?

Software that works

Facebook game vs. radiation therapy system

Fast enough

When does 10ms vs. 50ms matter?

Better sooner than later

A demo that's a day late for a trade show isn't worth much.

Affordable

How much more would you pay for a version of your favorite game that has half as many bugs?

I'd pay A LOT for a version of PowerPoint with more keyboard shortcuts!

Variety in type checking

Java is statically typed but casts introduce the possibility of a type error not being detected until execution.

C is statically typed but has casts that allow type errors during execution that are never detected.

Ruby, Python, and Icon have no static type checking whatsoever, but type errors during execution are always detected.

An example of a typing-related trade-off in execution time:

- C spends zero time during execution checking types.
- Java checks types during execution only in certain cases.
- Languages with dynamic typing check types on every operation, at least conceptually.

Is type inferencing applicable in a dynamically typed language?

UA CS TR 93-32a: Type Inference in the Icon Programming Language

"Why?" vs. "Why Not?"

"Why?" or "Why not?"

When designing a language some designers ask,
"Why should feature X be included?"

Some designers ask the opposite:
"Why should feature X not be included?"

Let's explore that question with Ruby.

More string literals!

A "here document" is a third way to literally specify a string.

```
>> s = <<XYZZY
    +------+
    | \\\ |
    | \*/ |
    | ''' |
    +------+
XYZZY
=> "\n    +------+\n\n    | \\ |\n\n    | */ |\n
\n    | ''' |\n\n    +------+\n\n"
```

The string following **<<** specifies a delimiter that ends the literal. **The ending occurrence must be at the start of a line.**

"There's more than one way to do it!"—a Perl motto

And that's not all!

Here's another way to specify string literals. See if you can discern some rules from these examples:

```
>> %q{ just testin' this... }
=> " just testin' this... "

>> %Q|\n\t|
=> "\n\t"

>> %q(\u0041 is Unicode for A)
=> "\\u0041 is Unicode for A"

>> %q.test.
=> "test"
```

%q follows single-quote rules. %Q follows double quote rules. Symmetrical pairs like (), {}, and <> can be used.

How much is enough?

Partial summary of string literal syntax in Ruby:

```
>> x = 5; s = "x is #{x}"
=> "x is 5"

>> '\'\\\n\t'.length
=> 6

>> hd = <<X
just
testing
X
=> "just\ntesting\n"
```

How many ways does Haskell have to make a string literal?

How many ways should there be to make a string literal?

What's the minimum functionality needed?

Which would you remove?

```
>> %q{ \n \t } + %Q|\n \t | + %Q(\u0021 \u{23})
=> " \\n \\t \n \t ! #"
```

"Why" or "Why not?" as applied to operator overloading

Here are some examples of operator overloading:

```
>> [1,2,3] + [4,5,6] + [ ] + [7]
=> [1, 2, 3, 4, 5, 6, 7]

>> "abc" * 5
=> "abcabcabcabcabc"

>> [1, 3, 15, 1, 2, 1, 3, 7] - [3, 2, 1, 3]
=> [15, 7]

>> [10, 20, 30] * "..."
=> "10...20...30"        # "intercalation"

>> "decimal: %d, octal: %o, hex: %x" % [20, 20, 20]
=> "decimal: 20, octal: 24, hex: 14"
```

"Why" or "Why not?", continued

What are some ways in which inclusion of a feature impacts a language?

- Increases the "mental footprint" of the language.
  - There are separate footprints for reading code and writing code.
- Maybe makes the language more expressive.
- Maybe makes the language useful for new applications.
- Probably increases size of implementation and documentation.
- Might impact performance.

Features come in all sizes!

Features come in all sizes!

Small: A new string literal escape sequence ("`\U{65}`" for "`A`")

Small: Supporting an operator on a new pair of types

Medium: Support for arbitrary precision integers

Large or small?

Support for object-oriented programming

Support for garbage collection

What would Ralph do?

At one of my first meetings with Ralph Griswold I put forth a number of ideas I had for new features for Icon.

He listened patiently. When I was done he said,
"Go ahead. Add all of those you want to."

As I left his office he added,
"But for every feature you add, first find one to remove."

The art of language design

There's a lot of science in programming language design but there's art, too.

Excerpt from interview with Perl Guru Damian Conway:

Q: "What languages other than Perl do you enjoy programming in?"

A: "I'm very partial to Icon. It's so beautifully put together, so elegantly proportioned, almost like a Renaissance painting."

http://www.pair.com/pair/current/insider/1201/damianconway.html (404 now!)

"Icon: A general purpose language known for its elegance and grace. Designed by Ralph Griswold to be successor to SNOBOL4."
—Digibarn "Mother Tongues" chart (see Intro slides)

Between SNOBOL4 and Icon there was there SL5 (SNOBOL Language 5).

I think of SL5 as an example of the "Second System Effect". It was never released.

Ralph once said, "I was laying in the hospital thinking about SL5. I felt there must be something simpler." That turned out to be Icon.

Design example: invocation in Icon

Procedure call in Icon:

```
    ][ reverse("programming")
      r := "gnimmargorp"  (string)

    ][ p := reverse
      r := function reverse  (procedure)

    ][ p("foo")
      r := "oof"  (string)
```

Doctoral student Steve Wampler added mutual goal directed evaluation (MGDE). A trivial example:

```
    ][ 3("one", 2, "III")
      r := "III"  (string)

    ][ (?3)("one", 2, "III")
      r := "one"  (string)
```

Invocation in Icon, continued

After a CSC 550A lecture where Ralph introduced MGDE, I asked,
"How about 'string invocation', so that `"+"(3,4)` would be `7`?"

What do you suppose Ralph said?
"How would we distinguish between unary and binary operators?"

Solution: Discriminate based on the operand count!

```
][ "-"(5,3)
  r := 2  (integer)
][ "-"(5)
  r := -5  (integer)
][ (?"+-")(3,4)
  r := -1  (integer)
```

Within a day or two I added string invocation to Icon.

Why did Ralph choose to allow this feature?
He felt it would increase the research potential of Icon.

Design example: Parallel assignment

An interesting language design example in Ruby is *parallel assignment*. Some simple examples:

```
>> a, b = 10, [20, 30]

>> a
=> 10

>> b
=> [20, 30]

>> c, d = b
>> c
=> 20

>> d
=> 30
```

Parallel assignment, continued

Could we do a swap with parallel assignment?

```
>> x, y = 10, 20

>> x,y = y,x

>> x
=> 20

>> y
=> 10
```

This swaps, too:

```
>> x,y=[y,x]
```

Contrast:

Icon has a swap operator: `x :=: y`

Parallel assignment, continued

Speculate: What does the following do?

```
>> a,b,c = [10,20,30,40,50]

>> [a,b,c]
=> [10, 20, 30]
```

Speculate again:

```
>> a,b,*c = [10,20,30,40,50]

>> [a,b,c]
=> [10, 20, [30, 40, 50]]

>> a,*b,*c = [10,20,30,40,50]
SyntaxError: (irb):57: syntax error, unexpected *
```

Section 4.5.5 in RPL has full details on parallel assignment. It is both more complicated and less general than pattern matching in Haskell. (!)

Control Structures

The **while** loop

Here's a loop to print the integers from 1 through 10, one per line.

```
i=1
while i <= 10 do        # "do" is optional
   puts i
   i += 1
end
```

When **i <= 10** produces **false**, control branches to the code following **end**, if any.

The body of the **while** is always terminated with **end**, even if there's only one expression in the body.

while, continued

Java control structures such as **if**, **while**, and **for** are driven by the result of expressions that produce a value whose type is **boolean**.

C has a more flexible view: control structures consider a scalar value that is non-zero to be "true".

PHP considers zeroes, the empty string, the string "0", empty arrays, and more to be false.

Python and JavaScript, too, have sets of "truthy" and "falsy/falsey" values.

Here's the Ruby rule:

Any value that is not **false** or **nil** is considered to be "true".

while, continued

Remember: Any value that is not **false** or **nil** is considered to be "true".

Let's analyze this loop, which reads lines from standard input using **gets**.

```
while line = gets
    puts line
end
```

gets returns a string that is the next line of the input, or **nil**, on end of file.

The expression **line = gets** has two side effects but also produces a value.
Side effects: (1) a line is read from standard input and (2) is assigned to **line**.
Value: The string assigned to **line**.

If the first line of the file is "**one**", then the first time through the loop what's evaluated is **while "one"**.

The value "**one**" is not **false** or **nil**, so the body of the loop is executed, causing "**one**" to be printed on standard output.

At end of file, **gets** returns **nil**. **nil** is assigned to **line** and produced as the value of the assignment, in turn terminating the loop.

LHtLaL sidebar: Partial vs. full understanding

From the previous slide:

```
while line = gets
    puts line
end
```

Partial understanding:

That loop reads and prints every line from standard input.

Full understanding:

What we worked through on the previous slide.

I think there's merit in full understanding.

Another example of full understanding:

Knowing the full set of truthy/falsy rules for a language.

while, continued

String's **chomp** method removes a carriage return and/or newline from the end of a string, if present.

Here's a program that's intended to flatten all input lines to a single line:

```
result = ""
while line = gets.chomp
    result += line
end
puts result
```

It doesn't work. What's wrong with it?

Here's the error:

```
% ruby while4.rb < lines.txt
while4.rb:2:in `<main>': undefined method `chomp' for
nil:NilClass (NoMethodError)
```

while, continued

At hand:

```
result = ""
while line = gets.chomp
    result += line
end
puts result
```

At end of file, **gets** returns **nil**, producing an error on **gets.chomp**.

Which of the two alternatives below is better? What's a third alternative?

```
result = ""
while line = gets
   line.chomp!
   result += line
end
puts result
```

```
result = ""
while line = gets
   result += line.chomp
end
puts result
```

while, continued

Problem: Write a **while** loop that prints the characters in the string **s**, one per line. Don't use the **length** or **size** methods of **String**.

Extra credit: Don't use any variables other than **s**.

Solution: (**while5.rb**)

```
i = 0
while c = s[i]
   puts c
   i += 1
end
```

Solution with only **s**: (**while5a.rb**)

```
while s[0]
   puts s[0]
   s[0] = ""
end
```

Source code layout

Unlike Java, Ruby does pay some attention to the presence of newlines in source code.

For example, a while loop cannot be trivially squashed onto a single line.

```
while i <= 10 puts i i += 1 end          # Syntax error
```

If we add semicolons where newlines originally were, it works:

```
while i <= 10; puts i; i += 1; end       # OK
```

There is some middle ground, too:

```
while i <= 10 do puts i; i+=1 end        # OK.  Note added "do"
```

Unlike Haskell and Python, indentation is never significant in Ruby.

Source code layout, continued

Ruby considers a newline to terminate an expression, unless the expression is definitely incomplete.

For example, the following is ok because "**i <=**" is definitely incomplete.

```
while i <=
10 do puts i; i += 1 end
```

Is the following ok?

```
while i
<= 10 do puts i; i += 1 end
```

Nope...

```
syntax error, unexpected tLEQ
<= 10 do puts i; i += 1 end
^
```

Source code layout, continued

Can you think of any pitfalls that the incomplete expression rule could produce?

Example of a pitfall: Ruby considers

```
x = a + b
    + c
```

to be two expressions: **`x = a + b`** and **`+ c`**.

Rule of thumb: If breaking an expression across lines, end lines with an operator:

```
x = a + b +
    c
```

Alternative: Indicate continuation with a backslash at the end of the line.

Expression or statement?

Academic writing on programming languages commonly uses the term "statement" to denote a syntactic element that performs operation(s) but does not produce a value.

The term "expression" is consistently used to describe a construct that produces a value.

Ruby literature sometimes talks about the "while statement" even though **while** produces a value:

```
>> i = 1
>> while i <= 3 do i += 1 end
=> nil
```

Dilemma: Do we call it the "while statement" or the "while expression"?

We'll see later that the **break** construct can cause a **while** loop to produce a value other than **nil**.

Logical operators

Ruby has operators for conjunction, disjunction, and "not" with the same symbols as Java and C, but with somewhat different semantics.

Conjunction is **&&**, just like Java, but note the values produced:

```
>> true && false
=> false

>> 1 && 2
=> 2

>> true && "abc"
=> "abc"

>> nil && 1
=> nil
```

Remember:
Any value that is not **false** or **nil** is considered to be "true".

Challenge: Concisely describe the rule that Ruby uses to determine the value of a conjunction operation.

Logical operators, continued

Disjunction is ||, also like Java. As with conjunction, the values produced are interesting:

```
>> 1 || nil
=> 1

>> false || 2
=> 2

>> "abc" || "xyz"
=> "abc"

>> s = "abc"
>> s[0] || s[3]
=> "a"

>> s[4] || false
=> false
```

Remember:
Any value that is not **false** or **nil** is considered to be "true".

Logical operators, continued

An exclamation mark inverts a logical value. The resulting value is always **true** or **false**.

```
>> ! true
=> false

>> ! 1
=> false

>> ! nil
=> true

>> ! (1 || 2)
=> false

>> ! ("abc"[5] || [1,2,3][10])
=> true

>> ![nil]
=> false
```

Remember:

Any value that is not **false** or **nil** is considered to be "true".

Logical operators, continued

There are also **and**, **or**, and **not** operators, but with very low precedence.

Why?

They eliminate the need for parentheses in some cases.

We can write this,

```
x < 2 && y > 3 or x * y < 10 || z > 20
```

instead of this:

```
(x < 2 && y > 3) || (x * y < 10 || z > 20)
```

LHtLaL problem: Devise an example for ! vs. **not**.

if-then-else

Here is Ruby's **if-then-else**:

```
>> if 1 < 2 then "three" else [4] end
=> "three"

>> if 10 < 2 then "three" else [4] end
=> [4]

>> if 0 then "three" else [4] end * 3
=> "threethreethree"
```

Observations?

Speculate: Is the following valid? If so, what will it produce?

```
if 1 > 2 then 3 end
```

if-then-else, continued

If a language's **if-then-else** returns a value, it creates an issue about the meaning of an **if-then** with no **else**.

In Ruby, if there's no **else** clause and the control expression is **false**, **nil** is produced:

```
>> if 1 > 2 then 3 end
=> nil
```

In the C family, **if-else** doesn't return a value.

Haskell and ML simply don't allow an **else**-less **if**.

In Icon, an expression like **if 2 > 3 then 4** is said to *fail*. No value is produced, and failure propagates to any enclosing expression, which in turn fails.

Ruby also provides **1 > 2 ? 3 : 4**, a ternary conditional operator, just like the C family. Is that a good thing or bad thing? (TMTOWTDI!)

if-then-else, continued

The most common Ruby coding style puts the **if**, the **else**, the **end**, and the expressions of the clauses on separate lines:

```
if lower <= x && x <= higher or inExRange(x, rangeList) then
    puts "x is in range"
    history.add x
else
    outliers.add x
end
```

Note the use of the low-precedence **or** instead of `||`.

The trailing **then** above is optional.

then is not optional in this one-line expression:

```
if 1 then 2 else 3 end
```

The **elsif** clause

Ruby provides an **elsif** clause for "else-if" situations.

```
if average >= 90 then
   grade = "A"
elsif average >= 80 then
   grade = "B"
elsif average >= 70 then
   grade = "C"
else
   grade = "F"
end
```

Note that there is no "**end**" to terminate the **then** clauses. **elsif** both closes the current **then** and starts a new clause.

It is not required to have a final **else**.

Is **elsif** syntactic sugar?

elsif, continued

At hand:

```
if average >= 90 then
   grade = "A"
elsif average >= 80 then
   grade = "B"
elsif average >= 70 then
   grade = "C"
else
   grade = "F"
end
```

```
grade =
   if average >= 90 then "A"
   elsif average >= 80 then "B"
   elsif average >= 70 then "C"
   else "F"
   end
```

Can we shorten it by thinking less imperatively and more about values?

See 5.1.4 in RPL for Ruby's **case** (a.k.a. "switch") expression.

if and **unless** as *modifiers*

if and **unless** can be used as *modifiers* to indicate conditional execution.

```
>> total, count = 123.4, 5        # Note: parallel assignment

>> printf("average = %g\n", total / count) if count != 0
average = 24.68
=> nil

>> total, count = 123.4, 0
>> printf("average = %g\n", total / count) unless count == 0
=> nil
```

The general forms are:

expr1 if *expr2*
expr1 unless *expr2*

What does '**x.f if x**' mean?

break and **next**

Ruby's **break** and **next** are similar to Java's **break** and **continue**.

Below is a loop that reads lines from standard input, terminating on end of file or when a line beginning with a period is read. Each line is printed unless the line begins with a pound sign.

```
while line = gets
   if line[0] == "." then
      break
   end
   if line[0] == "#" then
      next
   end
   puts line
end
```

```
while line = gets
   break if line[0] == "."
   next if line[0] == "#"
   puts line
end
```

Problem: Rewrite the above loop to use **if** as a modifier.

break and **next**, continued

Remember that **while** is an expression that by default produces the value **nil** when the loop terminates.

If a while loop is exited with **break** *expr*, the value of **expr** is the value of the **while**.

Here's a contrived example to show the mechanics of it:

```
% cat break2.rb
s = "x"
puts (while true do
          break s if s.size > 30
          s += s
        end)

% ruby break2.rb
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The **for** loop

Here are three examples of Ruby's **for** loop:

```
for i in 1..100 do            # as with while, the do is optional
   sum += i
end

for i in [10,20,30]
   sum += i
end

for msymbol in "x".methods
   puts msymbol if msymbol.to_s.include? "!"
end
```

The "in" expression must be an object that has an **each** method.

In the first case, the "in" expression is a **Range**. In the latter two it is an **Array**.

The **for** loop, continued

The **for** loop supports parallel assignment:

```
for s,n,sep in [["1",5,"-"], ["s",2,"o"], [" <-> ",10,""]]
   puts [s] * n * sep
end
```

Output:

```
1-1-1-1-1
sos
 <->  <->  <->  <->  <->  <->  <->  <->  <->  <-> 
```

Consider the feature of supporting parallel assignment in the **for**.

- How would we write the above without it?
- What's the mental footprint of this feature?
- What's the big deal since there's already parallel assignment?
- Is this creeping featurism?

Methods and more

Method definition

Here is a simple Ruby method:

```
def add x, y
  return x + y
end
```

The keyword **def** indicates that this is a method definition.

Next is the method name.

The parameter list follows, optionally enclosed in parentheses.
    No types can be specified.

Zero or more expressions follow

end terminates the definition.

Method definition, continued

If the end of a method is reached without encountering a **`return`**, the value of the last expression becomes the return value.

Here is a more idiomatic definition for **`add`**:

```
def add x, y
  x + y
end
```

Method definition, continued

As we saw in an early example, if no arguments are required, the parameter list can be omitted:

```
def hello
    puts "Hello!"
end
```

What does **hello** return?

What does the last expression in **hello** return?

Testing methods with **irb**

One way to test methods in a file is to use **load**, a Kernel method.

```
% cat simple.rb
def add x, y
   x + y
end

def hello
   puts "Hello!"
end
```

```
>> load "simple.rb"
=> true
>> add 3,4
=> 7
>> hello
Hello!
[...edit simple.rb in another window...]
>> load "simple.rb"
=> true
>> hello
Hello! (v2)
```

How does **load** in Ruby differ from **:load** in **ghci**?

> **load "simple.rb"** is simply a Ruby expression that's evaluated by **irb**. Its side-effect is that the specified file is loaded.

Where's the class?!

I claim to be defining methods **add** and **hello** but there's no class in sight!

Methods can be added to a class at run-time in Ruby!

A freestanding method found in a file is associated with an object referred to as "**main**", an instance of **Object**.

At the top level, the name **self** references that object.

```
>> [self.class, self.to_s]        => [Object, "main"]

>> methods_b4 = self.private_methods
>> load "simple.rb"

>> self.private_methods - methods_b4
=> [:add, :hello]
```

We see that loading **simple.rb** added two methods to **main**.

Where's the class, continued?

We'll later see how to define classes but our initial "mode" on the Ruby assignments will be writing programs in terms of top-level methods.

This is essentially procedural programming with an object-oriented library.

Default values for arguments

Ruby allows default values to be specified for a method's arguments:

```
def wrap s, wrapper = "()"             # wrap3.rb
   wrapper[0] + s + wrapper[-1]        # Why -1?
end

>> wrap "abc", "<>"
=> "<abc>"

>> wrap "abc"
=> "(abc)"

>> wrap it, "|"
=> "|(abc)|"
```

Lots of library methods use default arguments.

```
>> "a-b c-d".split          => ["a-b", "c-d"]
>> "a-b c-d".split "-"      => ["a", "b c", "d"]
```

Methods can't be overloaded!

Ruby does not allow the methods of a class to be overloaded. Here's a Java-like approach that does not work:

```
def wrap s
   wrap(s, "()")
end

def wrap s, wrapper
   wrapper[0] + s + wrapper[-1]
end
```

The imagined behavior is that if **wrap** is called with one argument it will call the two-argument **wrap** with "()" as a second argument. In fact, the second definition of **wrap** simply replaces the first. (Last **def** wins!)

```
>> wrap "x"
ArgumentError: wrong number of arguments (1 for 2)

>> wrap("testing", "[ ]")        => "[testing]"
```

Sidebar: A study in contrast

Different languages approach overloading and default arguments in various ways. Here's a sampling:

Java	Overloading; no default arguments
Ruby	No overloading; default arguments
C++	Overloading and default arguments
Icon	No overloading; no default arguments; use an idiom

How does the mental footprint of the four approaches vary? What's the impact on the language's written specification?

Here is **wrap** in Icon:

```
procedure wrap(s, wrapper)
   /wrapper := "()" # if wrapper is &null, assign "()" to wrapper
   return wrapper[1] || s || wrapper[-1]
end
```

Arbitrary number of arguments

Java's **String.format** and C's **printf** can accept any number of arguments.

This Ruby method accepts any number of arguments and prints them:

```
def showargs(*args)
   puts "#{args.size} arguments"
   for i in 0...args.size do       # Recall a...b is a to b-1
      puts "##{i}: #{args[i]}"
   end
end
```

The rule: If a parameter is prefixed with an asterisk, an array is made of all following arguments.

```
>> showargs(1, "two", 3.0)
3 arguments:
#0: 1
#1: two
#2: 3.0
```

Arbitrary number of arguments, continued

Problem: Write a method **format** that interpolates argument values into a string where percent signs are found.

```
>> format("x = %, y = %, z = %\n", 7, "ten", "zoo")
=> "x = 7, y = ten, z = zoo\n"

>> format "testing\n"
=> "testing\n"
```

Use **to_s** for conversion to **String.**

A common term for this sort of facility is "varargs"—variable number of arguments.

```
def format(fmt, *args)
   result = ""
   for i in 0...fmt.size do
      if fmt[i] == "%" then
         result += args.shift.to_s
      else
         result += fmt[i]
      end
   end
   result
end
```

Source File Layout

Here's an example of source file layout for a program with several methods:

```
def main
   puts "in main"; f; g
end

def f; puts "in f" end
def g; puts "in g" end

main  # This runs the program
```

Execution:

```
% ruby main1.rb
in main
in f
in g
```

A rule: the definition for a method must be seen before it is <u>executed</u>.

The definitions for **f** and **g** can follow the definition of **main** because they aren't executed until **main** is executed.

Could the line "**main**" appear before the definition of **f**?

Try shuffling the three definitions and "**main**" to see what works and what doesn't.

Testing methods when there's a "main"

I'd like to load the following file and then test **showline**, but **load**ing it in **irb** seems to hang. Why?

```
% cat main3.rb
def showline s
   puts "Line: #{s.inspect} (#{s.size} chars)"
end
def main
   while line = gets; showline line; end
end
main

% irb
>> load "main3.rb"
```

...no output or >> prompt after the load...

Actually, it's waiting for input! After the **def**s for **showline** and **main**, **main** is called. **main** does a **gets**, and that **gets** is waiting for input.

Testing methods when there's a "main", cont.

Here's a technique that lets the program run normally with **ruby** but not run **main** when loaded with **irb**:

```
% cat main3a.rb
def showline s
   puts "Line: #{s.inspect} (#{s.size} chars)"
end
def main
   while line = gets; showline line; end
end
main unless $0 == "irb"

% irb
>> load "main3a.rb"
>> showline "testing"
Line: "testing" (7 chars)
>> main
(waits for input)
```

Call **main** unless the name of the program being run is "**irb**".

Now I can test methods by hand in **irb** but still do **ruby main3.rb ...**

Scoping rules for variables

Ordinary variables are local to the method in which they're created.

Example: (global0.rb)

```
def f
   puts "f: x = #{x}"       # undefined local variable or method `x'
end

def g
   x = 100    # This x is visible only in g
end

x = 10        # This x is visible only at the top-level in this file.

g

puts "top-level: x = #{x}"
```

Global variables

Variables prefixed with a $ are global, and can be referenced in any method in any file, including top-level code.

```
def f
   puts "f: $x = #{$x}"
end

def g
   $x = 100
end

$x = 10
f
g

puts "top-level: $x = #{$x}"
```

The code at left...

1. Sets **$x** at the top-level.
2. Prints **$x** in **f**.
3. Changes **$x** in **g**.
4. Prints the final value of **$x** at the top-level.

Output:

```
f: $x = 10
top-level: $x = 100
```

Constants

A rule in Ruby is that if an identifier begins with a capital letter, it represents a *constant*.

The first assignment to a constant is considered initialization.

```
>> MAX_ITEMS = 100
```

Assigning to an already initialized constant is permitted but a warning is generated.

```
>> MAX_ITEMS = 200
(irb):4: warning: already initialized constant MAX_ITEMS
=> 200
```

Modifying an object referenced by a constant does not produce a warning:

```
>> L = [10,20]
=> [10, 20]

>> L.push 30
=> [10, 20, 30]
```

Constants, continued

Pitfall: If a method is given a name that begins with a capital letter, it compiles ok but it can't be run!

```
>> def Hello; puts "hello!" end

>> Hello
NameError: uninitialized constant Hello
```

Constants, continued

There are a number of predefined constants. Here are a few:

`RUBY_VERSION`
The version of Ruby that's running.

`ARGV`
An array holding the command line arguments, like the argument to `main` in a Java program.

`ENV`
An object holding the "environment variables" (shown with `env` on UNIX machines and `set` on Windows machines.)

`STDIN`, `STDOUT`
Instances of the `IO` class representing standard input and standard output (the keyboard and screen, by default).

Duck Typing

Duck typing

Definition from Wikipedia (c.2015):

> *Duck typing* is a style of typing in which an object's methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of an explicit interface.

Recall these examples of the **for** loop:

```
for i in 1..100 do ...end

for i in [10,20,30] do ... end
```

for only requires that the "**in**" value be an object that has an **each** method. (It doesn't need to be a subclass of **Enumerable**, for example.)

This is an example of *duck typing*, so named based on the "duck test":

> *If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.*

For the case at hand, the value produced by the "**in**" expression qualifies as a "duck" if it has an **each** method.

Duck typing, continued

For reference:

> *Duck typing* is a style of typing in which an object's methods and properties determine the valid semantics, rather than its inheritance from a particular class or implementation of an explicit interface.
>
> —Wikipedia (c.2015)

Duck typing is both a technique and a mindset.

Ruby both facilitates and uses duck typing.

We don't say Ruby is duck typed. We say that Ruby allows duck typing.

Duck typing, continued

The key characteristic of duck typing is that we only care about whether an object supports the operation(s) we require.

With Ruby's **for** loop, it is only required that the **in** value have an **each** method.

Consider this method:

```
def double x
    x * 2
end
```

Remember: **x * 2** actually means **x.*(2)** — invoke the method ***** on the object **x** and pass it the value **2** as a parameter.

What operation(s) does **double** require that **x** support?

Duck typing, continued

```
def double x
    x * 2
end
```

```
>> double 10
=> 20

>> double "abc"
=> "abcabc"

>> double [1,2,3]
=> [1, 2, 3, 1, 2, 3]

>> double Rational(3)
=> (6/1)

>> double 1..10
NoMethodError: undefined method `*' for 1..10:Range
```

Is it good or bad that **double** operates on so many different types?

Is **double** polymorphic? What's the type of **double**?

Should we limit **double** to certain types, like numbers, strings and lists?

Duck typing, continued

Recall: The key characteristic of duck typing is that we only care about whether an object supports the operation(s) we require.

Should we have **double** check for known types?

```
def double x
   if [Fixnum, Float, String, Array].include? x.class
      x * 2
   else raise "Can't double a #{x.class}!" end
end
```

```
>> double "abc"
=> "abcabc"

>> double 1..2
RuntimeError: Can't double a Range!

>> double Rational(3)
RuntimeError: Can't double a Rational!
```

Previously...

```
>> double 1..10
NoMethodError: undefined
method `*' for 1..10:Range
```

Duck typing, continued

Here's **wrap** from slide 125. What does it require of **s** and **wrapper**?

```
def wrap s, wrapper = "()"
   wrapper[0] + s + wrapper[-1]
end

>> wrap "test", "<>"
=> "<test>"
```

Will the following work?

```
>> wrap "test", ["<<<",">>>"]
=> "<<<test>>>"

>> wrap [1,2,3], [["..."]]
=> ["...", 1, 2, 3, "..."]

>> wrap 10,3
=> 11
```

Duck typing, continued

Recall: <u>The key characteristic of duck typing is that we only care about whether an object supports the operation(s) we require.</u>

Does the following Java method exemplify duck typing?

```
static double sumOfAreas(Shape shapes[]) {
    double area = 0.0;
    for (Shape s: shapes)
        area += s.getArea();
    return area;
    }
```

No! **sumOfAreas** requires an array of **Shape** instances.

Could we change **Shape** to **Object** above? Would that be duck typing?

Does duck typing require a language to be dynamically typed?

Iterators and blocks

Iterators and blocks

Some methods are *iterators*. One of the many iterators in the **Array** class is **each**.

each iterates over the elements of the array. Example:

```
>> x = [10,20,30]

>> x.each { puts "element" }
element
element
element
=> [10, 20, 30]   # (each returns its receiver but it's often not used)
```

An iterator is a method that can invoke a block.

The construct **{ puts "element" }** is a *block*.

Array#each invokes the block once for each element of the array.

Because there are three values in **x**, the block is invoked three times, printing "**element**" each time.

Iterators and blocks, continued

Recall: An iterator is a method that can invoke a block.

Iterators can pass one or more values to a block as arguments.

A block can access arguments by naming them with a parameter list, a comma-separated sequence of identifiers enclosed in vertical bars.

```
>> [10, "twenty", [30,40]].each { |e| puts "element: #{e}" }
element: 10
element: twenty
element: [30, 40]
=> [10, "twenty", [30, 40]]
```

The behavior of the iterator **Array#each** is to invoke the block with each array element in turn.

Iterators and blocks, continued

For reference:

```
[10, "twenty", [30,40]].each { |e| puts "element: #{e}" }
```

Problem: Using a block, compute the sum of the numbers in an array containing values of any type. (Use **e.is_a? Numeric** to decide whether **e** is a number of some sort.)

```
>> sum = 0
>> [10, "twenty", 30].each {  ??? }

>> sum          => 40
```

Note: **sum = ...** inside the block changes it outside the block. (Rules coming soon!)

```
>> sum = 0
>> (1..100).to_a.each { |e| sum += e if e.is_a? Numeric  }
>> sum          => 5050
```

Sidebar: Iterate with **each** or use a **for** loop?

Recall that the **for** loop requires the value of the "**in**" expression to have an **each** method.

That leads to a choice between a **for** loop,

```
for name in "x".methods do
    puts name if name.to_s.include? "!"
end
```

and iteration with **each**,

```
"x".methods.each {|name| puts name if name.to_s.include? "!" }
```

Which is better?

Iterators and blocks, continued

Array#each is typically used to create side effects of interest, like printing values or changing variables.

In contrast, with some iterators it is the value returned by an iterator that is of principle interest.

See if you can describe what the following iterators are doing.

```
>> [10, "twenty", 30].collect { |v| v * 2 }
=> [20, "twentytwenty", 60]

>> [[1,2], "a", [3], "four"].select { |v| v.size == 1 }
=> ["a", [3]]
```

What do those remind you of?

Iterators and blocks, continued

The block for **Array#sort** takes two arguments.

```
>> [30, 20, 10, 40].sort { |a,b| a <=> b}
=> [10, 20, 30, 40]
```

Speculate: what are the arguments being passed to **sort**'s block? How could we find out?

```
>> [30, 20, 10, 40].sort { |a,b| puts "call: #{a} #{b}"; a <=> b}
call: 30 10
call: 10 40
call: 30 40
call: 20 30
call: 10 20
=> [10, 20, 30, 40]
```

How could we reverse the order of the **sort**?

Iterators and blocks, continued

Problem: sort the words in a sentence by descending length.

```
>> "a longer try first".split.sort { |a,b| b.size <=> a.size  }
=> ["longer", "first", "try", "a"]
```

What do the following examples remind you of?

```
>> [10, 20, 30].inject(0) { |sum, i| sum + i }
=> 60

>> [10,20,30].inject([]) {
        |memo, element| memo << element << "---" }
=> [10, "---", 20, "---", 30, "---"]
```

Iterators in Enumerable

We can query the "ancestors" of a class like this:

```
>> Array.ancestors
=> [Array, Enumerable, Object, Kernel, BasicObject]
```

For now we'll simply say that an object can call methods in its ancestors.

`Enumerable` has a number of iterators. Here are some:

```
>> [2,4,5].any? { |n| n.odd? }
=> true

>> [2,4,5].all? { |n| n.odd? }
=> false

>> [1,10,17,25].find { |n| n % 5 == 0 }
=> 10
```

Iterators in **Enumerable**

At hand:

A object can call methods in its **ancestors**. An ancestor of **Array** is **Enumerable**.

Another **Enumerable** method is **max**:

```
>> ["apple", "banana", "grape"].max {
                    |a,b| v = "aeiou"
                    a.count(v) <=> b.count(v)
                    }
=> "banana"
```

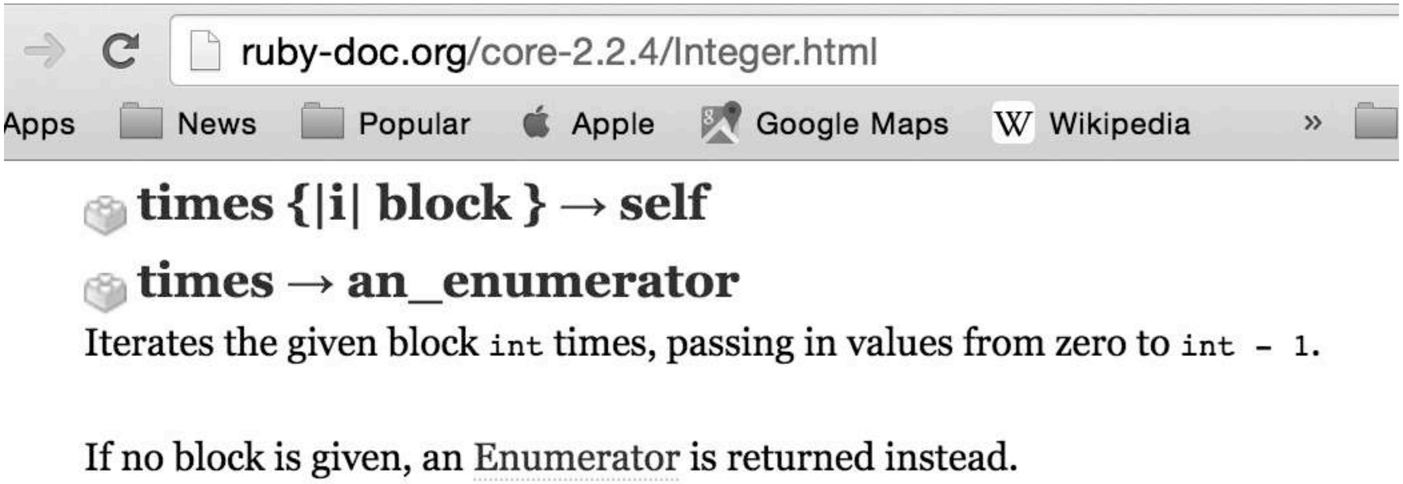
The methods in **Enumerable** use duck typing. They require only an **each** method except for **min**, **max**, and **sort,** which also require **<=>**.

See **http://ruby-doc.org/core-2.2.4/Enumerable.html**

Iterators abound!

Recall: An iterator is a method that can invoke a block.

Many classes have one or more iterators. One way to find them is to search their **ruby-doc.org** page for "block".



times {|i| block } → self

times → an_enumerator

Iterates the given block `int` times, passing in values from zero to `int - 1`.

If no block is given, an Enumerator is returned instead.

What will **3.times { |n| puts n }** do?

```
>> 3.times { |n| puts n }
0
1
2
=> 3
```

A few more iterators

Three more examples:

```
>> "abc".each { |c| puts c }
NoMethodError: undefined method `each' for "abc":String

>> "abc".each_char { |c| puts c }
a
b
c
=> "abc"

>> i = 0
>> "Mississippi".gsub("i") { (i += 1).to_s }
=> "M1ss2ss3pp4"
```

The "do" syntax for blocks

An alternative to enclosing a block in braces is to use **do**/**end**:

```
a.each do
   |element|
   print "element: #{element}\n"
end
```

Common style is to use brackets for one-line blocks, like previous examples, and **do**...**end** for multi-line blocks.

The opening brace or **do** for a block must be on the same line as the iterator invocation. Here's an error:

```
a.each
   do    # syntax error, unexpected keyword_do_block,
         #   expecting $end
   |element|
   print "element: #{element}\n"
end
```

Nested blocks

sumnums.rb reads lines from standard input, assumes the lines consist of integers separated by spaces, and prints their total, count, and average.

```
total = n = 0
readlines().each do
   |line|
   line.split(" ").each do
      |word|
      total += word.to_i
      n += 1
   end
end
printf("total = %d, n = %d, average = %g\n",
     total, n, total / n.to_f) if n != 0
```

```
% cat nums.dat
5 10 0 50

 200
1 2 3 4 5 6 7 8 9 10
% ruby sumnums.rb < nums.dat
total = 320, n = 15, average = 21.3333
```

Kernel#readlines reads/returns all of standard input as an array of lines.

The **printf** format specifier **%g** indicates to format a floating point number and select the better of fixed point or exponential form based on the value.

Scoping issues with blocks

Blocks raise issues with the scope of variables.

If a variable exists outside of a block, references to that variable in a block refer to that existing variable. Example:

```
>> sum = 0     Note: sum will accumulate across two iterator calls

>> [10,20,30].each {|x| sum += x}

>> sum
=> 60

>> [10,20,30].each {|x| sum += x}

>> sum
=> 120
```

Scoping issues with blocks, continued

If a variable is created in a block, the scope of the variable is limited to the block.

In the example below we confirm that **x** exists only in the block, and that the block's parameter, **e**, is local to the block.

```
>> e = "eee"
>> x
NameError: undefined local variable or method `x' ...

>> [10,20,30].each {|e| x = e * 2; puts x}
20
...
>> x
NameError: undefined local variable or method `x' ...
>> e
=> "eee"        # e's value was not changed by the block
```

LHtLaL

Scoping issues with blocks, continued

Pitfall: If we write a block that references a currently unused variable but later add a use for that variable outside the block, we might get a surprise.

Version 1:

```
a.each do |x|
    result = ... # first use of result in this method
    ...
end
```

Version 2:

```
result = ...       # new first use of result in this method
...
a.each do |x|
    ...
    result = ... # references/clobbers result in outer scope
end
...
...use result...   # uses value of result set in block.  Surprise!
```

Scoping issues with blocks, continued

We can make variable(s) local to a block by adding them at the end of the block's parameter list, preceded by a semicolon.

```
result = ...
...
a.each do
   |x; result, tmp|
     result = ... # result is local to block
     ...
end

...
...use result...   # uses result created outside of block
```

Writing iterators

A simple iterator

Recall: An iterator is a method that can invoke a block.

The **yield** expression invokes the block associated with the current method invocation. Arguments of **yield** become parameters of the block.

Here is a simple iterator that yields two values, a 3 and a 7:

```
def simple
    puts "simple: Starting..."
    yield 3
    puts "simple: Continuing..."
    yield 7
    puts "simple: Done..."
    "simple result"
end
```

Usage:

```
>> simple  {|x|puts "\tx = #{x}" }
simple: Starting...
        x = 3
simple: Continuing...
        x = 7
simple: Done...
=> "simple result"
```

The **puts** in **simple** are used to show when **simple** is active. Note the interleaving of execution between the iterator and the block.

A simple iterator, continued

At hand:

```
def simple
    puts "simple: Starting..."
    yield 3
    puts "simple: Continuing..."
    yield 7
    puts "simple: Done..."
    "simple result"
end
```

Usage:

```
>> simple { |x| puts "\tx = #{x}" }
simple: Starting...
        x = 3
simple: Continuing...
        x = 7
simple: Done...
=> "simple result"
```

There's no formal parameter that corresponds to a block. The block, if any, is implicitly referenced by **yield**.

The parameter of **yield** becomes the named parameter for the block.

Calling **simple** without a block produces an error on the first **yield**:

```
>> simple
simple: Starting...
LocalJumpError: no block given (yield)
```

Write **from_to**

Problem: Write an iterator **from_to(f, t, by)** that yields the integers from **f** through **t** in steps of **by**, which defaults to 1. Assume **f <= t**.

```
>> from_to(1,3) { |i| puts i }
1
2
3
=> 3

>> from_to(0,99,25) { |i| puts i }
0
25
50
75
=> 4
```

Parameters are passed to the iterator (the method) just like any other method.

from_to, continued

Solution:

```
def from_to(from, to, by = 1)
    n = from
    results = 0
    while n <= to do
        yield n
        n += by
        results += 1
    end
    results
end
```

Desired:

```
>> from_to(1,10,2) { |i| puts i }
1
3
5
7
9
=> 5
```

Another test:

```
>> from_to(-5,5,1) { |i| print i, " " }
-5 -4 -3 -2 -1 0 1 2 3 4 5 => 11
```

yield, continued

To pass multiple arguments for a block, specify multiple arguments for yield.

Imagine an iterator that produces overlapping pairs from an array:

```
>> elem_pairs([3,1,5,9]) { |x,y| print "x = #{x}, y = #{y}\n" }
x = 3, y = 1
x = 1, y = 5
x = 5, y = 9
```

Implementation:

```
def elem_pairs(a)
   for i in 0...(a.length-1)
      yield a[i], a[i+1]              # yield(a[i], a[i+1]) is ok, too
   end
end
```

Speculate: What will be the result with yield [a[i], a[i+1]]? (Extra [...])

A round-trip with **yield**

When **yield** passes a value to a block the result of the block becomes the value of the **yield** expression.

Here is a trivial iterator to show the mechanics:

```
def round_trip x
   r = yield x
   "yielded #{x} and got back #{r}"
end
```

Usage:

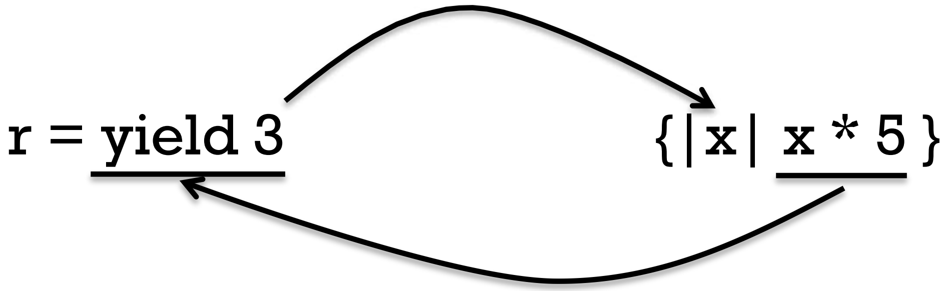
```
>> round_trip(3) {|x| x * 5 }      # parens around 3 are required!
=> "yielded 3 and got back 15"

>> round_trip("testing") {|x| x.size }
=> "yielded testing and got back 7"
```

A round-trip with **yield**, continued

At hand:

```
def round_trip x
   r = yield x
   "yielded #{x} and got back #{r}"
end

>> round_trip(3) {|x| x * 5 }
=> "yielded 3 and got back 15"
```

1. Iterator yields **3** to block. **x** becomes **3**.



r = yield 3   **{|x| x * 5 }**

2. Block returns **15**, which becomes value of **yield 3**.

3. Value of **yield** 3 is assigned to **r**.

Round trips with **yield**

Consider this iterator:

```
>> select([[1,2], "a", [3], "four"]) { |v| v.size == 1 }
=> ["a", [3]]

>> select("testing this here".split) { |w| w.include? "e" }
=> ["testing", "here"]
```

What does it appear to be doing?

Producing the elements in its argument, an array, for which the block produces true.

Problem: Write it!

Round trips with **yield**, continued

At hand:

```
>> select([[1,2], "a", [3], "four"]) { |v| v.size == 1 }
=> ["a", [3]]
```

Solution:

```
def select array
   result = [ ]

   for element in array
      if yield element then
         result << element
      end
   end

   result
end
```

What does the iterator/block interaction look like?

```
     Iterator                    Block
if yield [1,2] then  # [1,2].size == 1
   ~~result << [1,2]~~

if yield "a" then    # "a".size == 1
     result << "a"

if yield [3] then    # [3] .size == 1
     result << [3]

if yield "four" then # "four".size == 1
     ~~result << "four"~~
```

Round trips with **yield**, continued

```
def select array
   result = [ ]
   for element in array
      if yield element then
         result << element
      end
   end

   result
end
```

Is **select** limited to arrays?

```
>> select(1..10) {|n| n.odd? && n > 5 }
=> [7, 9]
```

Why does that work?

Because **for var in x** works for any **x** that has an **each** method. (Duck typing!)

What's a better name than **array** for **select**'s parameter?

Problem: Rewrite select to use the iterator **each** instead of a for loop. Also use an **if** modifier with the **yield**.

Round trips with **yield**, continued

Solution:

```
def select eachable
   result = []
   eachable.each do
      |element|
      result << element if yield element
   end
   result
end
```

```
def select array
   result = [ ]
   for element in array
      if yield element then
         result << element
      end
   end

   result
end
```

What's the difference between our **select**,

```
select([[1,2], "a", [3], "four"]) { |v| v.size == 1 }
```

And Ruby's **Array#select**?

```
[[1,2], "a", [3], "four"].select { |v| v.size == 1 }
```

Ruby's **Array#select** is a method of **Array**. Our **select** is added to the object "**main**". (See slide 123.)

Sidebar: Ruby vs. Haskell

```
def select array
  result = [ ]
  for element in array
    if yield element then
      result << element
    end
  end

  result
end

>> select(["just","a", "test"]) { |x| x.size == 4 }
=> ["just", "test"]
```

```
select _ [] = []
select f (x:xs)
   | f x = x : select f xs
   | otherwise = select f xs

> select (\x -> length x == 4) ["just","a", "test"]
["just","test"]
```

Which is better?

Various types of iteration side-by-side

```
>> [10, "twenty", [30,40]].each { |e| puts "element: #{e}" }
>> sum = 0; [1,2,3].each { |x| sum += x }
```

Invokes block with each element in turn for side-effect(s). Result of **each** uninteresting.

```
>> [10,20,30].map { |x| x * 2 }   => [20, 40, 60]
```

Invokes block with each element in turn and returns array of block results.

```
>> [2,4,5].all? { |n| n.odd? }       => false
```

Invokes block with each element in turn; each block result contributes to final result of **true** or **false**, possibly short-circuiting.

```
>> [[1,2], "a", [3], "four"].select { |v| v.size == 1 } => ["a", [3]]
```

Invokes block to determine membership in final result.

```
>> "try this first".split.sort {|a,b| b.size <=> a.size }  => [...]
```

Invokes block an arbitrary number of times; each block result guides further computation towards final result.

The **Hash** class

The **Hash** class

Ruby's **Hash** class is similar to the **Map** family in Java and dictionaries in Python. It's like an array that can be subscripted with values of any type.

The expression **{ }** (empty curly braces) creates a **Hash**:

```
>> numbers = {}        => {}

>> numbers.class      => Hash
```

Subscripting with a *key* and assigning a value stores that key/value pair.

```
>> numbers["one"] = 1

>> numbers["two"] = 2

>> numbers
=> {"one"=>1, "two"=>2}

>> numbers.size
=> 2
```

Hash, continued

At hand:

```
>> numbers
=> {"one"=>1, "two"=>2}
```

Subscripting with a key fetches the associated value. If the key is not found, **nil** is produced.

```
>> numbers["two"]
=> 2

>> numbers["three"]
=> nil
```

Hash, continued

At hand:

```
>> numbers   => {"one"=>1, "two"=>2}
```

The **Hash** class has many methods. Here's a sampling**:**

```
>> numbers.keys
=> ["one", "two"]

>> numbers.values
=> [1, 2]

>> numbers.invert
=> {1=>"one", 2=>"two"}

>> numbers.to_a
=> [["one", 1], ["two", 2]]
```

Some of the many **Hash** iterators: **delete_if**, **each_pair**, **select**

Hash, continued

At hand:

```
>> numbers
=> {"one"=>1, "two"=>2}
```

The value associated with a key can be changed via assignment.

```
>> numbers["two"] = "1 + 1"
```

A key/value pair can be removed with **Hash#delete**.

```
>> numbers.delete("one")
=> 1  # Returns associated value

>> numbers
=> {"two"=>"1 + 1"}

>> numbers["one"]
=> nil
```

Hash, continued

The rules for keys and values:

- Key values must have a **hash** method that produces a **Fixnum**. (Duck typing!)
- Any value can be the value in a key/value pair.

```
>> h = {};  a = [1,2,3]

>> h[a] = "-"

>> h[String] = ["a","b","c"]

>> h["x".class] * h[(1..3).to_a]
=> "a-b-c"

>> h[h] = h

>> h
=> {[1, 2, 3]=>"-", String=>["a", "b", "c"], {...}=>{...}}
```

Note that keys for a given **Hash** may be a mix of types. Ditto for values. (Unlike a Java **HashMap**.)

Hash, continued

Inconsistencies can arise when using mutable values as keys.

```
>> h = {}; a = []

>> h[a] = "x"

>> h
=> {[]=>"x"}

>> a << 1

>> h
=> {[1]=>"x"}

>> h[a]
=> nil
```

Ruby treats string-valued keys as a special case and makes a copy of them.

Hash, continued

Here's a sequence that shows some of the flexibility of hashes.

```
>> h = {}

>> h[1000] = [1,2]

>> h[true]  = {}

>> h[[1,2,3]] = [4]

>> h
=> {1000=>[1, 2], true=>{}, [1, 2, 3]=>[4]}

>> h[h[1000] + [3]] << 40

>> h[!h[10]]["x"] = "ten"

>> h
=> {1000=>[1, 2], true=>{"x"=>"ten"}, [1, 2, 3]=>[4, 40]}
```

Default values

An earlier simplification: If a key is not found, **nil** is returned.
Full detail: If a key is not found, the *default value* of the hash is returned.

The default value of a hash defaults to **nil** but an arbitrary default value can be specified when creating a hash with **new**:

```
>> h = Hash.new("Go Fish!")      # Example from ruby-doc.org

>> h.default
=> "Go Fish!"

>> h["x"] = [1,2]

>> h["x"]
=> [1, 2]

>> h["y"]
=> "Go Fish!"
```

tally.rb

Problem: write **tally.rb**, to tally occurrences of blank-separated "words" on standard input.

```
% ruby tally.rb
to be or
not to be
^D
{"to"=>2, "be"=>2, "or"=>1, "not"=>1}
```

How can we approach it?

tally.rb

Solution:

```
# Use default of zero so += 1 works
counts = Hash.new(0)

readlines.each do
    |line|
   line.split(" ").each do
      |word|
      counts[word] += 1
   end
end

# Like puts counts.inspect
p counts
```

```
% ruby tally.rb
to be or
not to be
^D
{"to"=>2, "be"=>2,
"or"=>1, "not"=>1}
```

Contrast with while/for vs. iterators:

```
counts = Hash.new(0)
while line = gets do
   for word in line.split(" ") do
      counts[word] += 1
   end
end
p counts
```

tally.rb, continued

The output of **tally.rb** is not customer-ready!

```
{"to"=>2, "be"=>2, "or"=>1, "not"=>1}
```

Hash#sort produces an array of key/value arrays ordered by the keys, in ascending order:

```
>> counts.sort
=> [["be", 2], ["not", 1], ["or", 1], ["to", 2]]
```

Problem: Produce nicely labeled output, like this:

```
Word    Count
be          2
not         1
or          1
to          2
```

tally.rb, continued

```
Word    Count
be          2
not         1
or          1
to          2
```

At hand:

```
>> counts.sort
[["be", 2], ["not", 1], ["or", 1], ["to", 2]]
```

Solution:

```
([["Word","Count"]] + counts.sort).each do
   |k,v| printf("%-7s %5s\n", k, v)
end
```

Notes:

- The minus in the format **%-7s** left-justifies, in a field of width seven.
- As a shortcut for easy alignment, the column headers are put at the start of the array, as a fake key/value pair.
- We use **%5s** instead of **%5d** to format the counts and accommodate "**Count**", too. (This works because **%s** causes **to_s** to be invoked on the value being formatted.)
- A next step might be to size columns based on content.

More on **Hash** sorting

Hash#sort's default behavior of ordering by keys can be overridden by supplying a block. The block is repeatedly invoked with two key/value pairs, like **["be", 2]** and **["or", 1]**.

Here's a block that sorts by descending count: (the second element of the two-element arrays)

```
>> counts.sort { |a,b| b[1] <=> a[1] }
=> [["to", 2], ["be", 2], ["or", 1], ["not", 1]]
```

How we could resolve ties on counts by alphabetic ordering of the words?

```
counts.sort do
    |a,b|
    r = b[1] <=> a[1]
    if r != 0 then r else a[0] <=> b[0] end
end
=> [["be", 2], ["to", 2], ["not", 1], ["or", 1]]
```

xref.rb

Let's turn **tally.rb** into a cross-reference program:

```
% cat xref.1
to be or
not to be is not
to be the question

% ruby xref.rb < xref.1
Word      Lines
be        1, 2, 3
is        2
not       2
or        1
question  3
the       3
to        1, 2, 3
```

```
counts = Hash.new(0)

readlines.each do
   |line|
  line.split(" ").each do
     |word|
     counts[word] += 1
  end
end
```

How can we approach it?

xref.rb, continued

Changes:

- Use **each_with_index** to get line numbers (0-based).
- Turn **counts** into **refs**, a **Hash** whose values are arrays.
- For each **word** on a line...
  - If **word** hasn't been seen, add a key/value pair with **word** and an empty array.
  - Add the current line number to **refs[word]**

Revised:

```
refs = {}
readlines.each_with_index do
    |line, num|
    line.split(" ").each do
        |word|
        refs[word] = [] unless refs.member? word
        refs[word] << num unless refs[word].member? num
    end
end
```

xref.rb, continued

If we add "**p refs**" after that loop, here's what we see:

```
% cat xref.1
to be or
not to be is not
to be the question

% ruby xref.rb < xref.1
{"to"=>[0, 1, 2], "be"=>[0, 1, 2], "or"=>[0], "not"=>[1],
"is"=>[1], "the"=>[2], "question"=>[2]}
```

We want:

```
% ruby xref.rb < xref.1
Word       Lines
be         1, 2, 3
is         2
not        2
...
```

xref.rb, continued

At hand:

```
{"to"=>[0, 1, 2], "be"=>[0, 1, 2], "or"=>[0], "not"=>[1], ...
```

We want:

```
Word        Lines
be          1, 2, 3
...
```

Let's get fancy and size the "`Word`" column based on the largest word:

```
max_len = refs.map {|k,v| k.size}.max
fmt = "%-#{max_len}s  %s\n"

print fmt % ["Word", "Lines"]
refs.sort.each do
   |k,v|
   printf(fmt, k, v.map {|n| n+1} * ", ")
end
```

Another **Hash** behavior

Observe:

```
>> h = Hash.new { |h,k| h[k] = [] }

>> h["to"]
=> []

>> h
=> {"to"=>[]}
```

If **Hash.new** is called with a block, that block is invoked when a nonexistent key is accessed.

The block is passed the **Hash** and the key.

What does the block above do when a key doesn't exist?

It adds a key/value pair that associates the key with a new, empty array.

Symbols

An identifier preceded by a colon creates a Symbol.

```
>> s1 = :testing
=> :testing

>> s1.class
=> Symbol
```

A symbol is much like a string but a given identifier always produces the same Symbol object.

```
>> s1.object_id             => 1103708
>> :testing.object_id       => 1103708
```

In contrast, two identical string literals produce two different String objects:

```
>> "testing".object_id    => 3673780
>> "testing".object_id    => 4598080
```

Symbols, continued

A symbol can also be made from a string with **to_sym**:

```
>> "testing".to_sym
=> :testing

>> "==".to_sym
=> :==
```

Recall that **.methods** returns an array of symbols:

```
>> "".methods.sort
=> [:!, :!=, :%, :*, :+, :<, :<<, :<=, :<=>, :==, :===, :=~, :>, :>=,
:__id__, :__send__, :ascii_only?, :b, :between?, :bytes,
:bytesize, :byteslice, :capitalize, :capitalize!, :casecmp, :center,
:chars, :chomp, :chomp!, :chop, :chop!, :chr, :class, :clear, ...
```

Symbols and hashes

Because symbols can be quickly compared, they're commonly used as hash keys.

```
moves = {}
moves[:up] = [0,1]
moves[:down] = [0,-1]

>> moves
=> {:up=>[0, 1], :down=>[0, -1]}

> moves["up".to_sym]
=> [0, 1]

>> moves["down"]
=> nil
```

Symbols and hashes, continued

Instead of a series of assignments we can use an initialization syntax:

```
>> moves = { :up => [0,1], :down => [0,-1] }
=> {:up=>[0, 1], :down=>[0, -1]}
```

There's even more syntactic sugar available:

```
>> moves = { up:[0,1], down:[0,-1] }
=> {:up=>[0, 1], :down=>[0, -1]}
```

Regular Expressions

A little theory

In computer science theory, a *language* is a set of strings. The set may be infinite.

The Chomsky hierarchy of languages looks like this:

Unrestricted languages	("Type 0")
Context-sensitive languages	("Type 1")
Context-free languages	("Type 2")
Regular languages	("Type 3")

Roughly speaking, natural languages are unrestricted languages that can only be specified by unrestricted grammars.

Programming languages are usually context-free languages—they can be specified with context-free grammars, which have restrictive rules.

- Every Java program is a string in the context-free language that is specified by the Java grammar.

A regular language is a very limited kind of context free language that can be described by a regular grammar.

- <u>A regular language can also be described by a regular expression</u>.

A little theory, continued

A regular expression is simply a string that may contain *metacharacters*—characters with special meaning.

Here is a simple regular expression:

```
a+
```

It specifies the regular language that consists of the strings {**a, aa, aaa, ...**}.

Here is another regular expression:

```
(ab)+c*
```

It describes the set of strings that start with **ab** repeated one or more times and followed by zero or more **c**'s.

Some strings in the language are **ab**, **ababc**, and **abababababcccccc**.

The regular expression **(north|south)(east|west)** describes a language with four strings: {**northeast, northwest, southeast, southwest**}.

Good news and bad news

Regular expressions have a sound theoretical basis and are also very practical.

UNIX tools such as the **`ed`** editor and the **`grep`** family introduced regular expressions to a wide audience.

Most current editors and IDEs support regular expressions in searches.

Many languages provide a library for working with regular expressions.

- Java provides the **`java.util.regex`** package.
- The command **`man regex`** shows the interface for POSIX regular expression routines, usable in C.

Some languages, Ruby included, have a regular expression <u>type</u>.

Good news and bad news, continued

Regular expressions as covered in a theory class are relatively simple.

Regular expressions as available in many languages and libraries have been extended far beyond their theoretical basis.

In languages like Ruby, regular expressions are truly a language within a language.

An edition of the "Pickaxe" book devoted four pages to its summary of regular expressions.

- Four more pages sufficed to cover integers, floating point numbers, strings, ranges, arrays, and hashes.

Entire books have been written on the subject of regular expressions.

A number of tools have been developed to help programmers create and maintain complex regular expressions.

Good news and bad news, continued

Here is a regular expression written by Mark Cranness and posted at RegExLib.com:

```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?=[\x01-\x7f])
[^"\\]|\\[\x01-\x7f])*"\x20*)*(? <angle><))?((?!\.)(?>\.?[a-zA-
Z\d!#$%&'*+\-/=?^_`{|}~]+)+|"((?=[\x01-\x7f])[^"\\]|\\[\x01-\
x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+[a-zA-Z]{2,}|\[(((?(?<!\[)
\.)(25[0-5]|2[0-4]\d|[01]?\d? \d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:
((?=[\x01-\x7f])[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```

It describes RFC 2822 email addresses.

My opinion: regular expressions are good for simple tasks but grammar-based parsers should be favored as complexity rises, especially when an underlying specification includes a grammar.

We'll cover a subset of Ruby's regular expression capabilities.

A simple regular expression in Ruby

One way to create a regular expression (RE) in Ruby is to use the *`/regexp/`* syntax, for regular expression literals.

```
>> re = /a.b.c/          => /a.b.c/

>> re.class              => Regexp
```

In a RE, a dot is a metacharacter (a character with special meaning) that will match any (one) character.

Letters, numbers, and some special characters simply match themselves.

The RE `/a.b.c/` matches strings that contain the five-character sequence `a<`*anychar*`>b<`*anychar*`>c`

Examples: "albacore", "barbecue", "drawback", and "iambic".

The match operator

The binary operator **=~** is called "match".

One operand must be a string and the other must be a regular expression. If the string contains a match for the RE, the position of the match is returned. **nil** is returned if there is no match.

```
>> "albacore" =~ /a.b.c/      => 0

>> "drawback" =~ /a.b.c/     => 2

>> "abc" =~ /a.b.c/            => nil

>> "abcdef" =~ /..f/
=> 3

>> "abcdef" =~ /.f./
=> nil

>> "abc" =~ /..../
=> nil
```

Regular expressions are "in deep" in Ruby

Language-wise, what's an implication of the following?

```
>> /x/.class   => Regexp
```

Ruby has syntactic support for regular expressions. We can say that regular expressions are *first-class values* in Ruby.

In general there are two levels of support for a type:

Syntactic support

Most languages have syntactic support for strings with "...".

Scala and ActionScript have syntactic support for XML.

In Icon, '**aeiou**' is a *character set*, not a string.

Library support

Java and Python have classes for working with REs.

C and Icon have function libraries for working with REs.

What are the tradeoffs between the two levels?

Example from Icon: **cset("aeiou")** vs. '**aeiou**'

Sidebar: rgrep.rb

The UNIX grep command reads standard input or files named as arguments and prints lines that contain a specified regular expression:

```
$ grep g.h.i < /usr/share/dict/words
lengthwise

$ grep l.m.n < /usr/share/dict/words | wc -l
   252     252    2825

$ grep  ....................... < /usr/share/dict/words
electroencephalograph's
```

Problem: Write a simple grep in Ruby that will handle the cases above.

Hint: #{...} interpolation works in /.../ (regular expression) literals.

rgrep.rb sidebar, continued

UNIX **grep**:

```
$ grep g.h.i < /usr/share/dict/words
```

Solution:

```
while line = STDIN.gets    # STDIN so "g.h.i" isn't opened for input
   puts line if line =~ /#{ARGV[0]}/
end
```

Usage:

```
$ ruby rgrep.rb g.h.i < /usr/share/dict/words
lengthwise

$ ruby rgrep.rb  .......................  < /usr/share/dict/words
electroencephalograph's
```

The match operator, continued

After a successful match we can use some cryptically named predefined global variables to access parts of the string:

$` Is the portion of the string that precedes the match. (That's a backquote—ASCII code 96.)

$& Is the portion of the string that was matched by the regular expression.

$' Is the portion of the string following the match.

Example:

```
>> "limit=300" =~ /=/      => 5
>> $`                      => "limit"  (left of the match)
>> $&                      => "="      (the match itself)
>> $'                      => "300"    (right of the match)
```

The match operator, continued

Here's a handy utility routine from the Pickaxe book:

```
def show_match(s, re)
   if s =~ re then
      "#{$`}<<#{$&}>>#{$'}"
   else
      "no match"
   end
end
```

Usage:

```
>> show_match("limit is 300",/is/)
=> "limit <<is>> 300"

>> %w{albacore drawback iambic}.
        each { |w| puts show_match(w, /a.b.c/) }
<<albac>>ore
dr<<awbac>>k
i<<ambic>>
```

LHtLaL

Great idea: Put it in your **.irbrc**! Call it "**sm**", to save some typing!

Character classes

[*characters*] is a *character class*—a RE that matches any one of the characters enclosed by the square brackets.

/[aeiou]/ matches a single lower-case vowel

```
>> show_match("testing", /[aeiou]/)
=> "t<<e>>sting"
```

A dash between two characters in a class specification creates a range based on the collating sequence. [0-9] matches a single digit.

```
>> show_match("Testing 1, 2, 3...", /[0-9]/)
=> "Testing <<1>>, 2, 3..."

>> show_match("Take five!", /[0-9]/)
=> "no match"
```

Character classes

[^*characters*] is a RE that matches any single character not in the class. (It matches the complement of the class.)

/[^0-9]/ matches a single character that is not a digit.

```
>> show_match("1,000", /[^0-9]/)
=> "1<<,>>000"
```

For any RE we can ask,

What is the shortest string the RE can match? What is the longest?

What is the shortest string that [A-Za-z345] can match? The longest?

One for both! [*anything*] always has a one-character match!

Character classes, continued

Describe what's matched by this regular expression:

```
/.[a-z][0-9][a-z]./
```

A five character string whose middle three characters are, in order, a lowercase letter, a digit, and a lowercase letter.

In the following, which portion of the string is matched, if any?

```
>> show_match("A1b33s4ax1", /.[a-z][0-9][a-z]./)
=> "A1b3<<3s4ax>>1"
```

Character classes, continued

String#gsub does global substitution with both plain old strings and regular expressions

```
>> "520-621-6613".gsub("-", "<DASH>")
=> "520<DASH>621<DASH>6613"

>> "520-621-6613".gsub(/[02468]/, "(e#)")
=> "5(e#)(e#)-(e#)(e#)1-(e#)(e#)13"
```

There's an imperative form of **gsub**, too.

Character classes, continued

Some frequently used character classes can be specified with *C*

\d Stands for [0-9]
\w Stands for [A-Za-z0-9_]
\s Whitespace—blank, tab, carriage return, newline, formfeed

The abbreviations \D, \W, and \S produce a complemented class.

Examples:

```
>> show_match("Call me at 555-1212", /\d\d\d-\d\d\d\d/)
=> "Call me at <<555-1212>>"

>> "fun double(n) = n * 2".gsub(/\w/,".")
=> "... ......(.) = . * ."

>> "BIOW 208, 14:00-15:15 TR".gsub(/\D/, "")
=> "20814001515"

>> "buzz93@tv-2000.com".gsub(/[\w-]/,"*")
=> "******@*******.***"
```

Backslashes suppress special meaning

Preceding an RE metacharacter with a backslash suppresses its meaning.

```
>> show_match("123.456", /.\../)
=> "12<<3.4>>56"

>> "5-3^2*2.0".gsub(/[\^.\-6]/, "_")
=> "5_3_2*2_0"

>> show_match("x = y[1] + z", /\[\d\]/)
=> "x = y<<[1]>> + z"
```

An old technique with regular expressions is to take advantage of the fact that metacharacters often aren't special when used out of context:

```
>> "5-3^2*2.0".gsub(/[-6^.]/, "_")
=> "5_3_2*2_0"
```

Alternatives

Alternatives can be specified with a vertical bar:

```
>> show_match("a green box", /red|green|blue/)
=> "a <<green>> box"

>> %w{you ate a pie}.select { |s| s =~ /ea|ou|ie/ }
=> ["you", "pie"]
```

Alternatives and grouping

Parentheses can be used for grouping. Consider this regular expression:

```
/(two|three) (apple|biscuit)s/
```

It corresponds to a regular language that is a set of four strings:

{two apples, three apples, two biscuits, three biscuits}

Usage:

```
>> "I ate two apples." =~ /(two|three) (apple|biscuit)s/
=> 6

>> "She ate three mice." =~ /(two|three) (apple|biscuit)s/
=> nil
```

Another:

```
>> %w{you ate a mouse}.select { |s| s =~ /.(ea|ou|ie)./ }
=> ["mouse"]
```

Simple app: looking for letter patterns

Imagine a program to look through a word list for a pattern of consonants and vowels specified on the command line, showing matches in bars.

```
% ruby convow.rb cvcvcvcvcvcvcvcvc < web2
c|hemicomineralogic|al
|hepatoperitonitis|
o|verimaginativenes|s
```

A capital letter means to match exactly that letter, in lowercase. `e` matches either consonant or vowel.

```
% ruby convow.rb vvvDvvv < web2
Chromat|ioideae|
Rhodobacter|ioideae|

% ruby convow.rb vvvCvvv < web2  | wc -l
24

% ruby convow.rb vvvevvv < web2  | wc -l
43
```

convow.rb

Here's a solution. We loop through the command line argument and build up a regular expression of character classes and literal characters, and then look for lines with a match.

```
re = ""
ARGV[0].each_char do |char|
   re += case char                    # An example of Ruby's case
          when "v" then "[aeiou]"
          when "c" then "[^aeiou]"
          when "e" then "[a-z]"
          else char.downcase
        end
end
puts re
re = /#{re}/          # Transform re from String to Regexp
STDIN.each do
   |line|
   puts [$`, $&, $'] * "|" if line.chomp =~ re
end
```

```
$ ruby convow.rb cvc
[^aeiou][aeiou][^aeiou]

$ ruby convow.rb cEEcc
[^aeiou]ee[^aeiou][^aeiou]
```

There are regular expression operators

A rule we've been using but haven't formally stated is this:

If $\mathtt{R}_1$ and $\mathtt{R}_2$ are regular expressions then $\mathtt{R}_1\mathtt{R}_2$ is a regular expression.
In other words, juxtaposition is the concatenation operation for REs.

There are also postfix operators on regular expressions.

If **R** is a regular expression, then...

R* matches zero or more occurrences of **R**

R+ matches one or more occurrences of **R**

R? matches zero or one occurrences of **R**

All have higher precedence than juxtaposition.

*****, **+**, and **?** are commonly called *quantifiers* but PA doesn't use that term.

The *, +, and ? quantifiers

At hand:

R* matches zero or more occurrences of R
R+ matches one or more occurrences of R
R? matches zero or one occurrences of R

What does the RE **ab*c+d** describe?

An '**a**' that is followed by zero or more '**b**'s that are followed by one or more '**c**'s and then a '**d**'.

```
>> show_match("acd", /ab*c+d/)
=> "<<acd>>"

>> show_match("abcccc", /ab*c+d/)
=> "no match"

>> show_match("abcabccccddd", /ab*c+d/)
=> "abc<<abccccd>>dd"
```

The *, +, and ? quantifiers, continued

At hand:

R* matches zero or more occurrences of R
R+ matches one or more occurrences of R
R? matches zero or one occurrences of R

What does the RE -?\d+ describe?

Integers with any number of digits

```
>> show_match("y is -27 initially", /-?\d+/)
=> "y is <<-27>> initially"

>> show_match("maybe --123.4e-10 works", /-?\d+/)
=> "maybe -<<-123>>.4e-10 works"

>> show_match("maybe --123.4e-10 works", /-?\d*/)  # *, not +
=> "<<>>maybe --123.4e-10 works"
```

The *, +, and ? quantifiers, continued

What does **a(12|21|3)*b** describe?

Matches strings like **ab**, **a3b**, **a312b**, and **a3123213123333b**.

Write an RE to match numbers with commas, like these:

```
58   4,297   1,000,000   446,744   73,709,551,616
```

/(\d?\d?\d)(,\d\d\d)*/ -- Alan Smith

```
(\d\d\d|\d\d|\d)(,\d\d\d)*      # Why is \d\d\d first?
```

Write an RE to match floating point literals, like these:

```
1.2    .3333e10    -4.567e-30   .0001
```

```
>> %w{1.2 .3333e10 -4.567e-30 .0001}.
     each {|s| puts show_match(s, /-?\d*\.\d+(e-?\d+)?/) }
<<1.2>>
<<.3333e10>>
<<-4.567e-30>>
<<.0001>>
```

Note the \. to match only a period.

`*`, `+`, and `?` are greedy!

The operators `*`, `+`, and `?` are "greedy"—each tries to match the longest string possible, and cuts back only to make the full expression succeed.

Example:

Given `a.*b` and the input '`abbb`', the first attempt is:

- `a` matches `a`
- `.*` matches `bbb`
- `b` fails—no characters left!

The matching algorithm then *backtracks* and does this:

- `a` matches `a`
- `.*` matches `bb`
- `b` matches `b`

*, +, and ? are greedy, continued

More examples of greedy behavior:

```
>> show_match("xabbbbc", /a.*b/)
=> "x<<abbbb>>c"

>> show_match("xabbbbc", /ab?b?/)
=> "x<<abb>>bbc"

>> show_match("xabbbbcxyzc", /ab?b?.*c/)
=> "x<<abbbbcxyzc>>"
```

Why are *, +, and ? greedy?

Lazy/reluctant quantifiers

In the following we'd like to match just '**abc**' but the greedy asterisk goes too far:

```
    show_match("x + 'abc' + 'def' + y", /'.*'/)
    => "x + <<'abc' + 'def'>> + y"
```

We can make ***** lazy by putting **?** after it, causing it to match only as much as needed to make the full expression match. Example:

```
    >> show_match("x + 'abc' + 'def' + y", /'.*?'/)
    => "x + <<'abc'>> + 'def' + y"
```

?? and **+?** are supported, too. The three are also called *reluctant quantifiers*.

Once upon a time, before ***?** was supported, one would do this:

```
    >> show_match("x + 'abc' + 'def' + y", /'[^']+'/)
    => "x + <<'abc'>> + 'def' + y"
```

Specific numbers of repetitions

We can use curly braces to require a specific number of repetitions:

```
>> show_match("Call me at 555-1212!", /\d{3}-\d{4}/)
=> "Call me at <<555-1212>>!"
```

There are also forms with **{min,max}** and **{min,}**

```
>> show_match("3/17/2013", /\d{1,2}\/\d{1,2}\/(\d{4}|\d{2})/)
=> "<<3/17/2013>>"
```

Note that the RE above has escaped slashes to match the literal slashes.

split and **scan** with regular expressions

We can split a string using a regular expression:

```
>> " one, two,three / four".split(/[\s,\/]+/) # w.s., commas, slashes
=> ["", "one", "two", "three", "four"]
```

Note that leading delimiters produce an empty string in the result.

If we can describe the <u>strings of interest</u> instead of <u>what separates them</u>, **scan** is a better choice:

```
    >> " one, two,three / four".scan(/\w+/)
    => ["one", "two", "three", "four"]

    >> "10.0/-1.3...5.700+[1.0,2.3]".scan(/-?\d+\.\d+/)
    => ["10.0", "-1.3", "5.700", "1.0", "2.3"]
```

Here's a way to keep all the pieces:

```
    >> " one, two,three / four".scan(/\w+|\W+/)
    => [" ", "one", ", ", "two", ",", "three", " / ", "four"]
```

Anchors

Reminder: `s =~ /x/` succeeds if "`x`" appears anywhere in `s`.

The metacharacter `^` is an *anchor* when used at the start of a RE. (At the start of a character class it means to complement.)

`^` doesn't match any characters but it constrains the following regular expression to appear at the beginning of the string being matched against.

```
>> show_match("this is x", /^x/)          => "no match"

>> show_match("this is x", /^this/)       => "<<this>> is x"
```

What will `/^x|y/` match? Hint: it's not the same as `/^(x|y)/`

What does `/^.[^0-9]/` match?

TODO: Talk about newlines, and \A, \Z, and \z

Anchors, continued

Another anchor is **$**. It constrains the preceding regular expression to appear at the end of the string.

```
    >> show_match("ending", /end$/)
    => "no match"

    >> show_match("the end", /end$/)
    => "the <<end>>"
```

What does /\d+$/ match?

    Can it be shortened?

Anchors, continued

We can combine the **^** and **$** anchors to fully specify a string.

Problem: Write a RE to match lines with only a curly brace and (maybe) whitespace.

```
>> show_match(" }   ", /^\s*[{}]\s*$/)
=> "<< }   >>"
```

Using **grep**, print lines in Ruby source files that are exactly three characters long.

```
% grep ^...$ *.rb
```

The sets of metacharacters recognized by **grep**, **egrep**, and **egrep -P** differ. **fgrep** treats all characters as literals. The set in **egrep -P** is closest to Ruby, but there's also **rgrep.rb** from slide 210.

Anchors, continued

What does `/\w+\d+/` specify?

One or more "word" characters followed by one or more digits.

How do the following matches differ from each other?

```
line =~ /\w+\d+/

line =~ /^\w+\d+/

line =~ /\w+\d+$/

line =~ /^\w+\d+$/

line =~ /^.\w+\d+.$/

line =~ /^.*\w+\d+$/
```

Sidebar: Dealing with too much input

Imagine a program that's reading dozens of large data files whose lines start with first names, like "**Mary**". We're getting drowned by the data.

```
for fname in files
  f = open(fname)
  while line = f.gets
    ...lots of processing to build a data structure, bdata...
  end
  p bdata        # outputs way too much to easily analyze!!
```

We *could* edit data files down to a few names but here's an RE-based solution.

```
for fname in files
  f = open(fname)
  while line = f.gets
     next unless line =~ /^(John|Dana|Mary),/
     ...processing...      # toomuch.rb
```

Note trailing comma!

Sidebar: **convow.rb** with anchors

Recall that **convow.rb** on slide 223 simply does **char.downcase** on any characters it doesn't recognize. **downcase** doesn't change **^** or **$**.

The command

```
% ruby convow.rb ^cvc$
```

builds this this RE

```
/^[^aeiou][aeiou][^aeiou]$/
```

web2 is in **spring16**

Let's explore with it:

```
% ruby convow.rb ^cvc$ < web2 | wc -l
858
% ruby convow.rb ^vccccv$ < web2 | wc -l
15
% ruby convow.rb ^vcccccv$ < web2
|oxyphyte|
```

Named groups

The following regular expression uses three *named groups* to capture the elements of a binary arithmetic expression

```
>> re = /(?<lhs>\d+)(?<op>[+\-*\/])(?<rhs>\d+)/
```

After a successful match, the predefined global **$~**, an instance of **MatchData**, shows us the groups:

```
>> re =~ "What is 100+23?"
=> 8

>> $~
=> #<MatchData "100+23" lhs:"100" op:"+" rhs:"23">

>> $~["lhs"]
=> "100"
```

Named groups are sometimes called *named backreferences* or *named captures*.

Named groups, continued

At hand:

```
/(?<lhs>\d+)(?<op>[+\-*\/])(?<rhs>\d+)/
```

Important: Named groups must always be enclosed in parentheses.

Consider the difference in these two REs:

```
/x(?<n>\d+)/
```

Matches strings like "`x10`" and "`testx7ing`"

```
/x?<n>\d+/
```

Matches strings like "`<n>10`", "`ax<n>10`", "`testx<n>10ing`"

Design lesson:

"`(?`" in a RE originally had no meaning, so it provided an opportunity for extension without breaking any existing REs.

Post-lecture addition: See "NAMED CAPTURES AND LOCAL VARIABLES" in RPL.

Application: Time totaling

Consider an application that reads elapsed times on standard input and prints their total:

```
% ruby ttl.rb
3h
15m
4:30
^D
7:45
```

Multiple times can be specified per line, separated by spaces and commas.

```
% ruby ttl.rb
10m, 3:30
20m 2:15 1:01 3h
^D
10:16
```

How can we approach it?

Time totaling, continued

```
def main
  mins = 0
  while line = gets do
    line.scan(/[^\s,]+/).each {|time| mins += parse_time(time) }
  end
  printf("%d:%02d\n", mins / 60, mins % 60)
end

def parse_time(s)
  if s =~ /^(?<hours>\d+):(?<mins>[0-5]\d)$/
    $~["hours"].to_i * 60 + $~["mins"].to_i
  elsif s =~ /^(?<n>\d+)(?<which>[hm])$/
    n = $~["n"].to_i
    if $~["which"] == "h" then n * 60
                          else n end
  else
    0 # return 0 for things that don't look like times
  end
end
main
```

Example: consuming a string

Problem: Write a method **pt(s)** that takes a string like "**[(10,'a'),(3,'x'),(7,'o')]**" and returns an array with the sum of the numbers and a concatenation of the letters. If **s** is malformed, **nil** is returned.

Examples:

```
>> pt "[(10,'a'),(3,'x'),(7,'o')]"
=> [20, "axo"]

>> pt "[(100,'c')]"
=> [100, "c"]

>> pt "[(10,'x'),(5,7,'y')]"
=> nil

>> pt "[(10,'x'),(5,'y'),]"
=> nil
```

Example, continued

Desired:

```
>> pt "[(10,'a'),(3,'x'),(7,'o')]"
=> [20, "axo"]
```

Approach:

1. Remove outer brackets: "(10,'a'),(3,'x'),(7,'o')"
2. Append a comma: "(10,'a'),(3,'x'),(7,'o'),". (Why?!)
3. Recognize (NUM,LET), and replace with ""
4. Repeat 3. until failure
5. If nothing left but an empty string, success!

Important: By appending that comma we produce a simple repetition,

(*tuple* ,)+

rather than

tuple (, *tuple*)*

Example, continued

Solution:

```
def pt(s) # process_tuples.rb
    if s =~ /^\[(?<tuples>.*)\]$/ then
        tuples = $~["tuples"] + ","
        sum, lets = 0, ""
        tuples.gsub!(/\((?<num>\d+),'(?<let>[a-z])'\),/) do
            sum += $~["num"].to_i
            lets << $~["let"]
            ""  # block result--replaces matched string in tuples
        end
        if tuples.empty? then
            [sum,lets]
        end
    end
end
```

TODO: Named groups can set local variables

Approach:

1. Remove outer brackets
2. Append a comma
3. Recognize **(NUM,LET)**, and replace with ""
4. Repeat 3. until failure
5. If nothing left but an empty string, success!

Avoiding repetitious REs

calc.rb on assignment 6 accepts input lines such as these:

```
x=7
yval=x+10*x
x+yval+z
```

Here's a very repetitious RE that recognizes **calc.rb** input lines:

```
valid_line = /^([a-zA-Z][a-zA-Z\d]*=)?([a-zA-Z][a-zA-Z\d]*|\d
+)([-+*\/]([a-zA-Z][a-zA-Z\d]*|\d+))*$/
```

Let's use some intermediate variables to build that same RE.

```
var = /[a-z][a-z\d]*/i       # trailing "i": case insensitive

expr = /(#{var}|\d+)/

op = /[-+*\/]/

valid_line = /^(#{var}=)?#{expr}(#{op}#{expr})*$/
```

Lots more with regular expressions

Our look at regular expressions ends here but there's lots more, like...

- Back references—`/(.)(.).\2\1/` matches 5-character palindromes
- Nested regular expressions
- Nested and conditional groups
- Conditional subpatterns
- Zero-width positive lookahead

Groups can be accessed in code with **`$1, $2, ...`**

Proverb:

A programmer decided to use regular expressions to solve a problem.
Then the programmer had two problems.

Regular expressions are great, up to a point.

SNOBOL4 patterns, Icon's string scanning facility, and Prolog grammars can all recognize unrestricted languages and are far less complex than the regular expression facility in most languages.

Defining classes

A tally counter

Imagine a class named **Counter** that models a tally counter.

Here's how we might create and interact with an instance of Counter:

```
c1 = Counter.new
c1.click
c1.click

puts c1    # Output: Counter's count is 2
c1.reset

c2 = Counter.new "c2"
c2.click

puts c2    # Output: c2's count is 1

c2.click
puts "c2 = #{c2.count}"    # Output: c2 = 2
```

Counter, continued

Here is a partial implementation of **Counter**:

```
class Counter
   def initialize(label = "Counter")
      ...
   end
   ...
end  # Counter.rb
```

Class definitions are bracketed with **class** and **end**. Class names must start with a capital letter. Unlike Java there are no filename requirements.

The **initialize** method is the constructor, called when **new** is invoked.

```
c1 = Counter.new
c2 = Counter.new "c2"
```

If no argument is supplied to **new**, the default value of "**Counter**" is used.

Counter, continued

Here is the body of **initialize**:

```
class Counter
   def initialize(label = "Counter")
      @count = 0
      @label = label
   end
end
```

Instance variables are identified by prefixing them with **@**.

An instance variable comes into existence when it is assigned to. The code above creates **@count** and **@label**. (There are no instance variable declarations.)

Just like Java, each object has its own copy of instance variables.

Counter, continued

Let's add **click** and **reset** methods, which are straightforward:

```
class Counter
   def initialize(label = "Counter")
      @count = 0
      @label = label
   end

   def click
      @count += 1
   end

   def reset
      @count = 0
   end
end
```

Counter, continued

In Ruby the instance variables of an object cannot by accessed by any other object.

The only way way to make the value of **@count** available to other objects is via methods.

Here's a simple "getter" for the counter's count.

```
def count
   @count
end
```

Let's override **Object#to_s** with a **to_s** that produces a detailed description:

```
def to_s
   return "#{@label}'s count is #{@count}"
end
```

In Ruby, there is simply no such thing as a public instance variable. All access must be through methods.

Counter, continued

Full source for **Counter** thus far:

```
class Counter
   def initialize(label = "Counter")
      @count = 0; @label = label
   end

   def click
      @count += 1
   end

   def reset
      @count = 0
   end

   def count # Note the convention: count, not get_count
      @count
   end

   def to_s
      return "#{@label}'s count is #{@count}"
   end
end # Counter.rb
```

Common error: omitting the **@** on a reference to an instance variable.

An interesting thing about instance variables

Consider this class: (instvar.rb)

```
class X
   def initialize(n)
      case n
         when 1 then @x = 1
         when 2 then @y = 1
         when 3 then @x = @y = 1
 end; end; end
```

What's interesting about the following?

```
>> X.new 1
=> #<X:0x00000101176838 @x=1>

>> X.new 2
=> #<X:0x00000101174970 @y=1>

>> X.new 3
=> #<X:0x0000010117aaa0 @x=1, @y=1>
```

Instances of a class can have differing sets of instance variables!

Addition of methods

If **class X ... end** has been seen and another **class X ... end** is encountered, the second definition adds and/or replaces methods.

Let's confirm **Counter** has no **label** method.

```
 >> c = Counter.new "ctr 1"

>> c.label
NoMethodError: undefined method `label' ...
```

Now we add a **label** method: (we're typing lines into **irb** but could **load**)

```
>> class Counter
>>   def label; @label; end
>> end

>> c.label       => "ctr 1"
```

What's an implication of this capability?

We can add methods to classes written by others!

Addition of methods, continued

In Icon, the unary **?** operator can be used to generate a random number or select a random value from an aggregate.

```
Icon Evaluator, Version 1.1
][ ?10
  r1 := 3  (integer)

][ ?"abcd"
  r2 := "b"  (string)
```

I miss that. Let's add something similar to Ruby!

If we call **Kernel#rand** with a **Fixnum n** it will return a random **Fixnum r** such that **0 <= r < n**.

There's no unary **?** to overload in Ruby so let's just add a **rand** method to **Fixnum** and **String**.

Addition of methods, continued

Here is random.rb:

```
class Fixnum
   def rand
     Kernel.rand(self)+1
   end
end

class String
   def rand
     self[size.rand-1]    # Uses Fixnum.rand
   end
end
```

MONKEY PATCHING

```
>> load "random.rb"
>> 12.times { print 6.rand, " " }
2 1 2 4 2 1 4 3 4 4 6 3

>> 8.times { print "HT".rand, " " }
H H T H T T H H
```

An interesting thing about class definitions

Observe the following. What does it suggest to you?

```
>> class X
>> end
=> nil

>> p (class Y; end)
nil
=> nil

>> class Z; puts "here"; end
here
=> nil
```

Class definitions are executable code!

Class definitions are executable code

At hand: A class definition is executable code. The following class definition uses a **case** statement to selectively execute **def**s for methods.

```
class X
   print "What methods would you like? "
   gets.split.each do |m|
      case m
         when "f" then def f; "from f" end
         when "g" then def g; "from g" end
         when "h" then def h; "from h" end
         end
   end
end
```

Use:

```
>> load "dynmethods1.rb"
What methods would you like? f  g
>> x = X.new        => #<X:0x007fc45c0b0f40>
>> x.f              => "from f"
>> x.g              => "from g"
>> x.h
NoMethodError: undefined method `h' for #<X:...>
```

Sidebar: Fun with **eval**

Kernel#eval parses a string containing Ruby source code and executes it.

```
>> s = "abc"

>> n = 3

>> eval "x = s * n"          => "abcabcabc"

>> x                         => "abcabcabc"

>> eval "x[2..-2].length"    => 6

>> eval gets
s.reverse
=> "cba"
```

Note that **eval** uses variables from the current scope and that an assignment to **x** is reflected in the current scope. (Note: There are details about scoping!)

Bottom line: A Ruby program can generate code for itself.

Sidebar, continued

mk_methods.rb prompts for a method name, parameters, and method body. It then creates that method and adds it to class **X**.

```
    >> load "mk_methods.rb"
    What method would you like? add
    Parameters? a, b
    What shall it do? a + b
    Method add(a, b) added to class X

    What method would you like? last
    Parameters? x
    What shall it do? x[-1]
    Method last(x) added to class X

What method would you like? ^D => true
    >> x = X.new        => #<X:0x0000010185d930>
    >> x.add(3,4)       => 7
    >> x.last "abcd"    => "d"
```

Sidebar, continued

Here is **mk_methods.rb.** Note that the body of the class is a **while** loop.

```
class X
   while (print "What method would you like? "; name = gets)
      name.chomp!

      print "Parameters? "
      params = gets.chomp

      print "What shall it do? "
      body = gets.chomp

      code = "def #{name} #{params}; #{body}; end"

      eval(code)
      print("Method #{name}(#{params}) added to class #{self}\n\n");
   end
end
```

Is this a useful capability or simply fun to play with?

Sidebar: Risks with eval

Does eval pose any risks?

```
while (print("? "); line = gets)
   eval(line)
end # eval1.rb
```

Interaction: (input is underlined)

```
% ruby eval1.rb
? puts 3*5
15
? puts "abcdef".size
6
? system("date")
Mon Mar 23 19:09:35 MST 2015
? system("rm –rf ...")
...
? system("chmod 777 ...")
...
```

Sidebar, continued

At hand:

```
% ruby eval1.rb
? system("rm –rf ...")
...
? system("chmod 777 ...")
...
```

```
while (print("? "); line = gets)
     eval(line)
end # eval1.rb
```

But, we can do those things without using Ruby!

eval gets risky when we can't trust the source of the data. Examples:

A collaborator on a project sends us a data file.

A Ruby on Rails web app calls **eval** with user-supplied data. (!)

It's very easy to fall victim to a variety of *code-injection attacks* when using **eval**.

The **define_method** (et. al) machinery is often preferred over **eval** but risks still abound!

Related topic: Ruby supports the notion of *tainted* data.

Class variables and methods

Like Java, Ruby provides a way to associate data and methods with a class itself rather than each instance of a class.

Java uses the **static** keyword to denote a class variable.

In Ruby a variable prefixed with two at-signs is a class variable.

Here is **Counter** augmented with a class variable that keeps track of how many counters have been created.

```
class Counter
   @@created = 0 # Must precede any use of @@created
   def initialize(label = "Counter")
      @count, @label = 0, label
      @@created += 1
   end
end
```

Note: Unaffected methods are not shown.

Class variables and methods, continued

To define a class method, simply prefix the method name with the name of the class:

```
class Counter
    @@created = 0
    ...
    def Counter.created
        return @@created
    end
end
```

Usage:

```
>> Counter.created              => 0
>> c = Counter.new
>> Counter.created              => 1
>> 5.times { Counter.new }
>> Counter.created              => 6
```

A little bit on access control

By default, methods are public. If **private** appears on a line by itself, subsequent methods in the class are private. Ditto for **public**.

```
class X
    def f; puts "in f"; g end     # Note: calls g
  private
    def g; puts "in g" end
end
```

Usage:

```
>> x = X.new
>> x.f
in f
in g
>> x.g
NoMethodError: private method `g' ...
```

Speculate: What are **private** and **public**? Keywords?
Methods in **Module**! (**Module** is an ancestor of **Class**.)

Getters and setters

If **Counter** were in Java, we might provide methods like **void setCount(int n)** and **int getCount()**.

Our **Counter** already has a **count** method as a "getter".

For a "setter" we implement **count=**, with a trailing equals sign.

```
def count= n
     puts "count=(#{n}) called"   # Just for observation (LHtLAL)
     @count = n unless n < 0
 end
```

Usage:

```
>> c = Counter.new
>> c.count = 10
count=(10) called
=> 10
>> c                          => Counter's count is 10
```

Getters and setters, continued

Here's a class to represent points on a Cartesian plane:

```
class Point
   def initialize(x, y)
      @x = x
      @y = y
   end
   def x; @x end
   def y; @y end
end
```

Usage:

```
>> p1 = Point.new(3,4)  => #<Point:0x00193320 @x=3, @y=4>
>> [p1.x, p1.y]         => [3, 4]
```

It can be tedious and error prone to write a number of simple getter methods like **Point#x** and **Point#y**.

Getters and setters, continued

The method **attr_reader** creates getter methods.

Here's an equivalent definition of **Point**:

```
class Point
   def initialize(x, y)
      @x = x
      @y = y
   end
   attr_reader :x, :y
end
```

Usage:

```
>> p = Point.new(3,4)
>> p.x                          => 3
>> p.x = 10
NoMethodError: undefined method `x=' for #<Point: ...>
```

Why does **p.x = 10** fail?

Getters and setters, continued

If you want both getters and setters, use **attr_accessor**.

```
class Point
   def initialize(x, y)
      @x = x
      @y = y
   end
   attr_accessor :x, :y
end
```

Usage:

```
>> p = Point.new(3,4)
>> p.x
=> 3
>> p.y = 10
```

It's important to appreciate that **attr_reader** and **attr_accessor** are methods that create methods. (What if Ruby didn't provide them?)

Operator overloading

Operator overloading

In most languages at least a few operators are "overloaded"—an operator stands for more than one operation.

C: **+** is used to express addition of integers, floating point numbers, and pointer/integer pairs.

Java: **+** is used to express addition and string concatenation.

Icon: ***x** produces the number of...

- characters in a string
- values in a list
- key/value pairs in a table
- results a "co-expression" has produced

Icon: **+** means only addition; **s1 || s2** is string concatenation

What are examples of overloading in Ruby? In Haskell?

Operators as methods

We've seen that Ruby operators can be expressed as method calls:

3 + 4 is 3.+(4)

Here's what subscripting means:

"abc"[2] is "abc".[](2)
"testing"[2,3] is "testing".[](2,3)

Unary operators are indicated by adding @ after the operator:

-5 is 5.-@()

!"abc" is "abc".!@()

Challenge: See if you can find a binary operation that can't be expressed as a method call.

Operator overloading, continued

Let's use a dimensions-only rectangle class to study overloading in Ruby:

```
class Rectangle
   def initialize(w,h)
      @width, @height = w, h
   end
   attr_reader :width, :height
   def area; width * height; end
   def inspect
       "#{width} x #{height} Rectangle"
   end
end
```

Usage:

```
>> r = Rectangle.new(3,4)    => 3 x 4 Rectangle
>> r.area                    => 12
>> r.width                   => 3
```

Operator overloading, continued

Let's imagine that we can compute the "sum" of two rectangles:

```
>> a = Rectangle.new(3,4)   => 3 x 4 Rectangle

>> b = Rectangle.new(5,6)   => 5 x 6 Rectangle

>> a + b                    => 8 x 10 Rectangle

>> c = a + b + b            => 13 x 16 Rectangle

>> (a + b + c).area         => 546
```

As shown above, what does `Rectangle + Rectangle` mean?

Operator overloading, continued

Our vision:

```
>> a = Rectangle.new(3,4); b = Rectangle.new(5,6)
>> a + b => 8 x 10 Rectangle
```

Here's how to make it so:

```
class Rectangle
  def + rhs
     Rectangle.new(self.width + rhs.width, self.height + rhs.height)
  end
end
```

Remember that **a + b** is equivalent to **a.+(b)**. We are invoking the method "**+**" on **a** and passing it **b** as a parameter.

The parameter name, **rhs**, stands for "right-hand side".

Do we need **self** in **self.width** or would just **width** work? How about **@width**?

<u>Even if somebody else had provided **Rectangle**, we could still overload **+**</u> on it—the lines above are additive, assuming **Rectangle.freeze** hasn't been done.

Operator overloading, continued

For reference:

```
 def + rhs
   Rectangle.new(self.width + rhs.width, self.height + rhs.height)
 end
```

Here is a **faulty implementation** of our idea of rectangle addition:

```
  def + rhs
    @width += rhs.width; @height += rhs.height
  end
```

What's wrong with it?

```
  >> a = Rectangle.new(3,4)
  >> b = Rectangle.new(5,6)

  >> c = a + b                          => 10

  >> a                                  => 8 x 10 Rectangle
```

The problem:

> *We're changing the attributes of the left operand instead of creating and returning a new instance of* ***Rectangle****.*

Operator overloading, continued

Just like with regular methods, we have complete freedom to define what's meant by an overloaded operator.

Here is a method for **Rectangle** that defines unary minus to be imperative "rotation" (a clear violation of the Principle of Least Astonishment!)

```
def -@      # Note: @ suffix to indicate unary form of -
    @width, @height = @height, @width
    self
end

>> a = Rectangle.new(2,5)   => 2 x 5 Rectangle
>> -a                       => 5 x 2 Rectangle
>> a + -a                   => 4 x 10 Rectangle
>> a                        => 2 x 5 Rectangle
```

Goofy, yes?

Operator overloading, continued

At hand:

```
def -@
    @width, @height = @height, @width
    self
end
```

What's a more sensible implementation of unary -?

```
def -@
    Rectangle.new(height, width)
end

>> a = Rectangle.new(5,2)   => 5 x 2 Rectangle
>> -a                       => 2 x 5 Rectangle
>> a                        => 5 x 2 Rectangle
>> a += -a; a               => 7 x 7 Rectangle
```

Operator overloading, continued

Consider "scaling" a rectangle by some factor. Example:

```
>> a = Rectangle.new(3,4)   => 3 x 4 Rectangle
>> b = a * 5                => 15 x 20 Rectangle
>> c = b * 0.77             => 11.55 x 15.4 Rectangle
```

Implementation:

```
def * rhs
   Rectangle.new(self.width * rhs, self.height * rhs)
end
```

A problem:

```
>> a          => 3 x 4 Rectangle
>> 3 * a
TypeError: Rectangle can't be coerced into Fixnum
```

What's wrong?

We've implemented only Rectangle * Fixnum

Operator overloading, continued

Imagine a case where it's useful to reference width and height uniformly, via subscripts:

```
>> a = Rectangle.new(3,4)   => 3 x 4 Rectangle
>> a[0]                     => 3
>> a[1]                     => 4
>> a[2]                     RuntimeError: out of bounds
```

Note that `a[n]` is `a.[ ](n)`

Implementation:

```
def [] n
   case n
      when 0 then width
      when 1 then height
      else raise "out of bounds"
   end
end
```

Is Ruby extensible?

A language is considered to be extensible if we can create new types that can be used as easily as built-in types.

Does our simple **Rectangle** class and its overloaded operators demonstrate that Ruby is extensible?

What would **a = b + c * 2** with **Rectangle**s look like in Java?
    Maybe: **Rectangle a = b.plus(c.times(2));**

How about in C?
    Would **Rectangle a = rectPlus(b, rectTimes(c, 2));** be workable?

Haskell goes further with extensibility, allowing new operators to be defined.

Ruby is mutable

Ruby is not only extensible; it is also mutable—we can change the meaning of expressions.

If we wanted to be sure that a program never used integer addition, we could start with this:

```
class Fixnum
   def + x
      raise "boom!"
   end
end
```

What else would we need to do?

Contrast: C++ is extensible, but not mutable. For example, in C++ you can define the meaning of **Rectangle * int** but you can't change the meaning of integer addition, as we do above.

Inheritance

A **Shape** hierarchy in Ruby

Here's the classic **Shape**/**Rectangle**/**Circle** inheritance example in Ruby:

```
class Shape
   def initialize(label)
      @label = label
   end

   attr_reader :label
end
```

```
class Rectangle < Shape
   def initialize(label, width, height)
      super(label)
      @width, @height = width, height
   end

   def area
      width * height
   end

   def inspect
      "Rectangle #{label} (#{width} x
#{height})"
   end

   attr_reader :width, :height
end
```

Rectangle < Shape specifies inheritance.

Note that **Rectangle** methods use the generated **width** and **height** methods rather than **@width** and **@height**.

Shape, continued

```
class Circle < Shape
  def initialize(label, radius)
    super(label)
    @radius = radius
  end

  attr_reader :radius

  def area
    Math::PI * radius * radius
  end

  def perimeter
    Math::PI * radius * 2
  end

  def inspect
    "Circle #{label} (r = #{radius})"
  end
end
```

Math::PI references the constant **PI** in the **Math** class.

Similarities to inheritance in Java

Inheritance in Ruby has a lot of behavioral overlap with Java:

- Subclasses inherit superclass methods.
- Methods in a subclass can call superclass methods.
- Methods in a subclass override superclass methods of the same name.
- Calls to a method **f** resolve to **f** in the most-subclassed (most-derived) class.

There are differences, too:

- Subclass methods can always access superclass fields.
- Superclass constructors aren't automatically invoked when creating an instance of a subclass.

There's no **abstract**

The **abstract** reserved word is used in Java to indicate that a class, method, or interface is abstract.

Ruby does not have any language mechanism to mark a class or method as abstract.

Some programmers put "abstract" in class names, like **AbstractWindow**.

A method-level practice is to have abstract methods raise an error if called:

```
class Shape
   def area
      raise "Shape#area is abstract"
   end
end
```

There is also an **abstract_method** "gem" (a package of code and more):

```
class Shape
   abstract_method :area
   ...
```

Inheritance is important in Java

A common use of inheritance in Java is to let us write code in terms of a superclass type and then use that code to operate on subclass instances.

With a **Shape** hierarchy in Java we might write a routine **sumOfAreas**:

```
static double sumOfAreas(Shape shapes[]) {
        double area = 0.0;
        for (Shape s: shapes)
            area += s.getArea();
        return area;
        }
```

We can make **Shape.getArea()** abstract to force concrete subclasses to implement **getArea()**.

sumOfAreas is written in terms of **Shape** but works with instances of any subclass of **Shape**.

Inheritance is less important in Ruby

Here is sumOfAreas in Ruby:

```
def sumOfAreas(shapes)
   area = 0.0
   for shape in shapes do
      area += shape.area
   end
   area
end
```

Does it rely on inheritance in any way?

Even simpler:

```
sum = shapes.inject (0.0) {|memo, shape| memo + shape.area }
```

Dynamic typing in Ruby makes it unnecessary to require common superclasses or interfaces to write polymorphic methods that operate on a variety of underlying types.

If you look closely, you'll find that some common design patterns are simply patterns of working with inheritance hierarchies in statically typed languages.

Example: **VString**

Imagine an abstract class **VString** with two concrete subclasses: **ReplString** and **MirrorString**.

A **ReplString** is created with a string and a replication count. It supports **size**, substrings with **[pos]** and **[start, len]**, and **to_s** operations.

```
>> r1 = ReplString.new("abc", 2)     => ReplString(6)

>> r1.size       => 6

>> r1[0]         => "a"

>> r1[10]        => nil

>> r1[2,3]       => "cab"

>> r1.to_s       => "abcabc"
```

VString, continued

A MirrorString represents a string concatenated with a reversed copy of itself.

```
>> m1 = MirrorString.new("abcdef")
=> MirrorString(12)

>> m1.to_s                  => "abcdeffedcba"

>> m1.size                  => 12

>> m1[3,6]                  => "deffed"
```

What's a trivial way to implement the VString/ReplString/MirrorString hierarchy?

A trivial **VString** implementation

```
class VString
   def initialize(s)
      @s = s
   end

   def [](start, len = 1)
      @s[start, len]
   end

   def size
      @s.size
   end

   def to_s
      @s.dup
   end

end
```

```
class ReplString < VString
   def initialize(s, n)
      super(s * n)
   end

   def inspect
      "ReplString(#{size})"
   end
end

class MirrorString < VString
   def initialize(s)
      super(s + s.reverse)
   end

   def inspect
      "MirrorString(#{size})"
   end
end
```

VString, continued

New requirements:

A **VString** can be created using either a **VString** or a **String**.
A **ReplString** can have a very large replication count.

Will **VString**s in constructors work with the implemetation as-is?

```
>> m2 = MirrorString.new(ReplString.new("abc",3))
NoMethodError: undefined method `reverse' for ReplString

>> r2 = ReplString.new(MirrorString.new("abc"),5)
NoMethodError: undefined method `*' for MirrorString
```

What's the problem?

The ***ReplString*** *and* ***MirrorString*** *constructors use* *** n** *and* **.reverse**

What will **ReplString("abc", 2_000_000_000_000)** do?

VString, continued

Here's some behavior that we'd like to see:

```
>> s1 = ReplString.new("abc", 2_000_000_000_000)
=> ReplString("abc",2000000000000)

>> s1[0]                        => "a"

>> s1[-1]                       => "c"

>> s1[1_000_000_000]            => "b"

>> s2 = MirrorString.new(s1)
=> MirrorString(ReplString("abc",2000000000000))

>> s2.size                      => 12000000000000

>> s2[-1]                       => "a"

>> s2[s2.size/2 - 3, 6]         => "abccba"
```

VString, continued

Let's review requirements:

- Both **ReplString** and **MirrorString** are subclasses of **VString**.
- A **VString** can be created using either a **String** or a **VString**.
- The **ReplString** replication count can be a **Bignum**.
- If **vs** is a **VString**, **vs[pos]** and **vs[pos,len]** produce **String**s.
- **VString#size** works, possibly producing a **Bignum**.
- **VString#to_s** "works" but is problematic with long strings.

How can we make this work?

VString, continued

Let's play computer!

```
>> s = MirrorString.new(ReplString.new("abc",1_000_000))
=> MirrorString(ReplString("abc",1000000))

>> s.size
=> 6000000

>> s[-1]
=> "a"

>> s[3_000_000]
=> "c"

>> s[3_000_000,6]
=> "cbacba"
```

VString stands for "virtual string"—the hierarchy provides the illusion of very long strings but uses very little memory.

To be continued, on assignment 7!

What data did you need to perform those computations?

Modules and "mixins"

Modules

A Ruby *module* can be used to group related methods for organizational purposes.

Imagine some methods to comfort a homesick Haskell programmer at Camp Ruby:

```
module Haskell
   def Haskell.head(a)   # Class method--prefixed with class name
      a[0]
   end

   def Haskell.tail(a)
      a[1..-1]
   end
   ...more...
end

>> a = [10, "twenty", 30, 40.0]
>> Haskell.head(a)
=> 10
>> Haskell.tail(a)
=> ["twenty", 30, 40.0]
```

Modules as "mixins"

In addition to providing a way to group related methods, a module can be "included" in a class. When a module is used in this way it is called a "mixin" because it mixes additional functionality into a class.

Here is a revised version of the **Haskell** module. The class methods are now written as instance methods; they use **self** and have no parameter:

```
module Haskell
   def head
      self[0]
   end

   def tail
      self[1..-1]
   end
end
```

Previous version:

```
module Haskell
   def Haskell.head(a)
      a[0]
   end

   def Haskell.tail(a)
      a[1..-1]
   end
end
```

Mixins, continued

We can mix our Haskell methods into the **Array** class like this:

```
% cat mixin1.rb
require './Haskell'     # loads ./Haskell.rb if not already loaded
class Array
   include Haskell
end
```

We can load **mixin1.rb** and then use **.head** and **.tail** on arrays:

```
>> load "mixin1.rb"
>> ints = (1..10).to_a => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.head
=> 1

>> ints.tail
=> [2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.tail.tail.head
=> 3
```

Mixins, continued

We can add those same capabilities to String, too:

```
class String
   include Haskell
end
```

Usage:

```
>> s = "testing"

>> s.head                   => "t"
>> s.tail                   => "esting"
>> s.tail.tail.head         => "s"
```

In addition to the include mechanism, what other aspect of Ruby facilitates mixins?

Modules and superclasses

The Ruby core classes and standard library make extensive use of mixins.

The class method **ancestors** can be used to see the superclasses and modules that contribute methods to a class:

```
>> Array.ancestors
=> [Array, Enumerable, Object, Kernel, BasicObject]

>> Fixnum.ancestors
=> [Fixnum, Integer, Numeric, Comparable, Object, Kernel,
BasicObject]

>> load "mixin1.rb"

>> Array.ancestors
=> [Array, Haskell, Enumerable, Object, Kernel, BasicObject]
```

Modules and superclasses, continued

The method **included_modules** shows the modules that a class **include**s.

```
>> Array.included_modules     => [Haskell, Enumerable, Kernel]

>> Fixnum.included_modules  => [Comparable, Kernel]
```

instance_methods can be used to see what methods are in a module:

```
>> Enumerable.instance_methods.sort =>
[:all?, :any?, :chunk, :collect, :collect_concat, :count, :cycle, :de
tect, :drop, :drop_while, :each_cons, :each_entry, ...more...

>> Comparable.instance_methods.sort
=> [:<, :<=, :==, :>, :>=, :between?]

>> Haskell.instance_methods
=> [:head, :tail]
```

The Enumerable module

When talking about iterators we encountered Enumerable. It's a module:

```
>> Enumerable.class
=> Module

>> Enumerable.instance_methods.sort =>
[:all?, :any?, :chunk, :collect, :collect_concat, :count, :cycle, :de
tect, :drop, :drop_while, :each_cons, :each_entry, :each_slice, :
each_with_index, :each_with_object, :entries, :find, :find_all, :f
ind_index, :first, :flat_map, :grep, :group_by, :include?, :inject,
:map, :max, :max_by, :member?, :min, :min_by, :minmax, :min
max_by, :none?, :one?, :partition, :reduce, :reject, ...
```

The methods in Enumerable use duck typing, requiring only an each method. min, max, and sort, also require <=> for values operated on.

If class implements each and includes Enumerable then all those methods become available to instances of the class.

The **Enumerable** module, continued

Here's a class whose instances simply hold three values:

```
class Trio
   include Enumerable
   def initialize(a,b,c); @values = [a,b,c]; end

   def each
      @values.each {|v| yield v }
   end
end
```

Because **Trio** implements **each** and includes **Enumerable**, lots of stuff works:

```
>> t = Trio.new(10, "twenty", 30)

>> t.member?(30) => true

>> t.map{|e| e * 2} => [20, "twentytwenty", 60]

>> t.partition {|e| e.is_a? Numeric } => [[10, 30], ["twenty"]]
```

What would the Java equivalent be for the above?

The **Comparable** module

Another common mixin is **Comparable**:

```
>> Comparable.instance_methods
=> [:==, :>, :>=, :<, :<=, :between?]
```

Comparable's methods are implemented in terms of **<=>**.

Let's compare rectangles on the basis of areas:

```
class Rectangle
    include Comparable
    def <=> rhs
        (self.area - rhs.area) <=> 0
    end
end
```

Comparable, continued

Usage:

```
>> r1 = Rectangle.new(3,4)  => 3 x 4 Rectangle
>> r2 = Rectangle.new(5,2)  => 5 x 2 Rectangle
>> r3 = Rectangle.new(2,2)  => 2 x 2 Rectangle

>> r1 < r2                     => false

>> r1 > r2                     => true

>> r1 == Rectangle.new(6,2)    => true

>> r2.between?(r3,r1)          => true
```

Is Comparable making the following work?

```
>> [r1,r2,r3].sort
=> [2 x 2 Rectangle, 5 x 2 Rectangle, 3 x 4 Rectangle]

>> [r1,r2,r3].min
=> 2 x 2 Rectangle
```

In conclusion...

What do you like (or not?) about Ruby?

- Everything is an object?
- Substring/subarray access with **x[...]** notation?
- Negative indexing to access from right end of strings and arrays?
- **if** modifiers? (**puts x if x > y**)
- Iterators and blocks?
- Ruby's support for regular expressions?
- ~~Monkey patching?~~ Adding methods to existing classes?
- Programmer-defined operator overloading?
- Dynamic typing?

Is programming more fun with Ruby?

If you know Python, do you prefer Python or Ruby?

My first practical Ruby program

September 3, 2006:

```
n=1
d = Date.new(2006, 8, 22)
incs = [2,5]
pos = 0
while d < Date.new(2006, 12, 6)
    if d != Date.new(2006, 11, 23)
        printf("%s %s, #%2d\n",
            if d.cwday() == 2: "T"; else "H";end,
                d.strftime("%m/%d/%y"), n)
        n += 1
    end
    d += incs[pos % 2]
    pos += 1
end
```

Output:

```
T 08/22/06, # 1
H 08/24/06, # 2
T 08/29/06, # 3
...
```

More with Ruby...

If we had more time, we'd...

- Learn about **lambda**s, blocks as explicit parameters, and **call**.
- Play with **ObjectSpace.** (Try **ObjectSpace.count_objects**)
- Do some metaprogramming with *hooks* like **method_missing**, **included**, and **inherited**.
- Experiment with internal Domain Specific Languages (DSL).
- Look at how Ruby on Rails puts Ruby features to good use.
- Write a Swing app with JRuby, a Ruby implementation for the JVM.
- Take a peek at BDD (Behavior-Driven Development) with Cucumber and RSpec.

Prolog

CSC 372, Spring 2016
The University of Arizona
William H. Mitchell
whm@cs

A little background on Prolog

The name comes from "programming in logic".

Developed at the University of Marseilles (France) in 1972.

First implementation was in FORTRAN and led by Alain Colmeraurer.

Originally intended as a tool for working with natural languages.

Achieved great popularity in Europe in the late 1970s.

Was picked by Japan in 1981 as a core technology for their "Fifth Generation Computer Systems" project.

Used in IBM's Watson for NLP (Natural Language Processing).

Prolog is a commercially successful language. Many companies have made a business of supplying Prolog implementations, Prolog consulting, and/or applications in Prolog.

Prolog resources

There are no Prolog books on Safari.

Here are two Prolog books that I like:

Prolog Programming in Depth, by Covington, Nute, and Vellino
Available for free at **http://www.covingtoninnovations.com/books/PPID.pdf**. That PDF is scans of pages and is not searchable.

The copy at **http://.../cs372/spring16/covington/ppid.pdf** has had a searchable text layer added.

Programming in Prolog, 5th edition, by Clocksin and Mellish ("C&M")
A PDF is available via a UA library link on the Piazza resources page.
(http://link.springer.com.ezproxy1.library.arizona.edu/book/10.1007%2F978-3-642-55481-0)

A PDF of Dr. Collberg's Prolog slides for 372 is here:
http://cs.arizona.edu/classes/cs372/spring16/CollbergProlog.pdf

There's no Prolog "home page" that I know of.

We'll be using SWI Prolog. More on it soon.

Facts and queries

Step one with Prolog

You'll eventually see lots of connections between elements of Prolog and other languages, especially Haskell, but for the moment...

Clear your mind!

Facts and queries

A Prolog program is a collection of *facts*, *rules*, and *queries*. We'll talk about facts first.

Here is a small collection of Prolog *facts*:

```
$ cat foods.pl            (in spring16/prolog/foods.PL)
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

These facts enumerate some things that are food. We might read them in English like this: "An apple is food", "Broccoli is food", etc.

A fact represents a piece of knowledge that the Prolog programmer deems to be useful. The name **food** was chosen by the programmer.

We can say that **facts.pl** holds a Prolog *database* or *knowledgebase.*

Facts and queries, continued

At hand:

```
$ cat foods.pl
food(apple).
food(broccoli).
...
```

food, **apple**, and **broccoli** are *atoms*, which can be thought of as multi-character literals. Atoms are not strings! Atoms are atoms!

Here are two more atoms:

```
'bell pepper'
'Whopper'
```

An atom can be written without single quotes if it starts with a lower-case letter and contains only letters, digits, and underscores.

Note the use of single quotes. (Double quotes mean something else!)

Facts and queries, continued

On lectura, we can start SWI Prolog and load a knowledgebase like this:

```
$ swipl foods.pl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
...
?-                          (?- is the swipl query prompt)
```

Once the knowledgebase is loaded we can perform *queries*:

```
?- food(carrot).
true.

?- food(pickle).
false.
```

Prolog responds based on the facts it has been given. We know that pickles are food but Prolog doesn't know that because there's no fact that says so.

Prolog queries have one or more *goals*. The queries above have one goal.

Facts and queries, continued

Here's a fact: **food(apple).**

Here's a query: **food(apple).**

Facts and queries have the same syntax. The interpretation depends on the context in which they appear.

If a line is typed at the interactive **?-** prompt, it is interpreted as a query.

If we do **swipl foods.pl**, the lines in **foods.pl** are interpreted as facts.

Loading a file of facts is also known as *consulting* the file.

We'll see later that files can contain "rules", too. Facts and rules are the two types of *clauses* in Prolog.

Simple rule for now: use all-lowercase filenames with the suffix **.pl** (PL) for Prolog knowledgebases (i.e. source files).

Sidebar: Reconsulting with **make**

After a **.pl** file has been consulted (loaded), we can query **make.** to cause any modified files to be reconsulted (reloaded), after editing the file.

```
$ swipl foods.pl
Welcome to SWI-Prolog ...

?- food(pickle).
false.
[Edit foods.pl in a different window, and add food(pickle).]

?- make.
% /home/whm/372/foods compiled 0.00 sec, 2 clauses
true.

?- food(pickle).
true.

?- make.
true.            (foods.pl hasn't changed since the last make)
```

Sidebar: Consulting via query

An alternative to specifying a file on the command line is to consult using a query:

```
$ swipl
Welcome to SWI-Prolog ...

?- [foods].       (do not include the .pl suffix)
% foods compiled 0.00 sec, 8 clauses
true.
```

Consulting a file via a query is commonly shown in texts.

The end result of the two methods is the same.

Sidebar: **food** in Haskell

How might the food information be represented in Haskell?

```
food "apple"        = True
food "broccoli"     = True
food "carrot"       = True
food "lettuce"      = True
food "rice"         = True
food  _             = False

> food "apple"
True
```

Maybe a list would be better:

```
foods = ["apple", "broccoli", "carrot", "lettuce", "rice"]

> "pickle" `elem` foods
False
```

How might we represent the food information in Ruby?

Facts and queries, continued

A query like **food(apple)** asks if it is known that apple is a food.

Speculate: What's the following query asking?

```
?- food(Edible).
Edible = apple <cursor is here>
```

Watch what happens when we type semicolons:

```
Edible = apple ;
Edible = broccoli ;
Edible = carrot ;
...
Edible = 'Big Mac'.
```

What's going on?

Facts and queries, continued

An alternative to specifying an atom, like `apple`, in a query is to specify a variable. An identifier that starts with a capital letter is a Prolog variable.

```
?- food(Edible).
Edible = apple <cursor is here>
```

The above query asks, "Tell me something that you know is a food."

Prolog finds the first `food` fact, based on file order, and responds with `Edible = apple`, using the variable name specified in the query.

If the user is satisfied with the answer `apple`, pressing `<ENTER>` terminates the query. Prolog responds by printing a period.

```
?- food(Edible).
Edible = apple .  % User hit <ENTER>; Prolog printed the period.

?-
```

Facts and queries, continued

If for some reason the user is not satisfied with the response **apple**, an alternative can be requested by typing a semicolon, without ***<ENTER>***.

```
?- food(Edible).
Edible = apple ;
Edible = broccoli ;
Edible = carrot ;
...
Edible = 'Big Mac'.

?-
```

Facts are searched in the order they appear in **foods.pl**. Above, the user exhausts all the facts by typing semicolon. Prolog prints '**.**' after the last.

Note that a simple set of facts lets us perform two distinct computations:

(1) We can ask if something is a food.
(2) We can ask what all the foods are.

How could we make an analog for those two in Java, Haskell, or Ruby?

Extra credit!

For three points of extra credit:

(1) Get a copy of **foods.pl** and try the examples previously shown.

(2) Create a small database (a file of facts) about something other than food and demonstrate some queries with it using **swipl**. Minimum: 5 facts.

(3) Copy/paste a transcript of your **swipl** session into a plain text file named **eca3.txt**.

(4) <u>Before the next lecture</u>, turn in **eca3.txt** with the following command:

```
$ turnin 372-eca3 eca3.txt
```

Needless to say, feel free to read ahead in the slides and show experimentation with the following material, too.

Experiment with syntax, too. Where can whitespace appear? What can appear in a fact other than atoms like **apple**?

Look ahead a few slides for information about installing SWI Prolog on your machine, or just use **swipl** on lectura.

Yes and **no** vs. **true.** and **false.**

Unlike SWI Prolog, most Prolog implementations use "yes" and "no" to indicate whether an interactive query succeeds. Here's GNU Prolog:

```
% gprolog
GNU Prolog 1.3.0
| ?- [foods].
compiling foods.pl for byte code...

| ?- food(apple).
yes

| ?- food(pickle).
no
```

Most Prolog texts, including Covington and C&M use **yes/no**, too. Just read "**yes**" as **true.** and "**no**" as **false.**

Remember: we're using SWI Prolog; GNU Prolog is shown above just for contrast.

"Can you prove it?"

One way to think about a query is that we're asking Prolog if something can be "proven" using the facts (and rules) it has been given.

The query

```
    ?- food(apple).
```

can be thought of as asking, "Can you prove that apple is a food?"

food(apple). is trivially proven because we've supplied a fact that says that apple is a food.

The query

```
    ?- food(pickle).
```

produces **false.** because Prolog can't prove that pickle is a food based on the database (the facts) we've supplied. (We've given it no rules, either.)

"Can you prove it?", continued

Consider again a query with a variable:

```
?- food(F).     % Remember that an initial capital denotes a variable.
F = apple ;
F = broccoli ;
F = carrot ;
...
F = 'Whopper' ;
F = 'Big Mac'.

?-
```

The query asks, "For what value~~s~~ of **F** can you prove that **F** is a food? By repeatedly entering a semicolon we see the full set of values for which that can be proven.

The collection of knowledge at hand, a set of facts about what is a food, is trivial but Prolog is capable of finding proofs for an arbitrarily complicated body of knowledge expressed as facts and rules.

"Can you prove it?", continued

write is one of many built-in *predicates*. It outputs a value.

```
?- write('Hello, world!').
Hello, world!
true.
```

Speculate: Why was "**true.**" output, too?

Prolog is reporting that it's able to prove **write('Hello, world!')!**

A side-effect of "proving" **write(X)** is outputting the value of **X**!

Speculate: What does Prolog think we're doing when we type **make.** ?

We're wanting to see if **make** can be proven!

A side effect of "proving" **make** is the knowledgebase is reconsulted (reloaded) if it's been modified.

Getting and running SWI Prolog

Getting and running SWI Prolog

swi-prolog.org is the home page for SWI Prolog.

On lectura, just run **swipl**.

For Windows, go to:

http://swi-prolog.org/download/stable

The non-64 bit version will be fine for our purposes:

SWI-Prolog 7.2.3 for Windows XP/Vista/7/8

Pick **Typical** as the **Install type**, **.pl** for file extension (or **.pro**, to avoid a collision with Perl)

Getting and Running SWI Prolog, continued

For OS X if you're **not** running El Capitan, go to

http://swi-prolog.org/download/stable

and get

SWI-Prolog 7.2.3 for MacOSX 10.6 (Snow Leopard) and later...

You'll need XQuartz 2.7.5 for development tools that use graphics, the handiest of which is perhaps the graphical tracer, launched with the **gtrace** predicate. (We'll see **gtrace** later.)

Set your firewall to block incoming connections for **X11.bin**.

If you are running El Capitan, go to

http://swi-prolog.org/download/devel

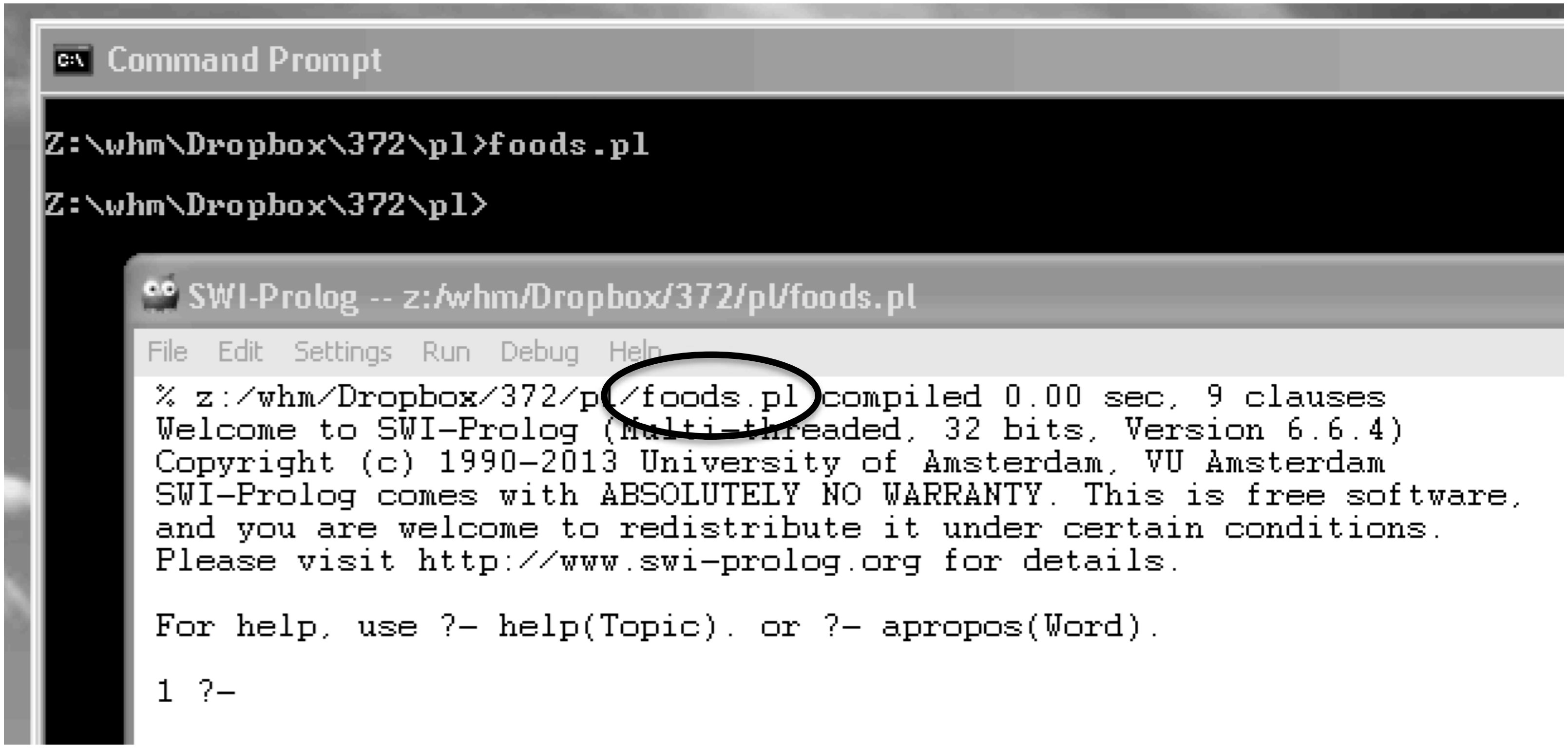
and get

SWI-Prolog 7.3.19 for MacOSX 10.6 (Snow Leopard) and ...

This version requires XQuartz 2.7.7

SWI Prolog on Windows

On Windows, assuming you associated **.pl** files with SWI Prolog, running **foods.pl** on the command line or opening **foods.pl** in Explorer opens a window running SWI Prolog and consults the file, as if **[foods].** had been typed at the prompt.



```
Z:\whm\Dropbox\372\pl>foods.pl

Z:\whm\Dropbox\372\pl>
```

```
% z:/whm/Dropbox/372/pl/foods.pl compiled 0.00 sec, 9 clauses
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.4)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
```

On Windows, a numbered query prompt is shown. ("**1 ?-**" above)

Remember: You can use **make.** to reconsult (reload) a file.

SWI Prolog on OS X

On my Mac, I have this alias in my **~/.bashrc**:

```
alias swipl='/Applications/SWI-Prolog.app/Contents/MacOS/swipl'
```

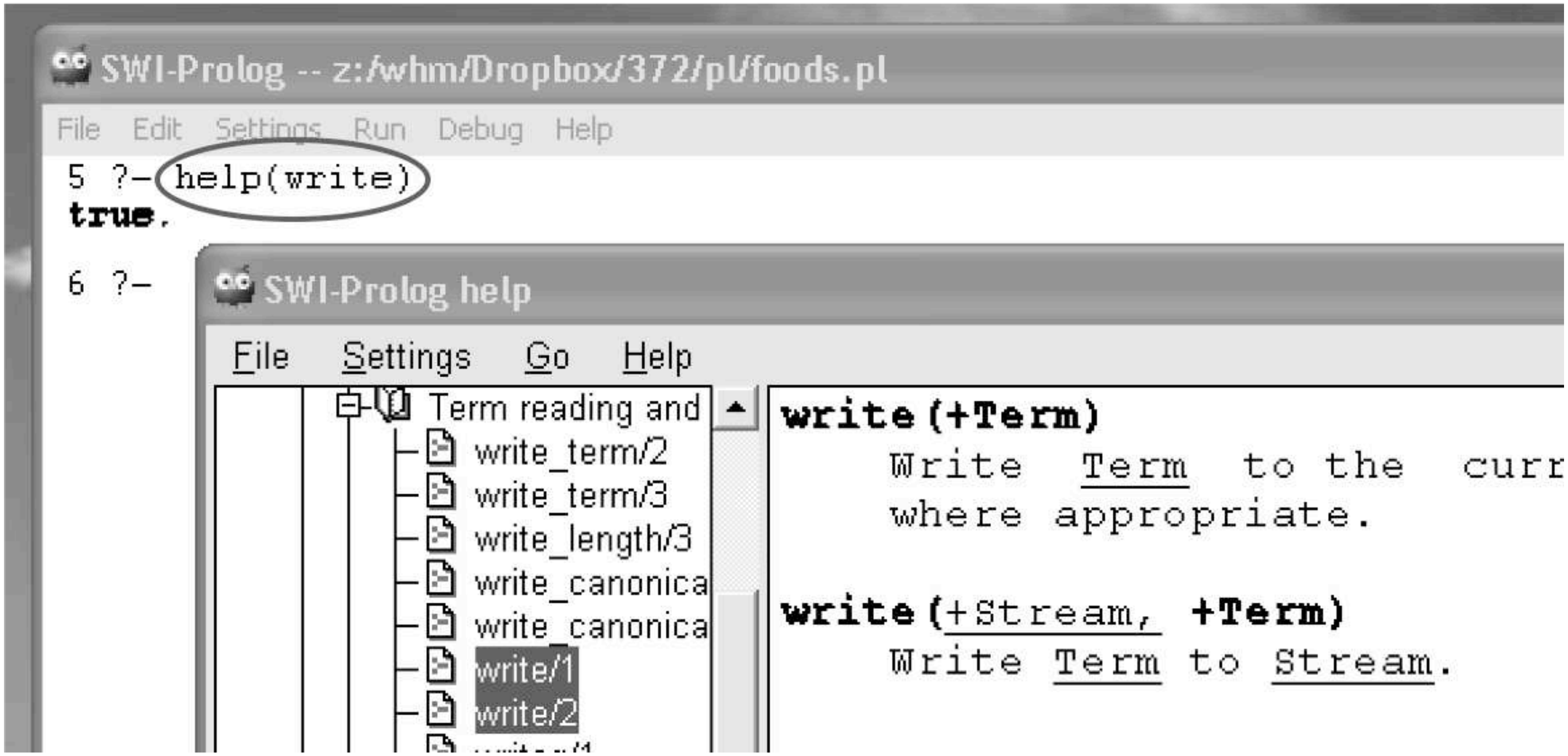
It lets me type **swipl** at the bash prompt.

If you have trouble with XQuartz, you can force text-only operation with this alias: (wrapped)

```
alias swipl='(export -n DISPLAY;  /Applications/SWI-Prolog.app/
Contents/MacOS/swipl)'
```

Getting help for predicates

To get help for a predicate, query **help(*predicate-name*).** On Windows you'll see:



```
5 ?- help(write)
true.

6 ?-
```

```
write(+Term)
    Write Term to the curr
    where appropriate.

write(+Stream, +Term)
    Write Term to Stream.
```

OS X will be similar, assuming you've got XQuartz 2.7.5+ installed. If not, or you're using the **swipl** alias with "**export -n DISPLAY...**", help will be text based.

Help will be text based on lectura, but if you login to lectura from a Linux machine in the CS labs with "**ssh -X ...**", you'll get window-based help there, too.

Getting out of SWI Prolog

On all platforms a control-D or querying **halt.** exits SWI Prolog.

```
$ swipl
...

?- halt.
$
```

A control-C while a query is executing will produce an **Action ...?** prompt. Then typing h produces a textual menu:

```
?- food(X).
X = apple ^C
Action (h for help) ? h
Options:
a:           abort       b:          break
c:           continue    e:          exit
g:           goals       t:          trace
h (?):       help
```

Use a to return to the prompt; e exits to the shell.

Building blocks

Atoms

We've seen that **apple**, **food**, and '**Big Mac**' are examples of *atoms*.

Typing an atom as a query doesn't do what we might expect!

```
?- 'just\ntesting'.
ERROR: toplevel: Undefined procedure: 'just\ntesting'/0 (DWIM
could not correct goal)
```

But we can output an atom with **write**.

```
?- write('just\ntesting').
just
testing
true.
```

Atoms composed of certain non-alphabetic characters do not require quotes:

```
?- write(#$&*+-./:<=>?^~\).
#$&*+-./:<=>?^~\
true.
```

Atoms, continued

We can use the predicate **atom** to query whether something is an atom:

```
?- atom(apple).
true.

?- atom('apple sauce').
true.

?- atom(Ant).
false.

?- atom("apple").
false.
```

Alternate view: "Can you prove **apple** is an atom?"

Numbers

Integer and floating point literals are *numbers*.

```
?- number(10).
true.

?- number(3.4).
true.

?- number(3.4e100).
true.

?- number('100').
false.
```

Numbers aren't atoms but they are "atomic" values.

```
?- atom(100).
false.

?- atomic(100).  % Note: atomic, not just atom.
true.
```

Numbers, continued

In Prolog, arithmetic doesn't work as you might expect:

```
?- 3 + 4.
ERROR: toplevel: Undefined procedure: (+)/2 (DWIM could
not correct goal)

?- y = 4 + 5.
false.

?- Y = 4 + 5.
Y = 4+5.

?- write(3 + 4 * 5).
3+4*5
true.
```

We'll learn about arithmetic later.

Predicates, terms, and structures

Here are some more examples of facts:

```
    color(sky, blue).  color(grass, green).

    odd(1). odd(3). odd(5).

    number(one, 1, 'English').
    number(uno, 1, 'Spanish').
    number(dos, 2, 'Spanish').
```

We can say that the facts above define three *predicates*: **color**, **odd**, and **number**.

"The collection of clauses for a given predicate is called a *procedure*."—C&M

It's common to refer to predicates using *predicate indicators* like **color/2**, **odd/1**, and **number/3**, where the number following the slash is the number of *terms*.

number/3 above doesn't collide with the built-in predicate **number/1** we saw earlier.

Predicates, terms, and structures, continued

A *term* is one of the following: atom, number, structure, variable.

Structures consist of a *functor* (always an atom) followed by one or more *terms* enclosed in parentheses.

Here are examples of structures:

```
color(grass, green)

odd(1)

'number'('uno', 1, 'Spanish') % 's not needed around number and uno

lunch(sandwich(ham), fries, drink(coke))
```

The structure functors are **color**, **odd**, **number**, and **lunch**, respectively.

Two of the terms of the **lunch** structure are structures themselves, with functors **sandwich** and **drink**.

A structure can serve as a fact or a goal, depending on the context.

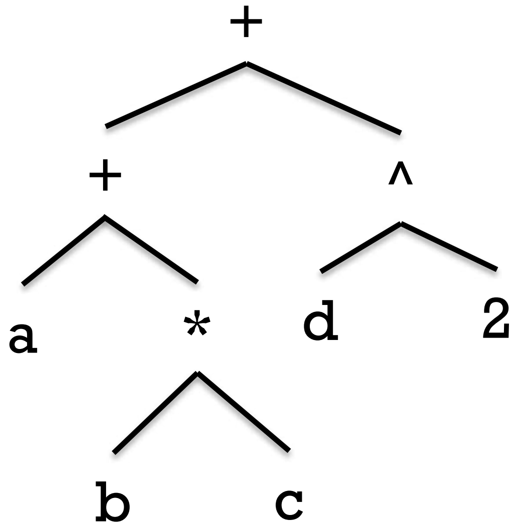
Structures with symbolic functors

Structures can have symbolic functors:

```
+(3,4)
+(3,*(4,5))
\/(x,y)
```

When Prolog encounters an expression with operators, it builds a structure. **display/1** can be used to examine such structures.

```
?- display(a+b*c+d^2).
+(+(a,*(b,c)),^(d,2))
true.

?- display(a /\ b \/ c /\ d).
/\(\/(/\(a,b),c),d)
true.
```



Some predicates evaluate structures but most do not, and simply treat the structure as a value.

Sidebar: **op/3**

Query **help(op)** to see the predefined operators and precedences. It shows this:

```
_________________________________________________________
| 1200 |xfx   |-->, :-                                   |
| 1200 | fx   |:-, ?-                                    |
| 1100 |xfy   |;, |                                      |
| 1050 |xfy   |->, *->                                   |
| 1000 |xfy   |,                                         |
|  990 |xfx   |:=                                        |
|  900 | fy   |\+                                        |
|  700 |xfx   |<, =, =.., =@=, \=@=, =:=, =<, ==,        |
|      |      |=\=, >, >=, @<, @=<, @>, @>=, \=, \==,    |
|      |      |as, is, >:<, :<                           |
|  600 |xfy   |:                                         |
|  500 | yfx  |+, -, /\, \/, xor                         |
|  500 | fx   |?                                         |
|  400 | yfx  |*, /, //, div, rdiv, <<, >>, mod, rem     |
|  200 |xfx   |**                                        |
|  200 |xfy   |^                                         |
|  200 | fy   |+, -, \                                   |
|  100 | yfx  |.                                         |
|_____1_|_fx__|$_________________________________________|
```

- Left column is precedence; `1200` is lowest.
- `xfy, yfx, fx, fy` etc. are specifications of associativity and infix/prefix/postfix forms.

op/3, continued

Operators can be created with **op/3**.

```
?- op(150,'xf',--).         % precedence 150 postfix operator
true.

?- op(200, xfy, @).         % right-associative infix operator
true.
```

The **f** in **xf** and **xfy** (above) specify where the functor can appear wrt. the operands.

```
?- display(x @ y @ zz--).
@(x,@(y,--(zz)))
true.
```

Operator predicates

Most operators are not predicates.

```
?- +(3,4).
ERROR: toplevel: Undefined procedure: (+)/2 ...
```

But a few operators are predicates. Two are \== and ==. They can be written in prefix or infix form:

```
?- \==(this,that).
true.

?- 3 == 3.
true.
```

They do deep comparison of their terms.

```
?- 3 == 2+1.
false.        % The number 3 is not equal to the structure 2+1.
```

== and \== do not produce a value! They simply succeed or fail.

Is **swipl** a REPL?

In conventional languages there are expressions.

A conventional REPL evaluates expressions and prints the value produced.

At **swipl**'s query prompt we can see if one or more goals can be proven.

In the process of trying to prove all the goals, side effects like output may occur and variables may be instantiated but the only result of evaluating goals is success or failure.

So is **swipl** a REPL?

More queries

More queries

```
$ cat foodcolor.pl
...food facts not shown...
color(sky, blue).
color(dirt, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(rice, white).
```

Here's a new knowledgebase.

A query about green things:

```
?- color(Thing, green).
Thing = grass ;
Thing = broccoli ;
Thing = lettuce.
```

How can we state it in terms of "Can you prove...?"

For what things can you prove that their color is green?

More queries

How could we query for each thing and its color?

```
?- color(Thing,Color).
Thing = sky,
Color = blue ;

Thing = dirt,
Color = brown ;

Thing = grass,
Color = green ;

Thing = broccoli,
Color = green ;
...
```

```
color(sky, blue).
color(dirt, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(rice, white).
```

How can we state it in terms of "Can you prove...?"

For what pairs of `Thing` *and* `Color` *can you prove* `color(Thing,Color)`*?*

Queries with multiple goals

A query can contain more than one goal.

Here's a query that directs Prolog to find a food that is green:

```
?- food(F), color(F,green).
F = broccoli ;
F = lettuce ;
false.
```

The query has two goals separated by a comma, which indicates conjunction—both goals must succeed in order for the query to succeed.

We might state it like this:

"Is there an F for which you can prove both food(F) and color(F, green)?

```
$ cat foodcolor.pl
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(orange).
food(rice).

color(sky, blue).
color(dirt, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(orange,orange).
color(rice, white).
```

Queries with multiple goals, continued

Let's see if any foods are blue:

```
    ?- color(F,blue), food(F).
    false.
```

Note that the ordering of the goals was reversed. How might the order make a difference?

<u>Goals are always tried from left to right.</u>

What's the following query asking?

```
  ?- food(F), color(F,F).
```

How about this one?

```
  ?- food(F), color(F,red), color(F,green).
```

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(orange).
food(rice).

color(sky, blue).
color(dirt, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(orange, orange).
color(rice, white).
```

Sidebar: The meaning of a fact

Which of the following is meant by `color(apple,red)`?

All apples are red.

Some apples are red.

Some apples have a red area.

Some apples have a red area at some point in time.

A red apple has existed.

Facts (and rules) are abstractions that we create for the purpose(s) at hand.

An abstraction emphasizes the important and suppresses the irrelevant.

Don't get bogged down by trying to perfectly model the real world!

Even more queries

Write these queries:

Who likes baseball?

```
?- likes(Who, baseball).
```

Who likes a food?

```
?- food(F), likes(Who,F).
```

Who likes green foods?

```
?- food(F), color(F,green),
likes(Who,F).
```

Who likes foods with the same color as foods that Mary likes?

```
?- likes(mary,F), food(F),
color(F, C), food(F2), color(F2,C),
likes(Who,F2).
```

```
$ cat fcl.pl
food(apple).
...more food facts...

color(sky, blue).
...more color facts...

likes(bob, carrot).
likes(bob, apple).
likes(joe, lettuce).
likes(mary, broccoli).
likes(mary, tomato).
likes(bob, mary).
likes(mary, joe).
likes(joe, baseball).
likes(mary, baseball).
likes(jim, baseball).
```

Even more queries, continued

Are any two foods the same color?

```
?- food(F1), food(F2), color(F1,C), color(F2,C).
F1 = F2, F2 = apple,   % an apple is the same color as an apple(!)
C = red ;

F1 = F2, F2 = broccoli,
C = green ;
...
```

Let's use \== to keep foods from matching themselves:

```
?- food(F1), food(F2), F1 \== F2, color(F1,C), color(F2,C).
F1 = broccoli,
F2 = lettuce,
C = green ;

F1 = carrot,
F2 = C, C = orange ;
...
```

Alternative representations

A given body of knowledge may be represented in a variety of ways using Prolog facts. Here is another way to represent the food and color information.

```
thing(apple, red, yes).
thing(broccoli, green, yes).
thing(carrot, orange, yes).
thing(dirt, brown, no).
thing(grass, green, no).
thing(lettuce, green, yes).
thing(orange, orange, yes).
thing(rice, white, yes).
thing(sky, blue, no).
```

What are orange foods?

```
?- thing(Name, orange, yes).
Name = carrot ;
Name = orange.
```

What things aren't foods?

```
?- thing(Name, _, no).
Name = dirt ;
Name = grass ;
Name = sky.
```

The underscore designates an anonymous variable. It indicates that any value matches and that we don't want to have the value associated with a variable (and thus displayed).

Alternate representation, continued

```
thing(apple, red, yes).
thing(broccoli, green, yes).
thing(carrot, orange, yes).
thing(dirt, brown, no).
thing(grass, green, no).
thing(lettuce, green, yes).
thing(orange, orange, yes).
thing(rice, white, yes).
thing(sky, blue, no).
```

What is green that is not a food?

```
?- thing(N,green,no).
N = grass ;
false.
```

What color is lettuce?

```
?- thing(lettuce,C,_).
C = green.
```

What foods are the same color as lettuce?

```
?- thing(lettuce,C,_), thing(N,C,yes), N \== lettuce.
C = green,
N = broccoli ;
false.
```

Is **thing/3** a better or worse representation of the knowledge than the combination of **food/1** and **color/2**?

Predicate/goal mismatches

Consider this knowledgebase:

```
x(just(testing,date(5,14,2014))).
x(10).
x(10,20).
```

The first fact's term is a structure but the second fact's term is a number. That inconsistency is not considered to be an error.

```
?- x(V).
V = just(testing, date(5, 14, 2014)) ;
V = 10.
```

Further, is it `x/1` or `x/2`?

```
?- x(A,B).
A = 10,
B = 20.
```

Predicate/goal mismatches, continued

At hand:

```
x(just(testing,date(5,14,2014))).
x(10).  x(A,B).
```

Here are some more queries:

```
?- x(abc).
false.

?- x([1,2,3]).   % A list...
false.

?- x(a(b)).
false.
```

The goals in the queries have terms that are an atom, a list, and a structure. There's no indication that those queries are fundamentally mismatched with respect to the terms in the facts.

Prolog says "**false**" in each case because nothing it knows about aligns with anything it's being queried about.

Predicate/goal mismatches, continued

At hand:

```
x(just(testing,date(5,14,2014))).
x(10).  x(A,B).
```

It's an error if there's no predicate defined that has the same number of terms as the goal in a query. Alternatives are suggested.

```
?- x(little,green,apples).
ERROR: Undefined procedure: x/3
ERROR:     However, there are definitions for:
ERROR:         x/1
ERROR:         x/2
```

What does the following tell us?

```
?- write(1,2).
ERROR: write/2: Domain error: `stream_or_alias' expected,
found `1'
```

Unification

== and \== are tests

Before talking about unification let's note that == and \== are tests. They are roughly equivalent to Haskell's == and /=, and Ruby's == and !=.

```
?- abc == 'abc'.
true.

?- 3 \== 5.
true.
```

Just like comparing tuples and lists in Haskell, and arrays in Ruby, structure comparisons in Prolog are "deep". Two structures are equal if they have the same functor, the same number of terms, and the terms are equal. (Recursive def'n.)

```
?- 3 + 4 == 4 + 3.
false.

?- abc(3 + 4 * 5) == abc(+(3,4*5)).
true.
```

Unification

The `=` operator, which we'll read as "unify" or "unify with", provides one way to do *unification*.

If a variable doesn't have a value it is said to be *uninstantiated*. At the start of a query all variables are uninstantiated.

If we unify an uninstantiated variable with a value, the variable is instantiated and unified with that value.

```
?- A = 10, write(A).
10
A = 10.
```

It can be read as "Unify `A` with 10 and write `A`."

That might look like assignment but **it is not assignment**!

Alternate reading: "Can you unify `A` with `10` and prove `write(A)`?"

Unification, continued

At hand:

```
?- A = 10, write(A).
10
A = 10.
```

An instantiated variable can be unified with a value only if the value equals (==) whatever value the variable is already unified with.

```
?- A = 10, write(A), A = 20, write(A).
10
false.
```

The unification of the uninstantiated `A` with `10` succeeds, and `write(A)` succeeds, but unification of `A` with `20` fails because `10 == 20` fails.

The query fails because its third goal, the unification `A = 20`, fails.

In essence the query is saying `A` must be 10 and `A` must be 20. Impossible!

Unification, continued

The lifetime of a variable is the query in which it is instantiated.

```
?- A = 10, B = 20, write(A), write(', '), write(B).
10, 20
A = 10,
B = 20.
```

If we use A, B, and (out of the blue) C in the next query, we find they are uninstantiated:

```
?- write(A), write(', '), write(B), write(', '), write(C).
_G1571, _G1575, _G1579
true.
```

Writing the value of an uninstantiated variable produces _G<NUMBER>.

Some say *bound variable* and *free variable* for instantiated and not.

Unification, continued

Consider the following:

```
?- A = B, C = 10, C = B, write(A).
10
A = B, B = C, C = 10.
```

The code above...

Unifies A with B (but both are still uninstantiated).

Unifies C (uninstantiated) with 10.

Unifies B with C.

Because A and B are already unified, and C is instantiated to 10.

Now A, B, and C have the value 10.

How will an initial instantiation for A affect the query?

```
?- A = 3, A = B, C = 10, C = B, write(A).
false.
```

Unification, continued

With uninstantiated (free) variables, unification has a behavior when unifying with values that resembles conventional assignment.

With instantiated (bound) variables, unification has a behavior when unifying with values that resembles comparison.

Unification of uninstantiated variables seems like aliasing of some sort.

But don't think of unification as assignment, comparison and aliasing rolled into one. Think of unification as a distinct new concept!

Another way to think about things:

Unification is not a question or an action, it is a demand!

`X = 3` is a goal that demands that `X` must be `3`. If not, the goal fails.

Yet another:

Unifications create constraints that Prolog upholds.

Unification with structures

Unification works with structures, too.

```
?- x(A, B) = x(10,20).
A = 10,
B = 20.

?- f(X, Y, Z) = f(just, testing,  f(a,b,c+d)).
X = just,
Y = testing,
Z = f(a, b, c+d).

?- f(X, Y, f(P1,P2,P3)) = f(just, testing,  f(a,b,c+d)).
X = just,
Y = testing,
P1 = a,
P2 = b,
P3 = c+d.
```

Unification with structures, continued

```
?- pair(A, A) = pair(3,5).
false.

?- pair(A, A) = pair(3,3).
A = 3.

?- lets(r,a,d,a,r) = lets(C1,C2,C3,C2,C1).
C1 = r,
C2 = a,
C3 = d.

?- f(X,20,Z) = f(10,Y,30),  New = f(Z,Y,X).
X = 10,
Z = 30,
Y = 20,
New = f(30, 20, 10).
```

Unification with structures, continued

Consider again this interaction:

```
?- food(F).
F = apple ;
F = broccoli ;
...
```

The query **food(F)** causes Prolog to search for facts that unify with **food(F)**.

Prolog is able to unify **food(apple)** with **food(F)**. It then shows that **F** is unified with **apple**.

When the user types semicolon, **F** is uninstantiated and the search for another fact to unify with **food(F)** resumes with the fact following **food(apple)**.

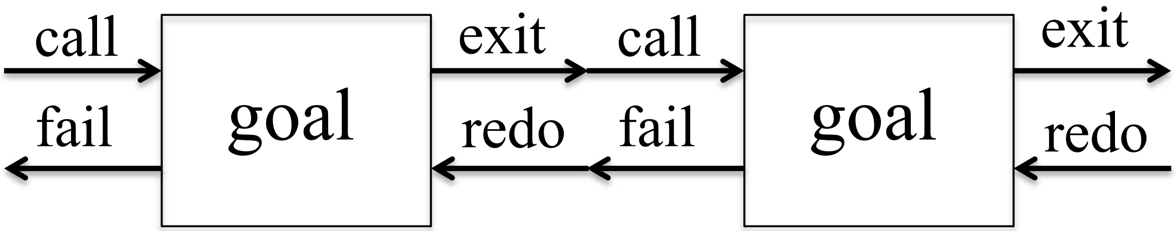
food(broccoli) is unified with **food(F)**, **F** is unified with **broccoli**, and the user is presented with **F = broccoli**.

The process continues until Prolog has found all the facts that can be unified with **food(F)** or the user is presented with a value for **F** that is satisfactory.

Query evaluation mechanics

Understanding query execution with the *port model*

Goals, like **food(fries)** or **color(What, Color)** can be thought of as having four *ports*:



In the *Active Prolog Tutor*, Dennis Merritt describes the ports in this way:

call: Using the current variable bindings, begin to search for the clauses which unify with the goal.

exit: Set a place marker at the clause which satisfied the goal. Update the variable table to reflect any new variable bindings. Pass control to the right.

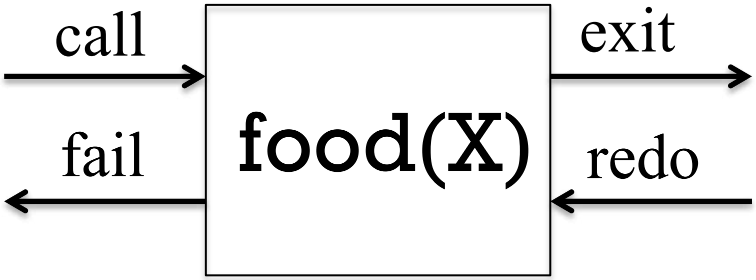
redo: Undo the updates to the variable table [that were made by this goal]. At the place marker, resume the search for a clause which unifies with the goal.

fail: No (more) clauses unify, pass control to the left.

The port model, continued

Example:

```
?- food(X).
X = apple ;
X = broccoli ;
X = carrot ;
X = lettuce ;
X = rice.

?-
```

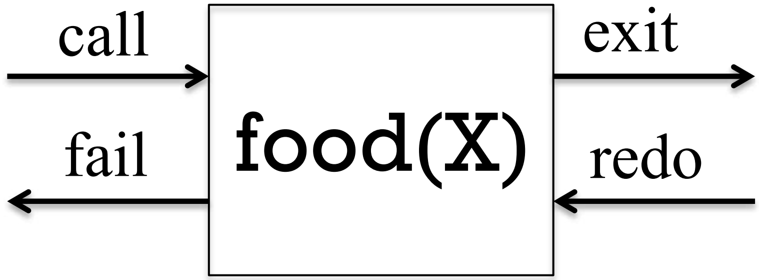


```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

The port model, continued

trace/0 activates "tracing" for a query.

```
?- trace, food(X).
   Call: (7) food(_G1571) ? creep
   Exit: (7) food(apple) ? creep
X = apple ;
   Redo: (7) food(_G1571) ? creep
   Exit: (7) food(broccoli) ? creep
X = broccoli ;
   Redo: (7) food(_G1571) ? creep
   Exit: (7) food(carrot) ? creep
```



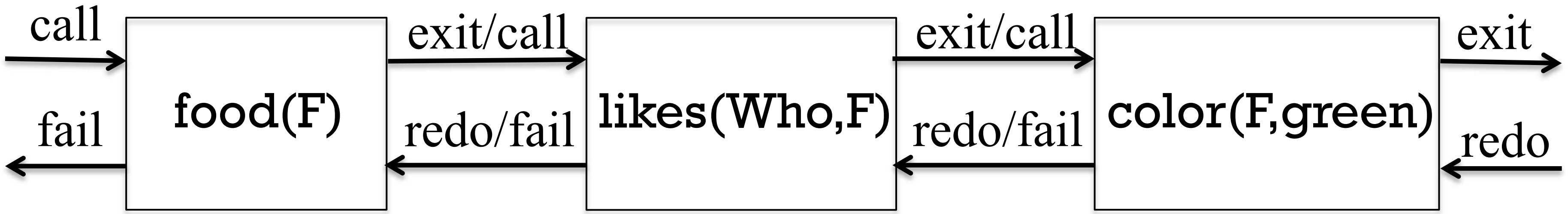
```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

Tracing shows the transitions through each port. The first transition is a call to the goal **food(X)**. The value shown, **_G1571**, stands for the uninstantiated variable **X**. We next see that goal being exited, with **X** instantiated to **apple**. The user isn't satisfied with the value, and by typing a semicolon forces the redo port to be entered, which causes **X**, previously bound to **apple**, to be uninstantiated. The next food fact, **food(broccoli)** is tried, instantiating **X** to **broccoli**, exiting the goal, and presenting **X = broccoli** to the user. (etc.)

The port model, continued

Who likes green foods?

```
?- food(F), likes(Who,F), color(F,green).
```



call

fail

food(F)

exit/call

redo/fail

likes(Who,F)

exit/call

redo/fail

color(F,green)

exit

redo

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(orange).
food(rice).
```

```
likes(bob, carrot).
likes(bob, apple).
likes(joe, lettuce).
likes(mary, broccoli).
likes(mary, tomato).
likes(bob, mary).
likes(mary, joe).
likes(joe, baseball).
likes(mary, baseball).
likes(jim, baseball).
```

```
color(sky, blue).
color(dirt, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(rice, white).
```

Next: Trace it!

Producing output

We've seen that **write/1** always succeeds and, as a side effect, outputs the term it is called with.

```
?- write(apple), write(' '), write(pie).
apple pie
true.
```

writeln/1 is similar, but appends a newline.

```
?- writeln(apple), writeln(pie).
apple
pie
true.
```

nl/0 outputs a newline. (Note the blank lines before and after **middl**e.)

```
?- nl, writeln(middle), nl.

middle

true.
```

Producing output, continued

The predicate **format/2** is conceptually like **printf** in Ruby, C, and others.

```
?- format('x = ~w\n', 101).
x = 101
true.
```

~w is one of many format specifiers. The "**w**" indicates to output the value using **write/1**. Use **help(format/2)** to see all the specifiers. (Don't forget the /2!)

If more than one value is to be output, the values must be in a list.

```
?- format('label = ~w, value = ~w, x = ~w\n', ['abc', 10, 3+4]).
label = abc, value = 10, x = 3+4
true.
```

We'll see more on lists later but for now note that we make a list by enclosing zero or more terms in square brackets. Lists are heterogeneous, like Ruby arrays.

Producing output, continued

A first attempt to print all the foods:

```
?- food(F), format('~w is a food\n', F).
apple is a food
F = apple ;
broccoli is a food
F = broccoli ;
carrot is a food
F = carrot ;
...
```

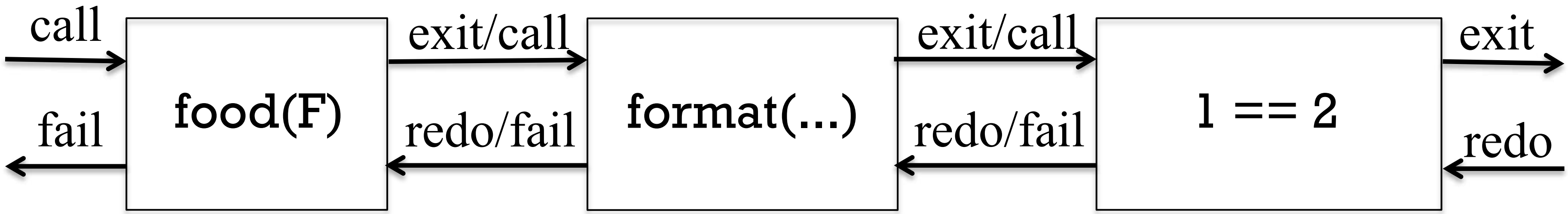
Ick—we have to type semicolons to cycle through them!

Any ideas?

Producing output, continued

Second attempt: Force alternatives by specifying a goal that always fails.

```
?- food(F), format('~w is a food\n', F), 1 == 2.
apple is a food
broccoli is a food
carrot is a food
...
```

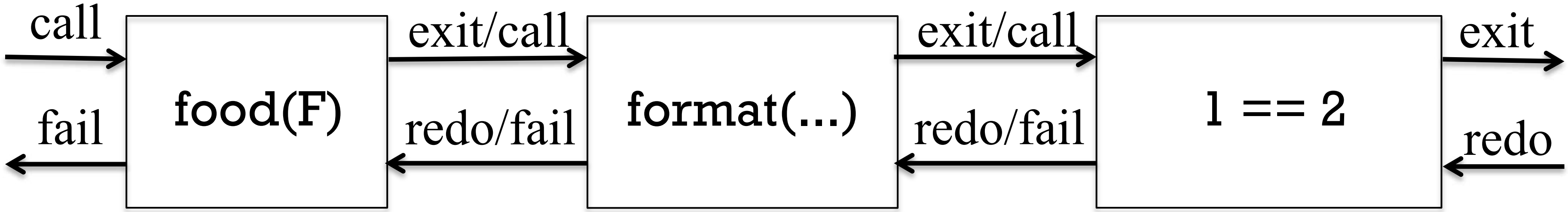


This query is a loop! **food(F)** unifies with the first **food** fact and instantiates **F** to its term, the atom **apple**. Then **format** is called, printing a string with the value of **F** interpolated. **1 == 2** always fails. Control then moves left, into the redo port of **format**. **format** doesn't erase the output but it doesn't have an alternatives either, so it fails, causing the redo port of **food(F)** to be entered. **F** is uninstantiated and **food(F)** is unified with the next **food** fact in turn, instantiating **F** to **broccoli**. The process continues, with control repeatedly moving back and forth until all the **food** facts have been tried.

Backtracking

At hand:

```
?- food(F),  format('~w is a food\n', F),  1 == 2.
apple is a food
broccoli is a food
...
```



The activity of moving leftwards through the goals is known as *backtracking*.

We might say, "The query gets a food **F**, prints it, fails, and then *backtracks* to try the next food."

Prolog does <u>not</u> analyze things far enough to recognize that it will never be able to "prove" what we're asking. Instead it goes through the motions of trying to prove it and as side-effect, we get the output we want. <u>This is a key idiom of Prolog programming.</u>

Backtracking, continued

At hand:

```
?- food(F),  format('~w is a food\n', F),  1 == 2.
apple is a food
broccoli is a food
...
false.
```

Predicates respond to "redo" in various ways. With only a collection of facts for `food/1`, redo amounts to advancing to the next fact, if any. If there is one, the goal exits (control goes to the right). If not, it fails (control goes to the left).

Some other possible examples of "redo" behavior:

A sequence of redos might cause a predicate to work through a series of URLs to find a current data source.

A geometry manager might force a collection of windows to produce a configuration that is mutually acceptable.

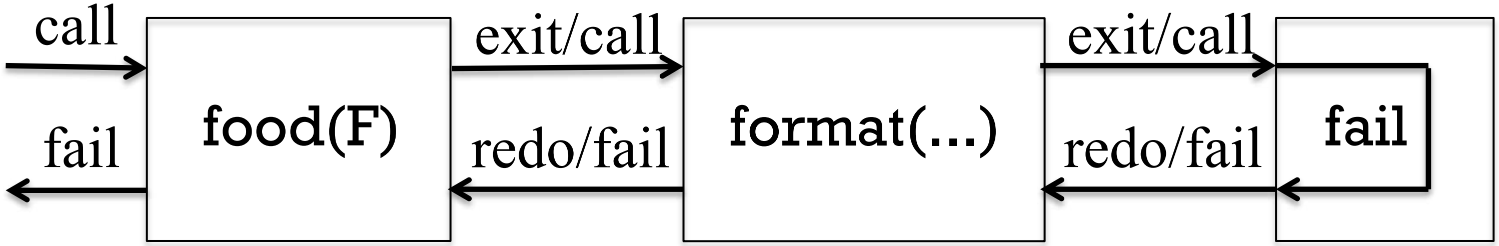
A predicate might create a file when called and delete it on redo.

The predicate **fail**

The predicate **fail/0** always fails. It's important to understand that an always-failing goal like **1 == 2** produces exhaustive backtracking but in practice we'd use **fail** instead:

```
?- food(F), format('~w is a food\n', F), fail.
apple is a food
broccoli is a food
...
rice is a food
false.
```

In terms of the four-port model, think of **fail** as a box whose call port is "wired" to its fail port:



Sidebar: **between**

The built-in predicate **between/3** can be used to instantiate a variable to a sequence of integer values:

```
?- between(1,3,X).
X = 1 ;
X = 2 ;
X = 3.
```

Problem: Print this sequence:

```
000
001
010
011
100
101
110
111
```

```
?- between(0,1,A),between(0,1,B),between(0,1,C),
     format('~w~w~w\n', [A,B,C]), fail.
```

How about this one?

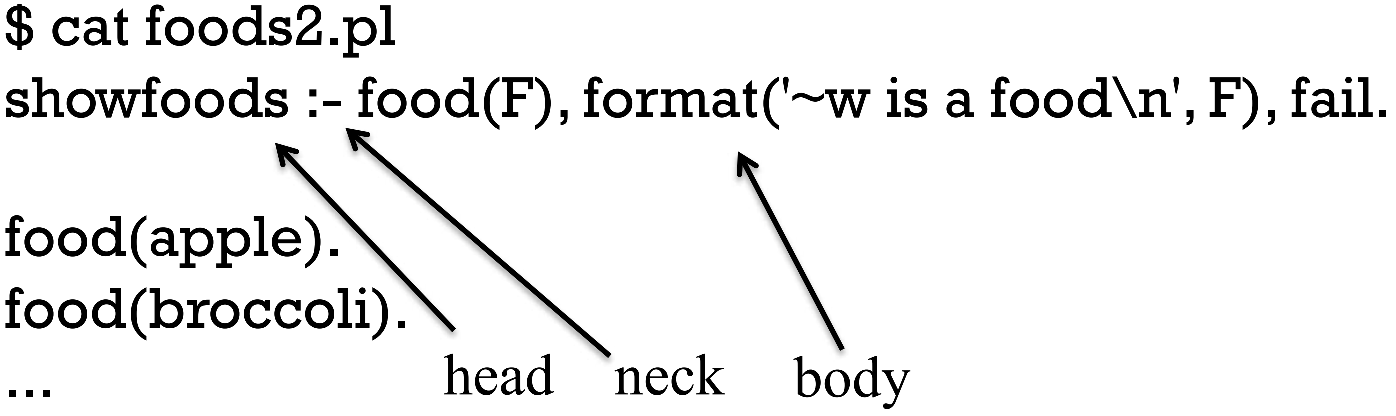
```
10101000
10101001
10101010
10101011
10111000
10111001
10111010
10111011
```

Rules

showfoods: a simple *rule*

Facts are one type of Prolog *clause*. The other type of clause is a *rule*.

foods2.pl starts with a rule and is followed by the food facts:



```
$ cat foods2.pl
showfoods :- food(F), format('~w is a food\n', F), fail.

food(apple).
food(broccoli).
...
```

Even though **showfoods/0** uses **food/1**, it can either precede or follow the clauses (the procedure) for that predicate.

showfoods, continued

At hand:

```
$ cat foods2.pl
showfoods :- food(F), format('~w is a food\n', F), fail.

food(apple).
food(broccoli).
...
```

Usage:

```
$ swipl foods2.pl
...
?- showfoods.
apple is a food
broccoli is a food
carrot is a food
lettuce is a food
orange is a food
rice is a food
false.
```

Sidebar: Horn Clauses

Prolog borrows from the idea of *Horn Clauses* in symbolic logic. A simplified definition of a Horn Clause is that it represents logic like this:

If Q_1, Q_2, Q_3, ..., Q_n, are all true, then P is true.

In Prolog we might represent a three-element Horn clause with this rule:

```
p :- q1, q2, q3.
```

The query

```
?- p.
```

which asks Prolog to "prove" **p**, causes Prolog to try and prove **q1**, then **q2**, and then **q3**. If it can prove all three, and can therefore prove **p**, Prolog will respond with **true.** (If not, then **false.**)

Note that this is an abstract example—we haven't defined the predicates **q1/0** et al.

showfoods, continued

At hand are the following rules:

```
p :- q1, q2, q3.

showfoods :- food(F), format('~w is a food\n', F), fail.
```

We saw that we can print all the foods with this query:

```
?- showfoods.
apple is a food
broccoli is a food
carrot is a food
...
rice is a food
false.
```

In its unsuccessful attempt to "prove" **showfoods**, and thus trying to prove all three goals in the body of the **showfoods** rule, Prolog ends up doing what we want: all the foods are printed.

We send Prolog on a wild goose chase to get our work done!

showfoods, continued

Let's print a header and footer around the foods:

```
?- writeln('-- Foods --'), showfoods, writeln('-- end --').
-- Foods --
apple is a food
broccoli is a food
carrot is a food
lettuce is a food
orange is a food
rice is a food
false.
```

What's wrong?

showfoods, continued

At hand:

```
?- writeln('-- Foods --'), showfoods, writeln('-- end --').
-- Foods --
apple is a food
...
rice is a food
false.
```

(No **--end--** *line!)*

Why does Prolog say **false.** after printing the foods?

Recall **showfoods** and note that it can never succeed. It always fails!

```
showfoods :- food(F), format('~w is a food\n', F), fail.
```

Because the second goal in

```
?- writeln('-- Foods --'), showfoods, writeln('-- end --').
```

always fails, we never reach the third goal.

showfoods, continued

The problem: We're using a **fail** to force display of all the foods but we want **showfoods** to ultimately succeed. Any ideas?

```
showfoods :- food(F), format('~w is a food\n', F), fail.
showfoods.
```

Result:

```
?- showfoods.
apple is a food
broccoli is a food
...
rice is a food
true.
```
% ***<u>Very important</u>***: *Now we get* **true**.*, not* ***false***.

Prolog tries the two clauses for the predicate **showfoods** in turn. The first clause, a rule, is ultimately a failure but prints the foods as a side-effect. Because the first clause fails, Prolog tries the second clause, a fact which is trivially proven.

showfoods, continued

At hand:

```
showfoods :- food(F), format('~w is a food\n', F), fail.
showfoods.

?- showfoods.
apple is a food
broccoli is a food
...
true.    % Very important: Now it says true., not false.
```

The underlying rule:

If a clause fails, Prolog tries the predicate's next clause. It continues until a clause succeeds or no clauses remain.

This is the same mechanism we've been using to go through foods, colors, and more.

showfoods2

Here's a variation:

```
showfoods2 :- writeln('-- Foods --'), food(F), writeln(F), fail.
showfoods2 :- writeln('-- end --').
```

How does it behave?

```
?- showfoods2.
-- Foods --
apple
broccoli
carrot
lettuce
orange
rice
-- end --
true.
```

Sidebar: goals don't nest!

Unlike expressions in conventional languages, Prolog goals don't nest.

Here's a failed attempt at nesting:

```
showfoods3 :- writeln('-- Foods --'), writeln(food(F)), fail.
showfoods3 :- writeln('-- end --').
```

How will it behave?

```
?- showfoods3.
-- Foods --
food(_G1183)
-- end --
true.
```

What's happening?

We're calling `writeln` with a one-term structure, `food`, whose term is the uninstantiated variable `F`.

Rules with arguments

This one-rule predicate asks if there is a food with a particular color:

```
food_color(Color) :- food(F), color(F,Color).  % in foods2.pl
```

Usage:

```
?- food_color(green).
true
```

To prove the goal **food_color(green)**, Prolog first searches its clauses for one that can be unified with the goal. It finds a rule (above) whose head can be unified with the goal. That unification causes **Color** to be instantiated to the atom **green**.

It then attempts to prove **food(F)**, and **color(F, green)** for some value **F**.

The response **true** tells us that at least one green food exists, but that's all we know.

Rules with arguments, continued

At hand:

```
food_color(Color) :- food(F), color(F,Color).
```

The last slide didn't tell the whole truth. The cursor pauses right after **true**:

```
?- food_color(green).
true _ (blink...blink...blink)
```

If we type semicolons we see this:

```
?- food_color(green).
true ;
true ;
false.
```

It reveals that **food_color(green)** is actually finding two green foods but we don't know what they are.

A failure:

```
?- food_color(blue).
false.
```

Rules with arguments, continued

At hand:

```
food_color(Color) :- food(F), color(F,Color).
```

Does **food_color** let us do anything other than asking if there is a food with a particular color?

We can ask for all the colors of foods.

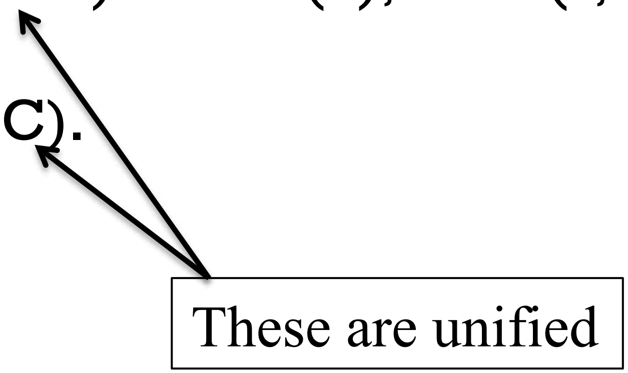
```
?- food_color(C).
C = red ;
C = green ;
C = orange ;
C = green ;
C = white.
```

We get **green** twice because there are two green foods. We'll later see ways to deal with that.

Rules with arguments, continued

At hand:

```
food_color(Color) :- food(F), color(F,Color).

?- food_color(C).
C = red ;
C = green ;
```



These are unified

A very important rule:

If a fact or the head of a rule in the knowledgebase unifies with a query, any corresponding variables are unified. (Instantiating one instantiates the other.)

In the above case the variable **C** first has the value **red** because:

C in the query was unified with **Color** in the head of the rule,
AND the goals in the body of the rule succeeded,
AND **Color** was instantiated to **red**.

When we type a semicolon in response to **C = red**, Prolog backtracks and ultimately comes up with **C = green**.

Instantiation as "return"

At hand:

```
food_color(Color) :- food(F), color(F,Color).
```

Prolog has no analog for "**return x**"! There's no way to say something like this,

```
?- Color = food_color(), writeln(Color), fail.
```

or this,

```
?- writeln(food_color()), fail.
```

Instead, predicates "return" values by instantiating logical variables.

```
?- food_color(C), writeln(C), fail.
red
green
...
```

Instantiation as "return", continued

Some examples of instantiation as "return" with built-in predicates:

```
?- atom_length(testing, Len).
Len = 7.

?- upcase_atom(testing, Caps).
Caps = 'TESTING'.

?- char_type('A', T).
T = alnum ;
T = alpha ;
...
T = upper(a) ;
...
T = xdigit(10).
```

Instantiation as "return", continued

Some two-term built-in predicates will fill in whichever term is uninstantiated.

```
?- term_to_atom(date(10,1,1891), A).
A = 'date(10,1,1891)'.

?- term_to_atom(T, 'date(10,1,1891)').
T = date(10, 1, 1891).
```

The following call specifies a **date** structure with uninstantiated variables for its terms.

```
?- term_to_atom(date(M,D,Y), 'date(10,1,1891)').
M = 10,
D = 1,
Y = 1891.
```

Instantiation as "return", continued

In Prolog, what is **ten-four**?

A two-term structure with the functor '**-**'. Its terms are the atoms **ten** and **four**. **display(ten-four)** shows **-(ten,four)**.

Consider this predicate:

```
swap_struct(X-Y, R) :- R = Y-X.
```

Usage:

```
?- swap_struct(ten-four, X).
X = four-ten.

?- swap_struct(X, 10-20).
X = 20-10.
```

Can **swap_struct** be simplified?

```
swap_struct(X-Y, Y-X).
```

A little more...

```
?- swap_struct(a-b*c,R).
R = b*c-a.

?- swap_struct(a-b+c,R).
false.

?- swap_struct(a-b-c,R).
R = c- (a-b).
```

Instantiation as "return", continued

Problem: Using **term_to_atom** write a predicate with this behavior:

```
?- swap('ten-four', R).
R = 'four-ten'.
```

First cut:

```
swap(A, Result) :-
    term_to_atom(T,A),
    First-Second = T,  Swapped = Second-First,
    term_to_atom(Swapped, Result).
```

How many variables are used in **swap/2** above?

Better:

```
swap2(A, Result) :-
    term_to_atom(First-Second, A),
    term_to_atom(Second-First, Result).
```

Instantiation as "return", continued

Problem: Write a predicate with these four behaviors:

```
?- describe_food(apple-X).
X = red.

?- describe_food(X-green).
X = broccoli ;
X = lettuce ;
false.

?- describe_food(X).
X = apple-red ;
X = broccoli-green ;
...
X = orange-orange ;
X = rice-white.
```

The fourth:

```
?- describe_food(apple-red).
true.

?- describe_food(apple-blue).
false.
```

Solution:

```
describe_food(Food-Color) :- food(Food), color(Food,Color).
```

Sidebar: Describing predicates

Recall **between(1,10,X).** Here's what **help(between)** shows:

```
between(+Low, +High, ?Value)
   Low  and High are  integers, High >= Low.   If Value is an  integer,
   Low =< Value =< High. When Value is a variable it is successively
   bound to all integers between Low and  High. ...
```

If an argument has a plus prefix, like **+Low** and **+High**, it means that the argument is an input to the predicate and must be instantiated. A question mark indicates that the argument can be input or output, and thus may or may not be instantiated.

The documentation implies that **between** can (1) generate values and (2) test for membership in a range.

```
?- between(1,10,X).
X = 1 ;
...

?- between(1,10,5).
true.
```

Note: This is a documentation convention; do not use the **+** and **?** symbols in code!

Describing predicates, continued

Another:

term_to_atom(?Term, ?Atom)

True if Atom describes a term that unifies with Term. When Atom is instantiated, Atom is converted and then unified with Term. ...

Here is a successor predicate:

succ(?Int1, ?Int2)

True if Int2= Int1+1 and Int1>=0. At least one of the arguments must be instantiated to a natural number. ...

```
?- succ(10,N).
N = 11.
```

There's no **pred** (predecessor) predicate. Why?

```
?- succ(N,10).
N = 9.
```

Describing predicates, continued

Here is the synopsis for **format/2**:

```
format(+Format, +Arguments)
```

Speculate: What does **sformat/3** do?

```
sformat(-String, +Format, +Arguments)
```

The minus in **-String** indicates that the term should be an uninstantiated variable.

```
?- sformat(S, 'x = ~w', 1).
S = "x = 1".

?- sformat("x = 1", 'x = ~w', 1).
false.
```

Next set of slides

Arithmetic

Why are there no arithmetic predicates?

We've seen that there are predicates for comparisons but not for arithmetic operations:

```
?- 3 == 4.
false.

?- 3 \== 4.
true.

?- 3 + 4.
ERROR: toplevel: Undefined procedure: (+)/2 (DWIM could
not correct goal)
```

Why is this the case?

Queries succeed or fail. The result of a comparison can be viewed as success or failure but there's simply no place for the result of 3 + 4 to appear. (There's no "outlet" for it.)

is/2

The predicate **is(?Value, +Expr)** evaluates **Expr**, a structure representing an arithmetic expression, and unifies the result with **Value**.

```
?- is(X, 3+4*5).
X = 23.
```

is is an operator and can be used in infix form:

```
?- X is 3 + 4, Y is 7 * 5, Z is X / Y.
X = 7,
Y = 35,
Z = 0.2.
```

All variables in the structure being evaluated by **is/2** must be instantiated.

```
?- A is 3 + X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

(The query **?- ground(3+X).** fails—the term **3+X** has free variables.)

Arithmetic, continued

is supports a number of arithmetic operations. Here are some of them:

-X	negation
X + Y	addition
X - Y	subtraction
X * Y	multiplication
X / Y	division—produces float quotient
X // Y	integer division
X rem Y	integer remainder
integer(X)	truncation to integer
float(X)	conversion to float
sign(X)	sign of **X**: -1, 0, or 1

help(rem) is a quick way to open up the documentation section with the arithmetic operations.
help(op) shows precedence.

```
?- X is 77777777777777777777777*3333333333333333333333333.
X = 259259259259259259259256640740740740740740740741.

?- X is 10 // 3.
X = 3.

?- X is e ** sin(pi).
X = 1.0000000000000002.
```

What are **e** *and* **pi***? Is* **sin** *a Prolog"function"?*

Arithmetic, continued

Problem: Write a predicate **around/3** that works like this:

```
?- around(P ,7, N).
P = 6,
N = 8.
```

Solution:

```
around(Prev,X,Next) :- Prev is X - 1, Next is X + 1.
```

We can use **around** to test, too, but the second term must be "ground".

```
?- around(1,2,3).
true.

?- around(1,X,3).
ERROR: is/2: Arguments are not sufficiently instantiated
```

Arithmetic, continued

Here are some predicates to compute the area of shapes. Note the use of unification to "label" the structure's term(s).

```
    area(rectangle(W,H), A) :- A is W * H.
    area(circle(R), A) :- A is pi * R ** 2.
```

Usage:

```
    ?- area(circle(3), A).
    A = 28.274333882308138.

    ?- area(rectangle(5,7), A).
    A = 35.
```

A "figure 8" is two circles touching at a point. Let's handle it by making two circles and computing the sum of their areas:

```
    area(figure8(R1,R2), A) :-
        area(circle(R1),A1), area(circle(R2),A2), A is A1 + A2.
```

Arithmetic, continued

For reference:

```
area(rectangle(W,H), A) :- A is W * H.
```

Note the "expressions" in the **rectangle** below. How do they work?

```
?- area(rectangle(2*3, 4*5), A).
A = 120.
```

Would **?- area(rectangle(1+2,3+4),A).** have precedence issues?

Experiment:

```
?- W = 1+2, H = 3+4, A = W * H, display(A), R is A.
*(+(1,2),+(3,4))
W = 1+2,
H = 3+4,
A = (1+2)* (3+4),
R = 21.
```

Arithmetic, continued

Here's a predicate that computes the length of a line between two points:

```
length(point(X1,Y1), point(X2,Y2), Length) :-
    Length is sqrt((X1-X2)**2+(Y1-Y2)**2).
```

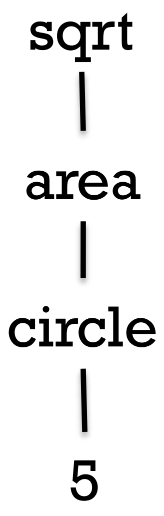
Note that the **sqrt** "function call" is just another structure whose functor is known to **is/2**.

Note also that **sqrt**'s "argument" is a structure that will be evaluated by **is**.

```
?- length(point(3,0),point(0,4),Len).
Len = 5.0.
```

Recalling **area** from earlier, will the following work?

```
?- X is sqrt(area(circle(5))).
ERROR: is/2: Arithmetic: `circle/1' is not a function
```



Comparisons

There are several numeric comparison operators.

X =:= Y	numeric equality
X =\= Y	numeric inequality
X < Y	numeric less than
X > Y	numeric greater than
X =< Y	numeric equal or less than (NOTE the order, not <= !)
X >= Y	numeric greater than or equal

Just like is/2, these operators evaluate their operands. Examples of usage:

```
?- 3 + 5 =:= 2*3+2.
true.

?- X is 3 / 5, X > X*X.
X = 0.6.

?- X is random(10), X > 5.
false.

?- X is random(10), X > 5.
X = 9.
```

Note that the comparisons produce no value; they simply succeed or fail.

Example: Grade computation (and "cut")

Example: grade computation

Here is **grade(+Score, ?Grade)**:

```
grade(Score, 'A') :- Score >= 90.
grade(Score, 'B')  :- Score >= 80, Score < 90.
grade(Score, 'C') :- Score >= 70, Score < 80.
grade(Score, 'F')  :- Score < 70.
```

Usage:

```
?- grade(95,G).
G = 'A' ;                (user entered semicolon)
false.

?- grade(82,G).
G = 'B' ;                (user entered semicolon)
false.

?- grade(50,G).
G = 'F'.                 (swipl printed period)
```

Why did the first two prompt the user?

There were still untried clauses for **grade/2**.

Grade computation, continued

Here are some student facts:

```
student('Ali', 85).
student('Chris',92).
student('Kendall', 89).
```

Problem: write **grades/0**, which behaves like this:

```
?- grades.
Current Grades
 Ali: B
 Chris: A
 Kendall: B
true.
```

Recall **grade/2**:

```
  ?- grade(95,G).
  G = 'A'
```

Solution:

```
grades :- writeln('Current Grades'),
   student(Student,Score), grade(Score,Grade),
   format(' ~w: ~w\n', [Student, Grade]),
   fail.
grades.
```

Grade computation, continued

Here's a new version of **grade/2**:

```
grade(Score, 'A') :- Score >= 90.
grade(Score, 'B') :- Score >= 80.
grade(Score, 'C') :- Score >= 70.
grade(_, 'F').          % Note use of underscore for "don't care".
```

Let's try **grades/0** again:

```
?- grades.
Current Grades
 Ali: B
 Ali: C
 Ali: F
 Chris: A
 Chris: B
 Chris: C
 Chris: F
 Kendall: B
 Kendall: C
 Kendall: F
true.
```

Here is **grades/0**:

```
grades :- writeln('Current Grades'),
    student(Student,Score), grade(Score,Grade),
    format(' ~w: ~w\n', [Student, Grade]), fail.
grades.
```

The old **grade/2**:

```
grade(Score, 'A') :- Score >= 90.
grade(Score, 'B') :- Score >= 80, Score < 90.
grade(Score, 'C') :- Score >= 70, Score < 80.
grade(Score, 'F') :- Score < 70.
```

What's wrong? →

The fail in **grades** is driving **grade** to try subsequent rules. Ali's 85 satisfies the last three rules in the new version of **grade/2**! Chris' 92 satisfies all four!

"Cut"

The predicate ! is "cut". It's just an exclamation mark.

Cut is a *control predicate*, like **fail/0**. It affects the flow of control.

When a cut is encountered in a rule it means,

> "If you get to here, you have picked the right rule to produce a final answer for this call of this predicate."

We can fix **grade/2** with some cuts:

```
grade(Score, 'A') :- Score >= 90, !.
grade(Score, 'B') :- Score >= 80, !.
grade(Score, 'C') :- Score >= 70, !.
grade(_, 'F').
```

```
?- grades.
Current Grades
 Ali: B
 Chris: A
 Kendall: B
true.
```

The rule **grade(Score, 'A') :- Score >= 90, !.** says,

> If score >= 90 then the grade is an "A", and that's my final answer.

Cut, continued

How does the behavior change if we do the cut first instead of last?

```
grade(Score, 'A') :- !, Score >= 90.
grade(Score, 'B') :- !, Score >= 80.
grade(Score, 'C') :- !, Score >= 70.
grade(_, 'F').
```

```
student('Ali', 85).
student('Chris',92).
student('Kendall', 89).
```

```
grades :- writeln('Current Grades'),
   student(Student,Score),
   grade(Score,Grade),
   format(' ~w: ~w\n', [Student, Grade]),
   fail.
grades.
```

Execution:

```
?- grades.
Current Grades
 Chris: A
true.
```

Why?

For Ali, **grade(85,Grade)** is called and **grade(Score, 'A') :- !, Score >= 90.** is executed. The cut is done first thing, commiting this rule to producing the final answer for **grade(85, Grade)**. It then fails on **Score >= 90**.

grades then backtracks and continues with the next student, Chris.

Sidebar: The "color" of a cut

A cut is said to be a "green cut" if it simply makes a predicate more efficient. By definition, adding or removing a green cut does not effect the set of results for any call of a predicate.

A "red cut" affects the set of results produced by a predicate.

Are the cuts on slide 111 green cuts or red cuts?

Red. For example, `grade(90,G)` produces one result, not four.

Below is the first version of `grade` with some cuts added. Are they red cuts or green cuts?

```
grade(Score, 'A') :- Score >= 90, !.
grade(Score, 'B')  :- Score >= 80, Score < 90, !.
grade(Score, 'C') :- Score >= 70, Score < 80, !.
grade(Score, 'F')  :- Score < 70.
```

There are also blue and "grue" cuts.

Cut, continued

Here's one way to write **max**: (Adapted from *Clause and Effect* by Clocksin)

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

Usage:

```
?- max(10,3,Max).
Max = 10 ;          (Prolog pauses because of a possible alternative.)
false.
```

Can we shorten it with a cut?

```
max(X, Y, X) :- X >= Y, !.
max(_, Y, Y).
```

Usage:

```
?- max(10,3,Max).
Max = 10.           (Prolog prints period because no alternatives.)
```

Sidebar: Clauses vs **if-else**

Here's that first version of **max**, with a Ruby analog beside it

max(X, Y, X) :- X >= Y.	if X >= Y then X end
max(X, Y, Y) :- X < Y.	if X < Y then Y end

Here's the version of **max** with a cut, also with a Ruby analog:

max(X, Y, X) :- X >= Y, !.	if X >= Y then X
max(_, Y, Y).	else Y end

Cut, continued

Cuts can be used to limit backtracking inside a query or rule.

Consider these facts:

```
f(' f1 ').    f(' f2 ').     f(' f3 ').
g(' g1 ').  g(' g2 ').   g(' g3 ').
```

Queries and cuts:

```
?- f(F), write(F), g(G), write(G), fail.
 f1  g1  g2  g3  f2  g1  g2  g3  f3  g1  g2  g3

?- f(F), write(F), !, g(G), write(G), fail.
 f1  g1  g2  g3

?- f(F), write(F), g(G), write(G), !, fail.
 f1  g1
```

Another analogy: A cut is like a door that locks behind you.

There is far more to know about "cut" but for now we'll use it for only one thing:

> "If you get to here, this rule will produce a final answer for this call to this predicate."

The "cut-fail" idiom

Predicates naturally fail when a desired condition is absent but sometimes we want a predicate to fail when a particular condition **is** present.

I'll consider a person to be "ok" if they're a bowler, they like BBQ, or they'll loan me money.

But no matter what, if a person went to UNC-CH, they're not "ok"!

We can express that with cut-fail:

```
    ok(Person) :- alma_mater(Person, 'UNC-CH'), !, fail.
    ok(Person) :- bowler(Person).
    ok(Person) :- likes(Person, bbq).
    ok(Person) :- will_loan_me_money(Person).
```

A cut says, "This is my final answer."

A cut-fail says, "My final answer is no!" (False, in SWI Prolog.)

A tax collection example with "cut-fail"

```
average_taxpayer(X) :-
   foreigner(X), !, fail.
```

A person is not an average taxpayer if they are a foreigner.

```
average_taxpayer(X) :-
   spouse(X,Spouse),
   gross_income(Spouse, SpouseIncome),
   SpouseIncome > 3000, !, fail.
```

A person is not an average taxpayer if they've got a spouse and the spouse makes over 3000.

```
average_taxpayer(X) :-
   gross_income(X, Inc),
   Inc > 2000, Inc =< 20_000.
```

A person <u>is</u> an average taxpayer if their income is between 2000 and 20,000.

```
gross_income(X,GrossIncome) :-
   receives_pension(X, GrossIncome),
   GrossIncome < 5000, !, fail.
```

A person is not considered to have a gross income if they receive a pension of less than 5000.

```
gross_income(X, GrossIncome) :-
   gross_salary(X, GrossSalary),
   investment_income(X,InvestmentIncome),
   GrossIncome is GrossSalary + InvestmentIncome.

investment_income(X, InvestmentIncome) :- ...
```

Example straight from C&M, page 91.

The "singleton" warning(!)

The "singleton" warning

Here's a predicate add(+X,+Y, ?Sum):

```
$ cat add.pl
add(X, Y, Sum) :- S is X + Y.
```

Bug: **Sum** is used in the head but **S** is used in the body!

Observe what happens when we load it:

```
$ swipl add.pl
Warning: /cs/www/classes/cs372/spring16/prolog/add.pl:1:
        Singleton variables: [Sum,S]
...
```

What is Prolog telling us with that warning?

The variables **Sum** and **S** appear only once in the rule on line 1.

Fact: If a variable appears only once in a rule, its value is never used.

A singleton warning may indicate a misspelled or misnamed variable.

Pay attention to singleton warnings!

Singletons, continued

print_stars(+N) prints N asterisks:

```
?- print_stars(10).
**********
true.
```

Here's a first version of it. Does it have any singletons?

```
print_stars(N) :- between(1,N,X), write('*'), fail.
print_stars(N).
```

Let's see...

```
$ swipl print_stars.pl
Warning: print_stars.pl:1: ... Singleton variables: [X]
Warning: print_stars.pl:2: ... Singleton variables: [N]
...
```

Should we worry about the warnings? How could we eliminate them?

```
print_stars(N) :- between(1,N,_), write('*'), fail.
print_stars(_).
```

Singleton warnings are easy to overlook!

Note that singleton warnings appear **before** "Welcome to SWI-Prolog"!

```
$ swipl print_stars.pl          (first version)
Warning: /cs/www/classes/cs372/spring16/prolog/print_stars.pl:1:
        Singleton variables: [X]
Warning: /cs/www/classes/cs372/spring16/prolog/print_stars.pl:2:
        Singleton variables: [N]
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
software, and you are welcome to redistribute it under certain
conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

In fact, all errors found when consulting a file appear before the welcome.

"Can't prove"

"can't prove"

The query \+*goal* succeeds if *goal* fails.

```
?- food(computer).
false.

?- \+food(computer).
true.
```

An incomplete set of facts can produce oddities.

```
?- \+food(cake).
true.
```

\+ is often spoken as "can't prove" or "fail if".

"can't prove", continued

Example: *What foods are not green?*

```
?- food(F), \+color(F,green).
F = apple ;
F = carrot ;
F = orange ;
F = rice ;
F = 'Big Mac'.
```

If there's no color fact for a food, will the query above list that food?

How can we see if there are any foods don't have a **color** fact?

```
?- food(F), \+color(F,_).
F = 'Big Mac'.
```

"can't prove", continued

Describe the behavior of **inedible/1**:

```
inedible(X) :- \+food(X).
```

inedible(X) succeeds if something is <u>not known</u> to be a food.

```
?- inedible(rock).
true.
```

What will the query **?- inedible(X).** do?

```
?- inedible(X).
false.
```

"can't prove", continued

What's the following query asking?

```
?- color(X,_), \+food(X).
X = sky ;
X = dirt ;
X = grass ;
false.
```

What are things with known colors that aren't food?

Let's try reversing the goals:

```
?- \+food(X), color(X,_).
false.
```

Why do the results differ?

"can't prove", continued

Important: variables are never instantiated by a "can't prove" goal.

Example:

```
?- \+food(X).
false.
```

Consider this attempt to ask for things that aren't purple.

```
?- \+color(Thing, purple).
true.
```

There are many such things but **Thing** is not instantiated.

"can't prove" with cut-fail

Here's how we could implement **\+** (can't prove) using the higher-order predicate **call/1** and a cut-fail:

```
cant_prove(G) :- call(G), !, fail.
cant_prove(_).
```

Usage:

```
?- cant_prove(food(apple)).
false.

?- cant_prove(food(computer)).
true.

?- cant_prove(color(_,purple)).
true.
```

Is **cant_prove** a higher-order predicate?

More with rules

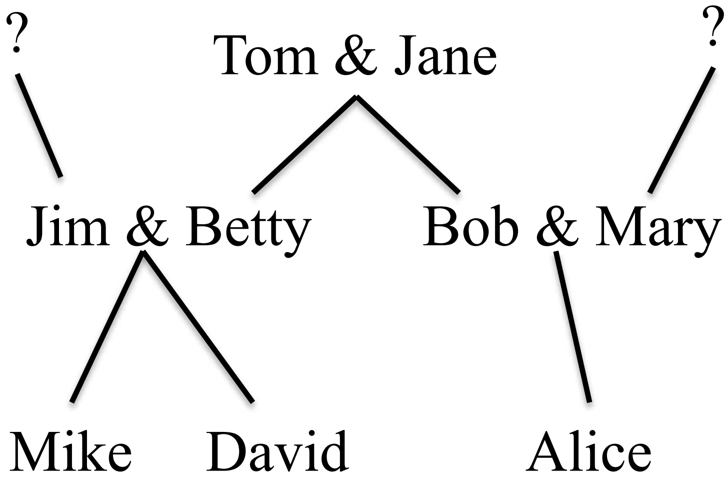
Parents and children

Here is a set of facts for parents and children:

```
male(tom).
male(jim).
male(bob).
male(mike).
male(david).

female(jane).
female(betty).
female(mary).
female(alice).
```

```
parent(tom,betty).
parent(tom,bob).
parent(jane,betty).
parent(jane,bob).
parent(jim,mike).
parent(jim,david).
parent(betty,mike).
parent(betty,david).
parent(bob,alice).
parent(mary,alice).
```

(parents.pl)



Define a rule for **father(F,C)**.

```
 father(F,C) :-
      male(F), parent(F,C).
```

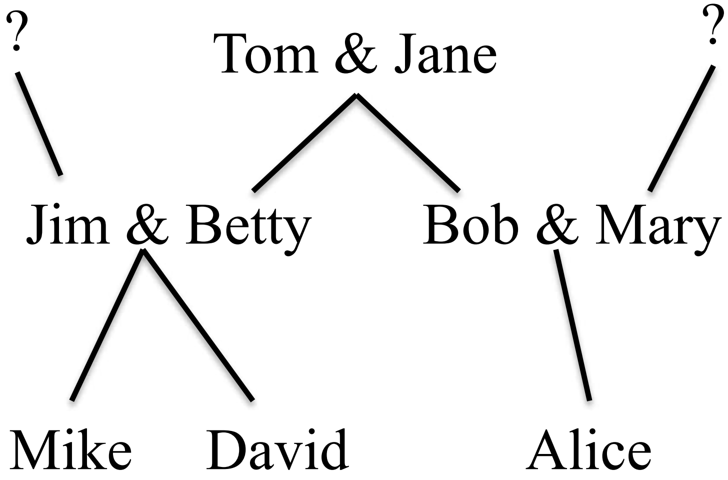
```
?- father(F,betty).
F = tom ;
false.

?- father(F,C).
F = tom,
C = betty ;
F = tom,
C = bob ;
...
false.

?- father(F,_).
F = tom ;
F = tom ;
F = jim ;
...
```

Parents and children, continued

Here is a set of facts for parents and children:

```
male(tom).
male(jim).
male(bob).
male(mike).
male(david).

female(jane).
female(betty).
female(mary).
female(alice).
```

```
parent(tom,betty).
parent(tom,bob).
parent(jane,betty).
parent(jane,bob).
parent(jim,mike).
parent(jim,david).
parent(betty,mike).
parent(betty,david).
parent(bob,alice).
parent(mary,alice).
```



Define **grandmother(GM,C)**.

```
grandmother(GM,C) :-
    female(GM), parent(GM, P),
    parent(P, C).

?- grandmother(GM,C).
GM = jane,
C = mike ;
GM = jane,
C = david ;
GM = jane,
C = alice ;
false.
```

Or, we could have defined **mother(M,C)** and written **grandmother** using **mother**.

Parents and children, continued

For who is Tom the father?

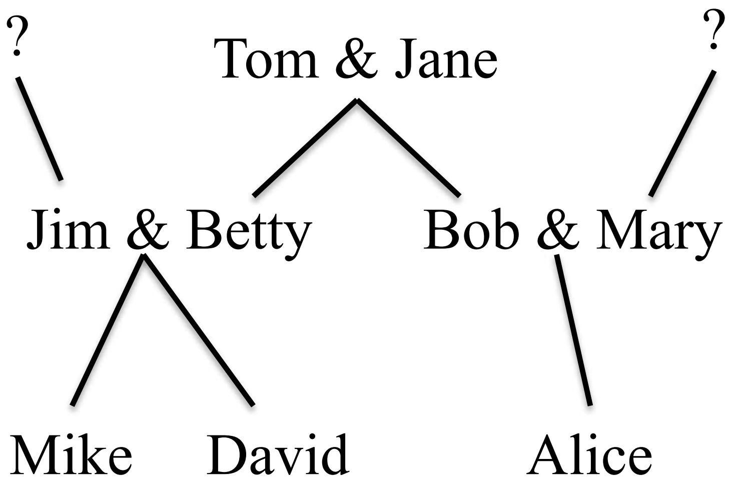
```
?- father(tom,C).
C = betty ;
C = bob.
```

What are all the father/daughter relationships?

```
?- father(F,D), female(D).
F = tom,
D = betty ;
F = bob,
D = alice ;
false.
```

Who is the father of Jim?

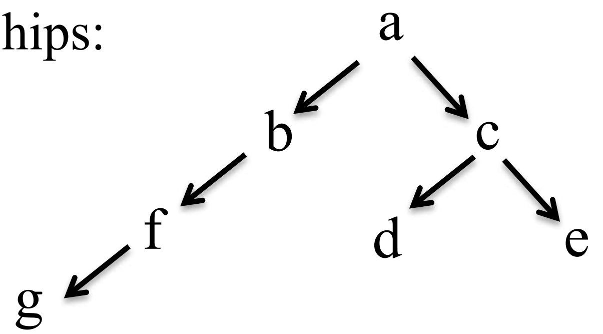
```
?- father(F,jim).
false.
```



Recursive predicates

Consider an abstract set of parent/child relationships:

```
parent(a,b).   parent(c,d).
parent(a,c).   parent(b,f).
parent(c,e).   parent(f,g).
```



Here is a <u>recursive</u> predicate for the relationship that A is an ancestor of X.

```
ancestor(A,X) :- parent(A, X).
ancestor(A,X) :- parent(P, X), ancestor(A,P).
```

<u>Experiment: swap these!</u>

In English:

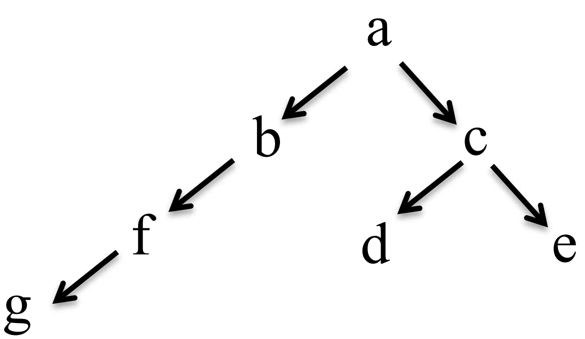
"A is an ancestor of X if A is the parent of X <u>or</u> P is the parent of X and A is an ancestor of P."

Usage:

```
?- ancestor(a,f).          % Is a an ancestor of f?
true

?- ancestor(c,b).          % Is c an ancestor of b?
false.
```

ancestors.pl

Recursive predicates



At hand:

```
parent(a,b).   parent(c,d).
...
ancestor(A,X) :- parent(A, X).
ancestor(A,X) :- parent(P, X), ancestor(A,P).
```

More examples:

```
?- ancestor(c,Descendant).  % Who are the descendants of c?
Descendant = e ;
Descendant = d ;
false.
```

What's the following query asking?

```
?- ancestor(A, e), ancestor(A,g).
A = a ;
false.
```

What are the common ancestors of `e` *and* `g`*?*

Iteration with recursion

A recursive rule can be used to perform an iterative computation.

Here is a predicate that prints the integers from 1 through N:

```
printN(0).
printN(N) :- N > 0, M is N - 1, printN(M), writeln(N).
```

Usage:

```
?- printN(3).
1
2
3
true .
```

Note that we're asking if **printN(3)** can be proven. The side effect of Prolog proving it is that the numbers 1, 2, and 3 are printed.

Is **printN(0).** needed?

Which is better—the above or using **between/3**?

More recursion

Here's a recursive predicate to compute the factorial of a number:

```
factorial(0, 1).
factorial(N, F) :-
    N > 0,
    M is N - 1,
    factorial(M, FM),
    F is N * FM.
```

Usage:

```
?- factorial(4,F).
F = 24 .
```

Note that this predicate can't be used to determine N:

```
?- factorial(N, 24).
ERROR: >/2: Arguments are not sufficiently instantiated
```

Sidebar: A common mistake with arithmetic

Here's a correct definition for **factorial**:

```
factorial(0, 1).
factorial(N, F) :- N > 0, M is N - 1, factorial(M, FM), F is N * FM.
```

Here is a **common mistake**:

```
factorial(0, 1).
factorial(N, F) :- N > 0, M is N - 1, factorial(M, F), F is N * F.
```

What's the mistake?

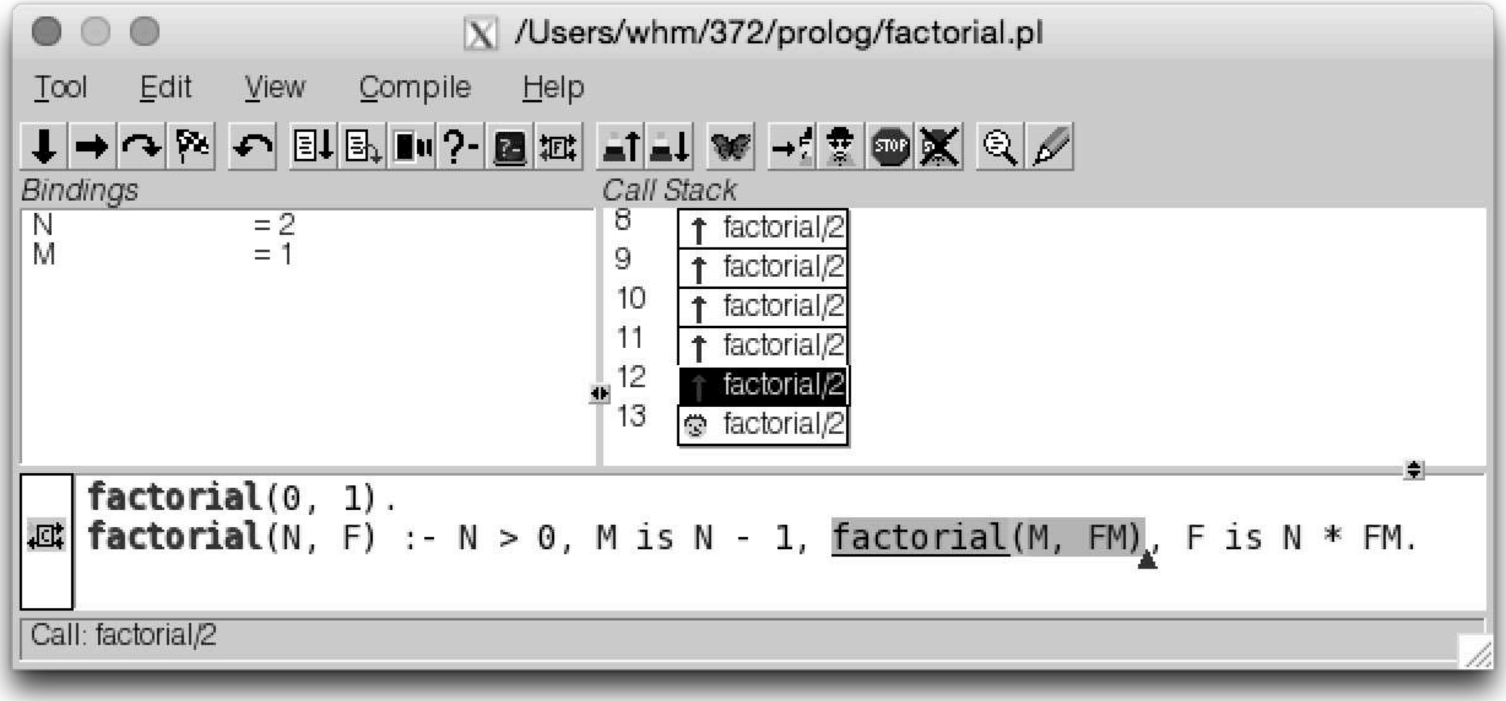
Remember that **is/2** unifies its left operand with the result of arithmetically evaluating its right operand. Further remember that unification is neither assignment nor comparison.

The goal **F is N * F** fails unless **N == 1**.

Sidebar: graphical tracing with **gtrace**

gtrace is the graphical counterpart of **trace**. Start it like this:

```
?- gtrace, factorial(6,F).
% The graphical front-end will be used for subsequent tracing
```



```
N        = 2
M        = 1
```

```
factorial(0, 1).
factorial(N, F) :- N > 0, M is N - 1, factorial(M, FM), F is N * FM.
```

Call: factorial/2

Type space to through step goals one at a time. Click on call stack elements to show bindings in that call. Try **showfoods**, **ancestor**, **grades**, and **printN**.

gtrace should work immediately on Windows and Macs. On Linux machines in the labs use "**ssh –X ...**" to login to lectura, and it should work there, too.

Generating alternatives with recursion

Here's a predicate that tests whether a number is odd:

```
    odd(N) :- N mod 2 =:= 1.
```

Note that **N mod 2** works because **=:=** evaluates its operands.

An alternative:

```
    odd(1).
    odd(N) :- odd(M), N is M + 2.
```

How does the behavior of the two differ?

Generating alternatives, continued

For reference:

```
    odd(1).
    odd(N) :- odd(M), N is M + 2.
```

Usage:

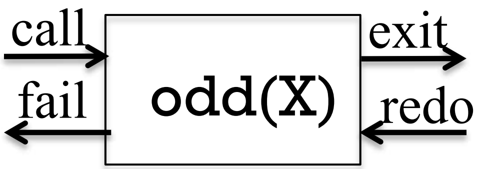
```
   ?- odd(5).
   true .

   ?- odd(X).
   X = 1 ;
   X = 3 ;
   X = 5 ;
   ...
```

What does **odd(2)** do?

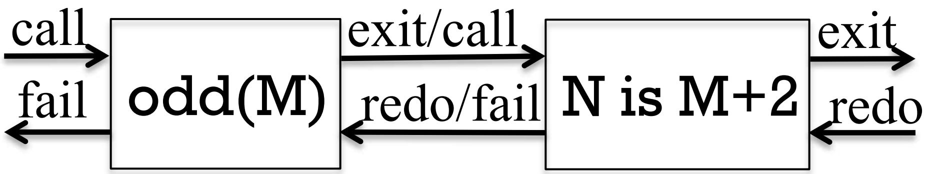
How does **odd(X)** work?

Generating alternatives, cont.

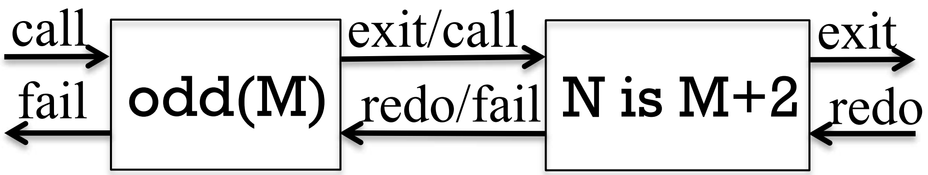
Query: ?- odd(X).



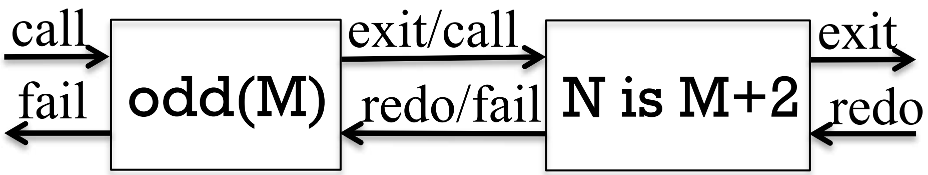
```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



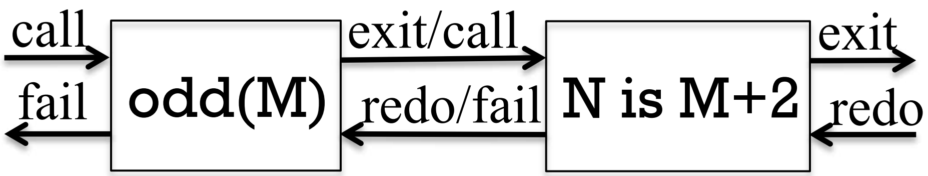
```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



Generating alternatives, continued

For reference:

```
odd(1).
odd(N) :- odd(M), N is M + 2.
```

The key point with generative predicates:

<u>If an alternative is requested, another activation of the predicate is created</u>.

As a contrast, think about how execution differs with this set of clauses:

```
odd(1).
odd(3).
odd(5).
odd(N) :- odd(M), N is M + 2.
```

Try `gtrace` with both the two-clause version at the top and the four-clause version just above.

Lists

List basics

A Prolog list can be literally specified by enclosing a comma-separated series of terms in square brackets:

```
[1, 2, 3]

[just, a, test, here]

[1, [one], 1.0, [a,[b,['c this']]]]
```

Note that there's no evaluation of the terms:

```
?- write([1, 2, odd(3), 4+5, atom(6)]).
[1,2,odd(3),4+5,atom(6)]
true.
```

If you enter a list literal as a query, it's taken as a request to consult files!

```
?- [abc, 123].
ERROR: source_sink `abc' does not exist ...
```

Unification with lists

Here are some unifications with lists:

```
?- [1,2,3] = [X,Y,Z].
X = 1,
Y = 2,
Z = 3.
```

```
?- [X,Y] = [1,[2,[3,4]]].
X = 1,
Y = [2, [3, 4]].
```

*Note unification of **Y** with list of lists.*

```
?- [X,Y] = [1].
false.
```

```
?- Z = [X,Y,X], X = 1, Y = [2,3].
Z = [1, [2, 3], 1],
X = 1,
Y = [2, 3].
```

*Note that **X** occurs twice in **Z**.*

We'll later see a head-and-tail syntax for lists.

Unification with lists, continued

Write a predicate **empty(X)** that succeeds iff **X** is an empty list. If called with something other than a non-empty list, it fails. Examples:

```
?- empty([]).
true.

?- empty([3,4,5]).
false.

?- empty(empty).
false.

?- empty(10).
false.
```

Solution:

```
empty([]).
```

Note: the latter two calls aren't errors; they just don't unify with **empty([])**.

Unification with lists, continued

Write a predicate `as123(X)` that succeeds iff `X` is a list with one, two, or three identical elements. Example:

```
?- as123([a]), as123([b,b]), write(ok), as123([1,2,3]),
     write('oops').
ok
false.

?- as123(L).
L = [_G2456] ;
L = [_G2456, _G2456] ;
L = [_G2456, _G2456, _G2456].
```

Solution:

```
as123([_]).
as123([X,X]).
as123([X,X,X]).
```

Not realizing the power of unification:

```
as123([X]).
as123([X,Y]) :- X = Y.
as123([X,Y,Z]) :- X = Y, Y = Z.
```

Built-in list-related predicates

SWI Prolog has a number of built-in predicates that operate on lists. One is **nth0**:

nth0(?Index, ?List, ?Elem)

True when Elem is the Index'th element of List. Counting starts at 0.

nth0 can be used in a variety of ways:

```
?- nth0(2, [a,b,a,d,c], X).      What is the third element of [a,b,a,d,c]?
X = a.

?- nth0(0, [a,b,a,d,c], b).      Is b the first element of [a,b,a,d,c]?
false.

?- nth0(N, [a,b,a,d,c], a).      Where are a's in [a,b,a,d,c]?
N = 0 ;
N = 2 ;
false.

?- nth0(N, [a,b,a,d,c], X).      What are the positions and values for all?
N = 0,
X = a ;
N = 1,
X = b ;
...
```

NOTE: **nth0** makes for a good example here, but use indexing judiciously! There are often better alternatives!

Built-ins for lists, continued

Recall:

```
as123([_]).
as123([X,X]).
as123([X,X,X]).
```

Problem: Using `as123` and `nth0`, write a predicate with this behavior:

```
?- gen3(test, L).
L = [test] ;
L = [test, test] ;
L = [test, test, test].
```

Solution:

```
gen3(X,L) :- as123(L), nth0(0, L, X).
```

Does the order of the goals matter?

More:

```
?- gen3(test, [test]).
true .

?- gen3(test, [a,b]).
false.
```

Built-ins for lists, continued

What do you think **length(?List, ?Len)** does?

Get the length of a list:

```
?- length([10,20,30],Len).
Len = 3
```

Anything else?

Make a list of uninstantiated variables:

```
?- length(L,3).
L = [_G907, _G910, _G913].
```

Speculate—what will **length(L,N)** do?

```
?- length(L,N).
L = [],
N = 0 ;
L = [_G919],
N = 1 ;
L = [_G919, _G922],
N = 2 ...
```

Built-ins for lists, continued

What does **reverse(?List, ?Reversed)** do?

Unifies a list with a reversed copy of itself.

```
?- reverse([1,2,3],R).
R = [3, 2, 1].

?- reverse([1,2,3],[1,2,3]).
false.
```

Write **palindrome(L)**.

```
palindrome(L) :- reverse(L,L).
```

Speculate—what's the result of **reverse(X,Y)**.?

```
?- reverse(X,Y).
X = Y, Y = [] ;
X = Y, Y = [_G913] ;
X = [_G913, _G916],
Y = [_G916, _G913] ;
X = [_G913, _G922, _G916],
Y = [_G916, _G922, _G913] ;
```

Built-ins for lists, continued

What might **numlist(+Low, +High, -List)** do?

```
?- numlist(5,10,L).
L = [5, 6, 7, 8, 9, 10].

?- numlist(10,5,L).
false.
```

Problem: Write **rnumlist(+High, +Low, -List)**

```
?- rnumlist(10,5,L).
L = [10, 9, 8, 7, 6, 5].
```

Solution:

```
rnumlist(High,Low,List) :-
    numlist(Low,High,List0), reverse(List0,List).
```

Built-ins for lists, continued

sumlist(+List, -Sum) unifies **Sum** with the sum of the values in **List**.

```
?- numlist(1,5,L), sumlist(L,Sum).
L = [1, 2, 3, 4, 5],
Sum = 15.
```

Will the following work?

```
?- sumlist([1+2, 3*4, 5-6/7], Sum).
Sum = 19.142857142857142.

?- X = 5, sumlist([X+X, X-X, X*X, X/X],R).
X = 5,
R = 36.
```

Is it good that **sumlist** handles arithmetic structures, too?

Sidebar: Developing a list-based predicate goal-by-goal

Write a predicate **sumGreater(+Target, -N, -Sum)** that finds the smallest **N** for which the sum of 1**..N** is greater than **Target**.

```
?- sumGreater(50, N, Sum).
N = 10,
Sum = 55 .

?- sumGreater(1000000, N, Sum).
N = 1414,
Sum = 1000405 .
```

Let's ignore Gauss's summation formula and have some fun with lists!

Sidebar, continued

Step one: Have a goal that instantiates **N** to 1, 2, ...

```
?- between(1, inf, N).
N = 1 ;
N = 2 ;
N = 3 ;
...
```

What's ***inf****?*

Step two: instantiate **L** to lists **[1]**, **[1,2]**, ...

```
?- between(1, inf, N), numlist(1, N, L).
N = 1,
L = [1] ;
N = 2,
L = [1, 2] ;
N = 3,
L = [1, 2, 3] ;
...
```

Sidebar, continued

Step three: Compute sum of 1..N.

```
?- between(1, inf, N), numlist(1,N,L), sumlist(L,Sum).
N = Sum, Sum = 1,
L = [1] ;
N = 2,
L = [1, 2],
Sum = 3 ;
...
```

Step four: Test sum against target value.

```
?- between(1, inf, N), numlist(1,N,L), sumlist(L,Sum), Sum > 20.
N = 6,
L = [1, 2, 3, 4, 5, 6],
Sum = 21 .
```

Note the incremental process followed, adding goals one-by-one and being sure the results for each step are what we expect.

Sidebar, continued

Step four, for reference:

```
?- between(1, inf, N), numlist(1,N,L), sumlist(L,Sum), Sum > 20.
N = 6,
L = [1, 2, 3, 4, 5, 6],
Sum = 21 .
```

Step five: Package as a predicate.

```
$ cat sg.pl
sumGreater(Target,N,Sum) :-
    between(1,inf,N), numlist(1,N,L), sumlist(L,Sum), Sum > Target.

% swipl sg.pl
...
?- sumGreater(1000,N,Sum).
N = 45,
Sum = 1035 ;
N = 46,
Sum = 1081 ;
```

Is it good or bad that it produces alternatives?

Built-ins for lists, continued

Here's **atom_chars(?Atom, ?Charlist)**:

```
?- atom_chars(abc,L).
L = [a, b, c].

?- atom_chars(A, [a, b, c]).
A = abc.
```

Problem: write **rev_atom/2**. Hint: Write it as a test, the latter case.

```
?- rev_atom(testing,R).
R = gnitset.

?- rev_atom(testing,gnitset).
true.
```

Solution:

```
rev_atom(A,RA) :-
    atom_chars(A,AL), reverse(AL,RL), atom_chars(RA,RL).
```

Built-ins for lists, continued

Problem: write **eqlen(+A1,+A2)**, to test whether two atoms are the same length.

```
?- eqlen(test,this).
true.

?- eqlen(test,it).
false.
```

Solution:

```
eqlen(A1,A2) :- atom_chars(A1,C1), length(C1,Len),
   atom_chars(A2,C2), length(C2,Len).
```

Note this vs. a **Len1 == Len2** goal!

Built-ins for lists, continued

msort(+List, -Sorted) unifies **Sorted** with a sorted copy of **List**:

```
?- msort([3,1,7], L).
L = [1, 3, 7].

?- atom_chars(prolog, L), msort(L,S), atom_chars(A,S).
L = [p, r, o, l, o, g],
S = [g, l, o, o, p, r],
A = gloopr.
```

If the list is heterogeneous, elements are sorted in "standard order":

```
?- msort([xyz, 5, [1,2], abc, 1, 5, x(a)], Sorted).
Sorted = [1, 5, 5, abc, xyz, x(a), [1, 2]].
```

sort/2 is like **msort/2** but also removes duplicates.

```
?- sort([xyz, 5, [1,2], abc, 1, 5, x(a)], Sorted).
Sorted = [1, 5, abc, xyz, x(a), [1, 2]].
```

The **member** predicate

member(?Elem, ?List) succeeds when **Elem** can be unified with a member of **List**.

member can be used to check for membership:

```
?- member(30, [10, twenty, 30]).
true.
```

member can be used to generate the members of a list:

```
?- member(X, [10, twenty, 30]).
X = 10 ;
X = twenty ;
X = 30.
```

Problem: Print the numbers from 100 through 1. (Use **member**!)

```
?- numlist(1,100,L), reverse(L,R), member(E,R), writeln(E), fail.
100
99
...
```

member, continued

Problem: Write a predicate **has_vowel(+Atom)** that succeeds iff **Atom** has a lowercase vowel.

```
?- has_vowel(ack).
true

?- has_vowel(pfft).
false.
```

Solution:

```
 has_vowel(Atom) :-
      atom_chars(Atom,Chars),
      member(Char,Chars),
      member(Char,[a,e,i,o,u]).
```

Explain it!

The append predicate

Here's how the documentation describes append/3:

```
?- help(append/3).
append(?List1, ?List2, ?List1AndList2)
   List1AndList2 is the concatenation of List1 and List2
```

Usage:

```
?- append([1,2], [3,4,5], R).
R = [1, 2, 3, 4, 5].

?- numlist(1,4,L1), reverse(L1,L2), append(L1,L2,R).
L1 = [1, 2, 3, 4],
L2 = [4, 3, 2, 1],
R = [1, 2, 3, 4, 4, 3, 2, 1].
```

What else can we do with append?

append, continued

What will the following do?

```
?- append(A, B, [1,2,3]).
A = [],
B = [1, 2, 3] ;
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
A = [1, 2, 3],
B = [] ;
false.
```

The query can be thought of as asking this:

"For what values of A and B is their concatenation [1,2,3]?

append, continued

Think of **append(L1,L2,L3)** as demanding a relationship between the three lists:

> **L3** must consist of the elements of **L1** followed by the elements of **L2**.

If **L1** and **L2** are instantiated, **L3** must be their concatenation.

If only **L3** is instantiated then **L1** and **L2** represent (in turn) all the possible ways to divide **L3**.

What are the other possibilities?

Important:

> We can do a lot of list processing by establishing constraints with **append** (and other predicates) and asking Prolog to find cases when those constraints are true.

append is the Swiss Army Knife of list processing in Prolog!

append, continued

Problem: Using **append**, write **starts_with(?List, ?Prefix)** that expresses the relationship that **List** starts with **Prefix**.

```
?- starts_with([1,2,3,4], [1,2]).
true.

?- starts_with([1,2,3], L).
L = [] ;
L = [1] ;
L = [1, 2] ;
L = [1, 2, 3] ;
false.
```

Solution:

```
starts_with(L, Prefix) :- append(Prefix, _, L).
```

Problem: Write **ends_with**.

```
?- ends_with([a,b,c],[d,e]).
false.

?- ends_with([a,b,c],[b,c]).
true ;
false.
```

Solution:

```
ends_with(List, Suffix) :-
    append(_, Suffix, List).
```

What will the following do?

```
?- starts_with(Start, [1,2,3]).
Start = [1, 2, 3|_G1182].
```

append, continued

Haskell meets Prolog:

```
take(L, N, Result) :-
     length(Result,N), append(Result, _, L).

?- take([1,2,3,4,5], 3, L).
L = [1, 2, 3].

?- take([1,2,3,4,5], N, L).
N = 0,
L = [] ;
N = 1,
L = [1] ;
N = 2,
L = [1, 2] ;
...
```

```
drop(L, N, Result) :-
    append(Dropped, Result, L), length(Dropped, N).
```

sumsegs (practice with **append**)

Write **sumsegs(+List, +N, -Sums)**, where **Sums** is a list with the sum of the first **N** elements of **List**, then the sum of the next **N**, and so forth.

```
?- sumsegs([1, 2, 3, 4, 5, 6],2,R).
R = [3, 7, 11] ;
false.

?- sumsegs([1,2,3,4,5,6],4,R).
R = [10] ;
false.

?- sumsegs([1,2,3,4,5,6],7,R).
R = [] ;
false.
```

How can we approach it?

sumsegs, continued

For reference:

```
    ?- sumsegs([1, 2, 3, 4, 5, 6], 2, R).
    R = [3, 7, 11] ;
```

Solution:

```
  % If fewer than N elements remain, produce an empty list.
  sumsegs(List, N, []) :- length(List,Len), Len < N.

  sumsegs(List, N, Sums) :-
    % Get the first N elements into Seg and compute their sum.
    length(Seg, N), append(Seg, Rest, List), sumlist(Seg, Sum),

    % Compute the sums for the rest of the list.
    sumsegs(Rest, N, RestOfSums),

    % Specify the result by forming a list whose first element is the
    % sum of the first segment followed by the sums for the rest of the list.
    append([Sum], RestOfSums, Sums).
```

Key technique!

Generation with **append**

Here's a predicate that generates successive N-long chunks of a list:

```
    chunk(L, N, Chunk) :-
       length(Chunk, N), append(Chunk, _, L).

    chunk(L, N, Chunk) :-
       length(Junk, N), append(Junk, Rest, L), chunk(Rest, N, Chunk).
```

Usage:

```
 ?- chunk([1,2,3,4,5],2,L).
 L = [1, 2] ;
 L = [3, 4] ;
 false.

 ?- numlist(1,100,L), chunk(L,5,C), sumlist(C,Sum), between(300,350,Sum).
 L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
 C = [61, 62, 63, 64, 65],
 Sum = 315 ;

 L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
 C = [66, 67, 68, 69, 70],
 Sum = 340 ;
 false.
```

Here's **chunk** again. How does it work?

```
chunk(L,N,Chunk) :-
   length(Chunk,N), append(Chunk,_,L).

chunk(L,N,Chunk) :-
   length(Junk, N), append(Junk,Rest,L), chunk(Rest,N,Chunk).
```

```
?- chunk([1,2,3,4,5],2,L).
L = [1, 2] ;
L = [3, 4] ;
false.
```

Consider the call **chunk([a,b,c,d,e,f], 2, Chunk)**:

The first clause produces the first **N** elements of **L**. (**Chunk = [a,b]**)

The second clause first uses **length** and **append** to form a list **Rest** that is **L** minus the first **N** elements (**Rest = [c,d,e,f]**).

The second clause then calls **chunk([c,d,e,f], 2, Chunk)**, creating another activation of **chunk**.

Its first clause will produce the first **N** elements of **[c,d,e,f]**.

Its second clause will end up calling **chunk([e,f], 2, Chunk)** creating a third activation of **chunk**.

Important: Note the similarity to **odd** on slide 144.

gensums (practice with generation)

Recall **sumsegs**:

```
?- sumsegs([1, 2, 3, 4, 5, 6],2,R).
R = [3, 7, 11] ;
false.
```

Problem: Instead of producing a list, generate the sums:

```
?- gensums([1,2,3,4,5,6,7], 2, R).
R = 3 ;
R = 7 ;
R = 11 ;
false.
```

Solution:

```
gensums(List, N, Sum) :-
    chunk(List, N, Seg), sumlist(Seg, Sum).
```

Exercise: Write **gensums** without using **chunk**.

Next set of slides

"Getting it" with **append**

"The concept encountered in [**a8**'s] **splits.pl** is simple in hindsight, but represents something pivotal to even vaguely understanding Prolog. There was a moment several minutes ago when it finally struck me that **append** is *not* a function, but some ephemeral statement of fact with several combinations of conditions that satisfy it."

—Bailey Swartz, Spring '15

The **findall** predicate

Here are some examples with a new predicate, **findall**:

```
 ?- findall(F, food(F), Foods).
 Foods = [apple, broccoli, carrot, lettuce, orange, rice].

?- findall(pos(N,X), nth0(N, [a,b,a,d,c], X), Posns).
Posns = [pos(0, a), pos(1, b), pos(2, a), pos(3, d), pos(4, c)].

?- findall(X, (between(1,100,X), X rem 13 =:= 0), Nums).
Nums = [13, 26, 39, 52, 65, 78, 91].
```

In your own words, what does **findall** do?

findall, continued

For reference:

```
?- findall(F, food(F), Foods).
Foods = [apple, broccoli, carrot, lettuce, orange, rice].
```

SWI's documentation: (with a minor edit)

findall(+Template, :Goal, -List)

> Create a list of the instantiations Template gets successively on backtracking over Goal and unify the result with List. Succeeds with an empty list if Goal has no solutions.

Template is not limited to being a single variable. It might be a structure.

The second argument can be a single goal, or several goals joined with conjunction.

The third argument is instantiated to a list of terms whose structure is determined by the template. Above, each term is just an atom.

findall, continued

For reference:

findall(+Template, :Goal, -Bag) *(The colon in* ***:Goal*** *means"meta-argument")*

Examples to show the relationship of the template and the resulting list:

```
?- findall(x, food(F), Foods).
Foods = [x, x, x, x, x, x].

?- findall(x(F), food(F), Foods).
Foods = [x(apple), x(broccoli), x(carrot), x(lettuce), x(orange), x(rice)].

?- findall(1-F, food(F), Foods).
Foods = [1-apple, 1-broccoli, 1-carrot, 1-lettuce, 1-orange, 1-rice].
```

What does **findall** remind you of?

Important:

findall is said to be a *higher-order predicate*. It's a predicate that takes a goal, **food(F)** in this case.

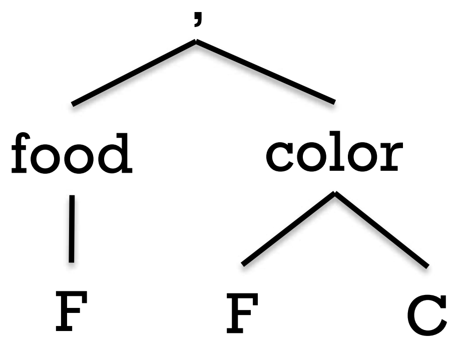
findall, continued

Here's a case where **:Goal** is a conjunction of two goals.

```
?- findall(F-C, (food(F),color(F,C)), FoodsAndColors).
FoodsAndColors = [apple-red, broccoli-green, carrot-orange,
lettuce-green, orange-orange, rice-white].
```

display sheds some light on that conjunction:

```
?- display((food(F),color(F,C))).
,(food(_G835),color(_G835,_G838))
true.
```



The conjunction is a two-term structure whose functor is a comma.

findall, continued

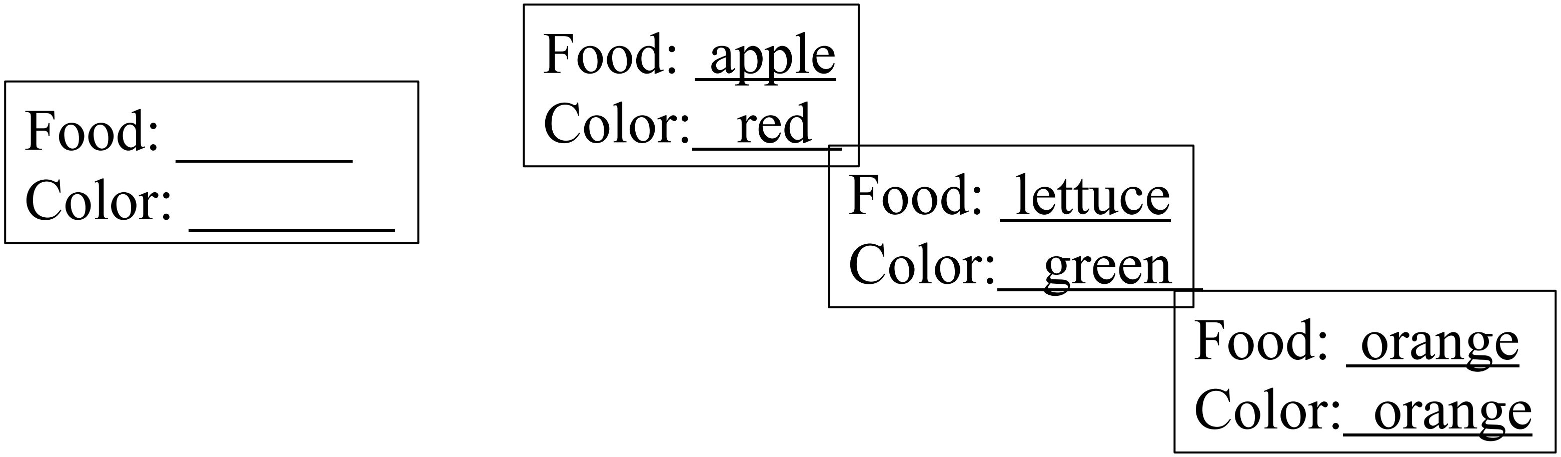
Original Thought from Noah Sleiman, Spring '14 :

An easy way to think of it when using the uninstantiated first term (to find the elements of interest) is this:

findall*(What I call it, How I got it, Where I put it)*

Another view:

- Think of the template (the first argument) as a paper form with some number of blanks to fill in.
- Each time the goal produces a result, we fill out a copy of that form and put it on the list.
- A series of completed forms is the result of **findall**.



Food: ______
Color: ________

Food: apple
Color: red

Food: lettuce
Color: green

Food: orange
Color: orange

member vs. findall

member and findall are somewhat inverses of each other.

If we want to generate values from a list, we can use member:

```
?- member(X, [a,b,c]).
X = a ;
X = b ;
X = c.
```

If we have a query that generates values, we can make a list with findall:

```
?- findall(X, member(X, [a,b,c]), Values).
Values = [a, b, c].
```

Practice with **findall**

Problem: Write a predicate **sumlists** that produces a list of the sums of integer lists.

```
?- sumlists([[1,2], [10,20,30], []],Sums).
Sums = [3, 60, 0].
```

Recall sumlist:

```
?- sumlist([1,2,3],Sum).
Sum = 6.
```

Solution:

```
sumlists(Lists, Sums) :-
    findall(Sum, (member(List,Lists),sumlist(List,Sum)), Sums).
```

Note that **findall**'s goal is a conjunction of two goals.

Practice with **findall**, continued

Problem: Write a variant of **sumlists** that requires sums to meet a minimum:

```
?- minsums([[10,20,30],[1,2,3],[50]], 25, Sums).
Sums = [sum([10, 20, 30], 60), sum([50], 50)].

?- minsums([[10,20,30],[1,2,3],[50]], 250, Sums).
Sums = [].
```

Note that the result is a list of structures holding both the list and its sum.

Solution:

```
minsums(Lists, Min, Sums) :-
    findall(
       sum(List,Sum),
       (member(List,Lists),sumlist(List,Sum),Sum>=Min),
       Sums).
```

A scoping issue with **findall**

What's happening in the following query?

```
?- X=a, findall(X-Y, member(Y, [a,b,c]), Values), write(X-Y).
a-_G1095
X = a,
Values = [a-a, a-b, a-c].
```

Was **X=Y**

The scope of variables created during a **findall** query is limited to that query.

Above, **X** is bound prior to the **findall** and can be used in it.

The **Y** inside the **findall** is unrelated to the **Y** in **write(X-Y)**.

Typing in Prolog

Static or dynamic?

Recall that with a statically typed language, the type of every variable and expression can be determined by static analysis of code.

Is Prolog statically typed or dynamically typed? Or is it something else?

Wikipedia says, "Prolog is an untyped language." (4/19/16)
Does Prolog not have types?

BCPL is sometimes described as an untyped language where all values are word-sized objects.

Imagine a language where everything is a string. Is it untyped?

"A programming language is untyped if it allows [you] to apply any operation on any data, and all datatypes are considered as sequences of bits of various lengths."—**http://progopedia.com/typing/untyped**

The books say...

There are only two clear references to data types in C&M:

> p. 28: "The functor names the general kind of structure, and corresponds to a **datatype** in an ordinary programming language."
>
> p. 122, under "Classifying Terms": If we wish to define predicates which will be used with a wide variety of argument **types**, it is useful to be able to distinguish in the definition what should be done for each possible **type**."

Covington has several references to types, including these:

> p. 93: "Terms of this form are called STRUCTURES. The functor is always an atom, but the arguments can be terms of any **type** whatever."
>
> p. 130: "If `number_codes` is given a string that doesn't make a valid number, or if either of its arguments is of the wrong **type**, it raises a runtime error condition."

Another voice:

> ISO Prolog's exception handling mechanism has a `type_error(Type,Term)` structure.

swipl says...

Let's see if any predicates concern types.

```
?- apropos(type).
...
integer/1       Type check for integer
rational/1      Type check for a rational number
number/1        Type check for integer or float
atom/1          Type check for an atom
blob/2          Type check for a blob
string/1        Type check for string
```

Can we produce a type error?

```
?- atom_length(a(1), Len).
ERROR: atom_length/2: Type error: `text' expected, found `a(1)'
```

Could we find the above error with static analysis?

Bottom line: I'm comfortable saying that Prolog has types.

Back to "statically typed or dynamically typed?"

Again:

> In a statically typed language, the type of every variable and expression can be determined by static analysis of code.

Can we construct a Prolog program where a value's type cannot be determined by looking at the code?

Here's such a program:

```
  f('one').  f(a(1)).

  prog :- f(X), random(2) > 0,
     atom_length(X, Len), writeln(Len).
```

```
?- prog.
3
true .

?- prog.
false.

?- prog.
ERROR: atom_length:
Type error: ...
```

The type of **X** depends on a random number and thus varies from run to run.

Therefore, Prolog is dynamically typed! Right?

Low-level list processing

Heads and tails

The list [1,2,3] can be specified in terms of a head and a tail, like this:

```
[1 | [2,3]]
```

More generally, a list can be specified as a sequence of initial elements and a tail.

The list [1,2,3,4] can be specified in any of these ways:

```
[1 | [2,3,4]]

[1,2 | [3,4]]

[1,2,3 | [4]]

[1,2,3,4 | []]
```

Haskell equivalents:

```
1:[2,3,4]

1:2:[3,4]

1:2:3:[4]

1:2:3:4:[]
```

General form: [E_1, E_2, ..., E_n | Tail]

Unifications with lists

Consider this unification:

```
?- [H|T] = [1,2,3,4].
H = 1,
T = [2, 3, 4].
```

What instantiations are produced by these unifications?

```
?- [X,Y | T] = [1, 2, 3].
X = 1,
Y = 2,
T = [3].

?- [X,Y | T] = [1, 2].
X = 1,
Y = 2,
T = [].

?- [1, 2 | [3,4]] = [H | T].
H = 1,
T = [2, 3, 4].

?- A = [1], B = [A|A].
A = [1],
B = [[1], 1].
```

Simple list predicates

Here's a rule that describes the relationship between a list and and its head:

```
head(L, H) :- L = [H|_].
```

The head of **L** *is* **H** *if* **L** *unifies with a list whose head is* **H***.*

Usage:

```
?- head([1,2,3],H).
H = 1.

?- head([2],H).
H = 2.

?- head([],H).
false.

?- L = [X,X,b,c], head(L, a).
L = [a, a, b, c],
X = a.
```

Can we make better use of unification and define **head/2** more concisely?

```
head([H|_], H).
```

The head of a list whose head is **H** *is* **H***.*

Prolog vs. Haskell

Note the contrast between Haskell and Prolog:

Haskell:

head is a function that produces the first element of a list.

Prolog:

head is a predicate that describes the relationship between a value and the first element of a list.

In Prolog **head** can:

- Produce the first element of a list.
- See if the first element of a list is a given value.
- Produce a list that will unify with any list whose head is a given value.

Implementing **member**

Here is one way to define the built-in **member/2** predicate:

```
member(X,L) :- L = [X|_].
member(X,L) :- L = [_|T], member(X, T).
```

Usage:

```
?- member(1, [2,1,4,5]).
true ;
false.

?- member(a, [2,1,4,5]).
false.

?- member(X, [2,1,4,5]).
X = 2 ;
X = 1 ;
X = 4 ;
X = 5.
```

member, continued

For reference:

```
member(X,L) :- L = [X|_].
member(X,L) :- L = [_|T], member(X, T).
```

Problem: Define **member** more concisely.

```
member(X,[X|_]).
```

X is a member of the list having X as its head

```
member(X,[_|T]) :- member(X,T).
```

X is a member of the list having T as its tail if X is a member of T

Exercise: Following the example of slide 144, trace through how **member** generates elements from a list, like this:

```
?- member(X, [a,b,c]).
X = a ;
X = b ;
...
```

Implementing **last**

Problem: Define a predicate **last(L,X)** that describes the relationship between a list **L** and its last element, **X**.

```
?- last([a,b,c],X).
X = c.

?- last([],X).
false.

?- last(L,last), head(L,first), length(L,2).
L = [first, last] .
```

last is a built-in predicate but here's how we'd write it.

```
last([X],X).
last([_|T],X) :- last(T,X).
```

allsame

Problem: Define a predicate **allsame(L)** that describes lists in which all elements have the same value.

```
?- allsame([a,a,a]).
true

?- allsame([a,b,a]).
false.

?- L = [A,B,C], allsame(L), B = 7, write(L).
[7,7,7]
L = [7, 7, 7],
A = B, B = C, C = 7 .

?- length(L,5), allsame(L), head(L,x).
L = [x, x, x, x, x] .
```

Here's another way to test it:

```
?- allsame(L).
L = [_G1635] ;
L = [_G1635, _G1635] ;
L = [_G1635, _G1635, _G1635] ;
...
```

Solution:

```
allsame([_]).
allsame([X,X|T]) :- allsame([X|T]).
```

adjacent

Write a predicate **adjacent(?A, ?B, ?L)** that expresses the relationship that **A** and **B** are adjacent in the list **L**.

```
?- adjacent(3, 4, [1,2,3,4,5]).
true ;
false.

?- adjacent(a, X, [a,b,a,a,c,a]).
X = b ;
X = a ;
X = c ;
false.

?- adjacent(A,B,[1,2,3,4]).
A = 1, B = 2 ;
A = 2, B = 3 ;
A = 3, B = 4 ;
false.
```

```
?- adjacent(A,B,L).
L = [A, B|_G28] ;
L = [_G30, A, B|_G28] ;
L = [_G30, _G36, A, B|_G28] ;
```

Solution: (hint—use append!)

```
    adjacent(A,B,L) :- append(_, [A,B|_], L).
```

sf_gen

Write a predicate **sf_gen** that generates elements from a list in this order:

Second, first, fourth, third, sixth, fifth, ...

Usage:

```
?- sf_gen([a,b,c,d,e],X).
X = b ;
X = a ;
X = d ;
X = c ;
false.   % doesn't produce e because it would break the pattern.
```

sf_gen, continued

```
?- sf_gen([a,b,c,d,e],X).
X = b ;
X = a ;
X = d ;
X = c ;
```

Solution:

```
    % Produce the second element.
    sf_gen([_,X|_], X).

    % Produce the first element, if at least two.
    sf_gen([X,_|_], X).

    % Get rid of the first two elements and start all over.
    sf_gen([_,_|T], X) :- sf_gen(T,X).
        sf_gen([a, b|[c,d,e]], X) :- sf_gen([c,d,e], X).
```

Implementing **numlist**

Problem: Implement a slight variant of the built-in **numlist** predicate.

```
?- numlist(5,10,L).
L = [5, 6, 7, 8, 9, 10].

?- numlist(5,1,L).  % the built-in numlist fails for this case
L = [].
```

Solution, v1:

```
numlist(Low, High, []) :- Low > High, !.

numlist(Low, High, Result) :-
    Next is Low + 1,
    numlist(Next, High, Rest),
    Result = [Low|Rest].
```

```
numlist(1,4, Result) :-
    Next is 1 + 1,
    numlist(2, 4, Rest),
        Rest gets bound to [2,3,4]
    Result = [1|[2,3,4]].
```

What happens if we remove the cut?

numlist, continued

Solution, v1:

```
numlist(Low, High, []) :- Low > High, !.

numlist(Low, High, Result) :-
    Next is Low + 1,
    numlist(Next, High, Rest),
    Result = [Low|Rest].
```

How can we make better use of unification?

```
numlist(Low, High, []) :- Low > High, !.

numlist(Low, High, [Low|Rest]) :-
    Next is Low + 1,
    numlist(Next, High, Rest).
```

delete

Problem: Implement the built-in predicate **delete**.

```
?- delete([a,b,a,c,b,a], a, R).
R = [b, c, b].
```

Solution:

```
delete([], _, []).
delete([X|T], X, R) :- delete(T, X, R), !.
delete([E|T], X, [E|R]) :- delete(T, X, R).
```

How could we write it without a cut?

```
delete([], _, []).
delete([X|T], X, R) :- delete(T, X, R).
delete([E|T], X, [E|R]) :- E \== X, delete(T, X, R).
```

Implementing **length**

Problem: Write a predicate that behaves like the built-in **length/2**.

```
?- length([],N).
N = 0.

?- length([a,b,c,d], N).
N = 4.

?- length(L,1).
L = [_G901] .

?- length(L,N).
L = [],
N = 0 ;
L = [_G913],
N = 1 ;
L = [_G913, _G916],
N = 2 ;
...
```

Solution:

```
    length([], 0).
    length([_|T], Len) :-
        length(T,TLen),
        Len is TLen + 1.
```

Implementing **append**

Recall the description of the built-in **append** predicate:

```
?- help(append/3).
append(?List1, ?List2, ?List1AndList2)
   List1AndList2 is the concatenation of List1 and List2
```

The usual definition of **append**:

```
append([], X, X).
append([X|L1], L2, [X|L3]) :-  append(L1, L2, L3).
```

How does it work?

Note the similarity to ++ in Haskell:

```
(++) [] rhs = rhs
(++) (x:xs) rhs = x : (xs ++ rhs)
```

But, Haskell's ++ only lets us concatenate lists. Prolog's **append** expresses a relationship between three lists.

Implementing **append**, continued

At hand:

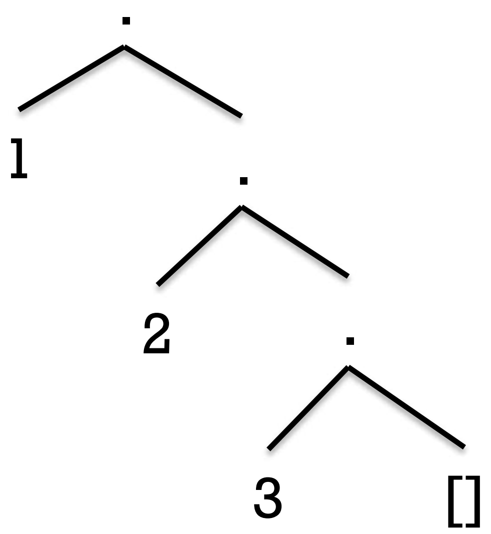
```
append([], X, X).
append([X|L1], L2, [X|L3]) :-  append(L1, L2, L3).

?- trace, append([1,2,3],[a,b,c],X).
   Call: (8) append([1, 2, 3], [a, b, c], _G971) ? creep
   Call: (9) append([2, 3], [a, b, c], _G1097) ? creep
   Call: (10) append([3], [a, b, c], _G1100) ? creep
   Call: (11) append([], [a, b, c], _G1103) ? creep
   Exit: (11) append([], [a, b, c], [a, b, c]) ? creep
   Exit: (10) append([3], [a, b, c], [3, a, b, c]) ? creep
   Exit: (9) append([2, 3], [a, b, c], [2, 3, a, b, c]) ? creep
   Exit: (8) append([1, 2, 3], [a, b, c], [1, 2, 3, a, b, c]) ? creep
X = [1, 2, 3, a, b, c].
```

Note that all of the **Exit:** lines in the trace above show an **append** relationship that's true.

Lists are structures

In fact, lists are structures:

```
?- display([1,2,3]).
.(1,.(2,.(3,[])))        % not in 7.2.3... ☹
```



Essentially, **./2** is the "cons" operation in Prolog.

By default, lists are shown using the [...] notation:

```
?- X = .(a, .(b,[])).
X = [a, b].
```

We can write **member/2** like this:

```
member(X, .(X,_)).
member(X, .(_,T)) :- member(X,T).
```

What does the following produce?

```
?- X = .(3,4).
X = [3|4].
```

A Lisp programmer would call this a "dotted-pair".

=..

=../2, spoken as "univ", expresses a relationship between structures and lists:

```
?- f(a,b,c) =.. L.
L = [f, a, b, c].

?- f(a,g(c,d),e(f)) =.. L.
L = [f, a, g(c, d), e(f)].

?- 1*2+3/4 =.. L.
L = [+, 1*2, 3/4].

?- S =.. [writeln,hello], call(S).
hello
S = writeln(hello).
```

"univ", continued

Problem: Create a predicate **functor** that produces the functors in a binary tree.

```
?- functor(1 * 2 + 3 / 4, F).
F = (+) ;
F = (*) ;
F = (/) ;
false.
```

Solution:

```
functor(S,F) :- S =.. [F|T], T \== [].
functor(S,F) :- S =.. [_,Left,_], functor(Left,F).
functor(S,F) :- S =.. [_,_,Right], functor(Right,F).
```

Which is the better name for this predicate, **functor** or **functor<u>s</u>**?

Sidebar: "univ"

C&M 5e p.130 says,

> The predicate "=.." (pronounced "univ" for historical reasons)...

I asked about it on the prolog channel on irc.freenode.net:

> x77686d: C&M says that =.. is called "univ" for historical reasons.
> Anybody know the story behind that?
> dmiles: for a long time we could only used named operators
> dmiles: why it was called univ instead of t2l .. i dont know
> dmiles: oh unify vector
> dmiles: erl v in prolog means array/vector

The first edition of C&M (1981) has that same line...

Database (knowledgebase) manipulation

assert and **retract**

A Prolog program is a database of facts and rules.

The database can be changed dynamically by adding facts with **assert/1** and deleting facts with **retract/1**.

A predicate to establish that certain things are foods:

```
makefoods :-                                          % foods3.pl
   assert(food(apple)),
   assert(food(broccoli)),  assert(food(carrot)),
   assert(food(lettuce)),  assert(food(rice)).
```

Evaluating **makefoods** <u>adds facts to the database</u>:

```
?- food(F).      ("positive-control" test—be sure no foods already!)
ERROR: toplevel: Undefined procedure: food/1

?- makefoods.
true.

?- findall(F, food(F), L).
L = [apple, broccoli, carrot, lettuce, rice].
```

assert and **retract**, continued

A fact can be removed with **retract**:

```
?- retract(food(carrot)).
true.

?- food(carrot).
false.
```

retractall removes all matching facts.

```
?- retractall(food(_)).
true.

?- food(X).
false.
```

assert and **retract**, continued

If we query **makefoods** multiple times, it makes multiple sets of food facts.

```
?- makefoods.
true.

?- makefoods.
true.

?- findall(F,food(F),Foods).
Foods = [apple, broccoli, carrot, lettuce, rice, apple, broccoli,
carrot, lettuce|...].
```

Let's start **makefoods** with a **retractall** to get a clean slate every time.

```
makefoods :-
    retractall(food(_)),
    assert(food(apple)),
    assert(food(broccoli)), assert(food(carrot)),
    assert(food(lettuce)), assert(food(rice)).
```

assert and **retract**, continued

Important: asserts and retracts are not undone with backtracking.

```
?- assert(f(1)), assert(f(2)), fail.
false.

?- f(X).
X = 1 ;
X = 2.

?- retract(f(1)), fail.
false.

?- f(X).       A redo of retract(f(1)) did not restore f(1).
X = 2.
```

There is no ability to directly change a fact. Instead, a fact is changed by retracting it and then asserting it with different terms.

assert and **retract**, continued

A rule to remove foods of a given color (assuming the **color/2** facts are present):

```
rmfood(C) :- food(F), color(F,C),
   retract(food(F)),
   write('Removed '), write(F), nl, fail.
```

Usage:

```
?- rmfood(green).
Removed broccoli
Removed lettuce
false.

?- findall(F, food(F), L).
L = [apple, carrot, rice].
```

The color facts are not affected—**color(broccoli, green)** and **color(lettuce,green)** still exist.

A simple calculator

Here's a very simple calculator: (**calc.pl**)

```
?- calc.
> print.
0
> add(20).
> sub(7).
> print.
13
> set(40).
> print.
40
> exit.
true.
```

Note that the commands themselves are Prolog terms.

Simple calculator, continued

A loop that reads and prints terms:

```
calc0 :- prompt(_, '> '),
        repeat, read(T), format('Read ~w~n', T), T = exit, !.
```

Interaction:

```
?- calc0.
> a.
Read a
> ab(c,d,e).
Read ab(c,d,e)
> exit.
Read exit
true.
```

How does the loop work?

`prompt/2` sets the prompt that's printed when `read/1` is called.

`repeat/0` always succeeds. If `repeat` is backtracked into, it simply sends control back to the right. (Think of its redo port being wired to its exit port.)

The predicate `read(-X)` reads a Prolog term and unifies it with `X`.

Simple calculator, continued

Partial implementation:

```
init :-
     retractall(value(_)),
     assert(value(0)).

do(set(V)) :-
     retract(value(_)),
     assert(value(V)).

do(print) :- value(V), writeln(V).

do(exit).

calc :-
     init, prompt(_, '> '),
     repeat, read(T), do(T), T = exit, !.
```

```
?- calc.
> print.
0
> add(20).
> sub(7).
> print.
13
> set(40).
> print.
40
> exit.
true.
```

How can **add(N)** and **sub(N)** be implemented? (No repetitious code, please!)

Simple calculator, continued

add and **subtract**:

```
    do(add(X)) :-
        value(V0),
        V is V0 + X,
        do(set(V)).     % Is this a nested call to set(V)?!

    do(sub(X0)) :-
        X is -X0,
        do(add(X)).
```

Could **sub** be shortened to the following?

```
    do(sub(X)) :- do(add(-X)).
```

Try **add(3+4*5)**, too.

Exercise: Add **double** and **halve** commands.

Word tally

We can use facts like we might use a Java map or a Ruby hash.

Imagine a word tallying program in Prolog:

```
?- tally.
|: to be or
|: not to be ought not
|: to be the question
|: (Empty line ends the input.)

-- Results --
be          3
not         2
or          1
ought       1
question    1
the         1
to          3
true.
```

Input handling for **tally**

read_line_to_codes produces a list of ASCII character codes for a line of input.

```
?- read_line_to_codes(user_input, Codes).
|: ab CD 12
Codes = [97, 98, 32, 67, 68, 32, 49, 50].

?- read_line_to_codes(user_input, Codes).
|: (hit ENTER)
Codes = [].
```

atom_codes can be used to form an atom from a list of codes.

```
?- atom_codes(Atom, [97, 98, 10, 49, 50]).
Atom = 'ab\n12'.
```

readline reads a line and produces an atom.

```
readline(Line) :-
    read_line_to_codes(user_input, Codes),
    atom_codes(Line, Codes).

?- readline(Line).
|: a test of this
Line = 'a test of this'.
```

Counting words

Let's use **word(Word, Count)** facts to maintain counts.

Let's write a **count(Word)** predicate to create and update **word/2** facts.

Example of operation:

```
?- retractall(word(_,_)).
true.

?- count(test).
true.

?- word(W,C).
W = test,
C = 1.

?- count(this), count(test), count(now).
true.

?- findall(W-C, word(W,C), L).
L = [this-1, test-2, now-1].
```

count implementation

For reference:

```
?- retractall(word(_,_)).

?- count(test), count(this), count(test), count(now).

?- findall(W-C, word(W,C), L).
L = [this-1, test-2, now-1].
```

Problem: Implement the predicate **count**.

```
count(Word) :-
    word(Word,Count0),
    retract(word(Word,_)),
    Count is Count0+1,
    assert(word(Word,Count)), !.

count(Word) :- assert(word(Word,1)).
```

Top-level and a helper

tally clears the counts then loops, reading lines and processing each.

```
    tally :-
       retractall(word(_,_)),
       repeat,
          readline(Line),
          do_line(Line),
          Line == '', !,              % note that '' is an empty atom
          show_counts.
```

How does **tally** terminate?

do_line breaks up a line into words and calls **count** on each word.

```
    do_line('').
    do_line(Line) :-
          atomic_list_concat(Words, ' ', Line),   % splits Line on blanks
          member(Word, Words),
          count(Word), fail.
    do_line(_).
```

Showing the counts

keysort/2 sorts a list of **A-B** structures on the value of the **A** terms.

```
?- keysort([zoo-3, apple-1, noon-4],L).
L = [apple-1, noon-4, zoo-3].
```

With **keysort** in hand we're ready to write **show_counts**.

```
show_counts :-
    writeln('\n-- Results --'),
    findall(W-C, word(W,C), Pairs),
    keysort(Pairs, Sorted),
    member(W-C, Sorted),
    format('~w~t~12|~w~n', [W,C]), fail.
show_counts.
```

```
-- Results --
be          3
not         2
or          1
ought       1
question    1
the         1
to          3
```

Full source is in **tally.pl**

Facts vs. Java maps, Ruby hashes, etc.

What's a key difference between using Prolog facts and maps/hashes/etc. to maintain word counts?

A hash or map can be passed around as a value, but Prolog facts are fundamentally objects with global scope. The collection of **word/2** facts can be likened to a Ruby global, like **$words = {}**

If we wanted to maintain multiple tallies simultaneously we could add an id of some sort to **word** facts.

Example: We might tally word counts for quotations in a document separately from word counts for body content. Calls to **count** might look like this,

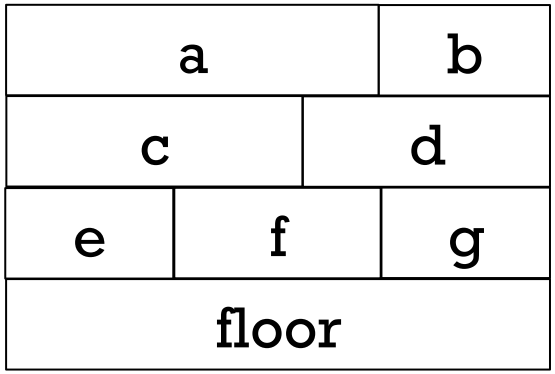
```
    count(Type, Word)
```

and create facts like these:

```
    word(quotes, testing, 3)
    word(body, testing, 10)
```

Example: Unstacking blocks

Consider a stack of blocks, each of which is uniquely labeled with a letter:



This arrangement could be represented with these facts:

```
on(a,c).    on(c,e).    on(e,floor).
on(a,d).    on(c,f).    on(f,floor).
on(b,d).    on(d,f).    on(g,floor).
            on(d,g).
```

Problem: Define a predicate **clean** that will print a sequence of blocks to remove from the floor such that no block is removed until nothing is on it.

What's suitable sequence of removals for the above diagram?

a, c, e, b, d, f, g

Another: a, b, c, d, e, f, g.

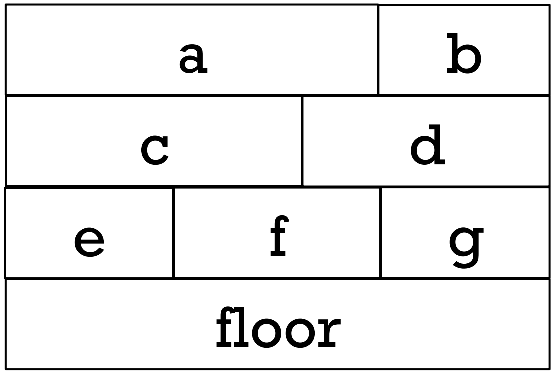
Unstacking blocks, continued

Here's one solution: (**blocks.pl**)

```
removable(B) :- \+on(_,B).

remove(B) :-
    removable(B),
    retractall(on(B,_)),
    format('Remove ~w\n', B).

remove(B) :-
    on(Above,B),
    remove(Above),
    remove(B).

clean :- on(B,floor), remove(B), clean, !.
clean :- \+on(_,_).
```

How long would in be in Java or Ruby?

Can we tighten it up?



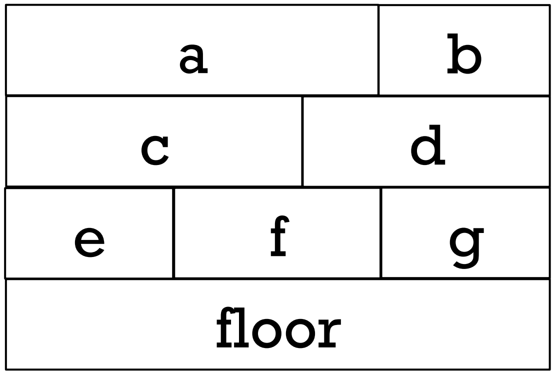
```
on(a,c).  on(a,d).  on(b,d). ...
```

```
?- clean.
Remove a
Remove c
Remove e
Remove b
Remove d
Remove f
Remove g
true.
```

Unstacking blocks, continued

A more concise solution:

```
clean :-
        on(Block,_), \+on(_,Block),
        format('Remove ~w\n', Block),
        retractall(on(Block,_)), clean, !.

clean :- \+on(_,_).
```



a b

c d

e f g

floor

```
on(a,c).  on(a,d).  on(b,d). ...
```

Output:

```
?- clean.
Remove a
Remove b
Remove c
Remove d
Remove e
Remove f
Remove g
true.
```

Previous sequence:

```
?- clean.
Remove a
Remove c
Remove e
Remove b
Remove d
Remove f
Remove g
true.
```

Find a block that's on something and that has nothing on it, and remove it.

Recurse, continuing as long as there's a block that's on something.

Pit-crossing Puzzle

The problem

Consider the problem of crossing over a series of pits using wooden planks as bridges.

Here's a case with two pits:

```
----+      +--+  +-----
    |      |  |  |
    |      |  |  |
    +------+  +--+
    5     12  15 18
```

Pits are represented with **pit/2** facts, with a starting position and a width:

```
pit(5,7)          % Think of the interval as [5,12).
pit(15,3)
```

There may be any number of **pit** facts. Pits never overlap. Pits always have some ground between them.

The problem, continued

Here's a crossing of distance 20 with the sequence of planks [3, 10, 10]:

```
===          ==========
   ==========
----+      +--+  +-------
    |      |  |  | ^
    |      |  |  | 20
    +------+  +--+
    5     12  15 18
```

Planks are drawn with vertical offsets to show their widths.

- Planks must be placed so that both ends rest on solid ground, rather than having an end over a pit.

- Planks must extend continuously from a starting point to (or through) a specified length.

The problem, continued

Here's an invalid crossing, with the sequence [9,11]:

```
=========
         ===========
----+      +--+  +----
    |      |  |  | ^
    |      |  |  | 20
    +------+  +--+
    5     12  15 18
```

[11,9] and [16,9] are invalid crossings, too.

It's invalid because the two planks meet over a pit, at distance 9.

- A joint at distance **D** is considered to be over a pit if
  start-of-pit <= D < end-of-pit

- Examples of over-pit distances for the above pits are 6, 10, and 17.

- Valid joint starting positions include 4, 13, 14, and 19.

The problem, continued

For reference, with two pits: **pit(5,7)** and **pit(15,3)**:

```
----+      +--+  +-----
    |      |  |  |
    |      |  |  |
    +------+  +--+
    5     12  15 18
```

Our task is to write **cross(+Distance, +Planks, -Solution)**.

```
?- cross(20, [10,10,3], S).
S = [3, 10, 10] .

?- cross(20, [9,11], S).
false.

?- cross(20, [1,2,4,5,5,9], S).
S = [4, 9, 1, 5, 2] .
```

Distance **D** is over a pit if
pit-start <= D < pit-end

layplanks

At hand:

```
?- cross(20, [1,2,4,5,5,9], S).
S = [4, 9, 1, 5, 2] .
```

Let's start with a helper predicate:

```
layplanks(+Goal, +Supply, +Current, -Solution)
```

- It succeeds if we can reach from the **Current** distance to the **Goal** with the given **Supply** of planks.
- **Solution** is instantiated to a suitable sequence of planks.

layplanks will be recursive. What's the base case?

```
layplanks(Goal, _, Current, []) :- Current >= Goal.
```

If we're at or past the goal distance, it takes no planks to reach the goal distance.

```
?- layplanks(10, [3,1,5], 12, S).
S = [] .
```

layplanks, continued

What should happen with a **layplanks** call like the following?

```
?- layplanks(20, [10,8,3], 0, S).
```

Pick a plank and see if we can add it to the solution.

- If so, then solve from the new distance with the remaining planks
- If not, pick a different plank.
- If no plank works, fail.

```
----+      +--+  +----
    |      |  |  |
    +------+  +--+
    5     12  15 18
```

What if we pick **10**?

We're over a pit!

What if we pick **3**?

We're not over a pit, so we lay down the plank and see if we can go from **3** to **20** with the remaining planks.

```
?- layplanks(20, [10,8], 3, S).
S = [10, 8]
```

layplanks, continued

For reference:

Pick a plank and see if we can add it to the solution.

- If so, then solve from the new state with the remaining planks
- If not, pick a different plank.
- If no plank works, fail.

```
----+      +--+  +----
    |      |  |  |
    +------+  +--+
    5     12  15 18
```

Current state:

?- layplanks(20, [10,8], 3, S).

What if we pick 8?

We're over a pit! (3+8 == 11)

What if we pick 10?

?- layplanks(20, [8], 13, S).

Picks 8 and does

?- layplanks(20, [], 21, S).

S = [] .

The full sequence via a spy point

We can see the sequence of calls and returns with a *spy point*:

```
?- spy(layplanks).
% Spy point on layplanks/4
true.

[debug]  ?- layplanks(20, [10,8,3], 0, S).
 * Call: (7) layplanks(20, [10, 8, 3], 0, _G1566)? leap
 * Call: (8) layplanks(20, [10, 8], 3, _G1646)? leap
 * Call: (9) layplanks(20, [8], 13, _G1658)? leap
 * Call: (10) layplanks(20, [], 21, _G1664)? leap
 * Exit: (10) layplanks(20, [], 21, [])? leap
 * Exit: (9) layplanks(20, [8], 13, [8])? leap
 * Exit: (8) layplanks(20, [10, 8], 3, [10, 8])? leap
 * Exit: (7) layplanks(20, [10, 8, 3], 0, [3, 10, 8])?l
S = [3, 10, 8]
```

Note that once the recursion hits the base case, the solution is built tail-first as the recursive calls to **layplanks** return.

Writing **layplanks**

layplanks needs to pick a plank and know which planks are left.

We'll use the built-in **select** for that:

```
select(?Elem, ?List1, ?List2)
   Is true when List1, with Elem removed, results in List2.
```

Example:

```
?- select(Plank, [10,8,3], Remaining).
Plank = 10,
Remaining = [8, 3] ;
Plank = 8,
Remaining = [10, 3] ;
Plank = 3,
Remaining = [10, 8] ;
false.
```

An implementation of **select**:

```
select(X, [X|T], T).
select(X, [H|T], [H|N]) :- select(X, T, N).
```

Example: layplanks(20, [10,8,3], 0, S).

Recall our base case:

```
layplanks(Goal,_,Current,[]) :- Current >= Goal.
```

Now we're ready to write the recursive case:

```
layplanks(Distance, Planks, Current, [Plank|MorePlanks]) :-
    % Pick a plank.
    select(Plank, Planks, Remaining),

    % See how far it extends.
    NewEnd is Current + Plank,

    % Be sure we're not over a pit.
    \+over_pit(NewEnd),  % todo!

    % Solve it from here with the remaining planks.
    layplanks(Distance, Remaining, NewEnd, MorePlanks).
```

Loose ends

Here's **over_pit**:

```
over_pit(N) :-
    pit(Start,Width),
    End is Start + Width,
    N >= Start, N < End.
```

Finally, **cross** calls **layplanks** with a current distance of zero to get things started:

```
cross(Goal, Planks, Solution) :-
     layplanks(Goal, Planks, 0, Solution).
```

```
?- cross(20, [1,2,4,5,5,9], S).
S = [4, 9, 1, 5, 2] .
```

```
====         =     ==
    ========= =====
----+      +--+  +----
    |      |  |  |
    +------+  +--+
    5     12  15 18
```

Experiment with this! It's in **cross.pl**.

Backtracking makes this work!

Key point:

A failure when attempting to place the very last plank may cause backtracking across predicate calls all the way back through the choice of the first plank!

Here's the general pattern for problems involving finding a valid sequence of parts, steps, movements, etc.:

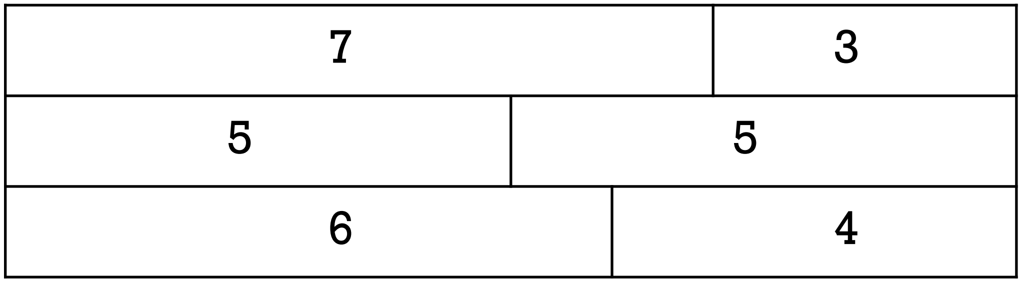
- Pick one of the things to add to the solution
- If it can be added, compute the new state.
  - If it can't be added, pick a different thing, or fail.
- Solve it from the new state with the remaining things

Note that **cross** isn't very smart. It didn't even check to see if we had enough planks to go the full distance, irrespective of the pits.

Brick laying puzzle

Brick laying

Consider six bricks of lengths 7, 5, 6, 4, 3, and 5. One way they can be laid into three rows of length 10 is like this:



7 3

5 5

6 4

Problem: Write a predicate **laybricks** that produces a suitable sequence of bricks for three rows of a given length:

```
?- laybricks([7,5,6,4,3,5], 10, Rows).
Rows = [[7, 3], [5, 5], [6, 4]] ;
Rows = [[7, 3], [5, 5], [4, 6]] ;
Rows = [[7, 3], [6, 4], [5, 5]] .

?- laybricks([7,5,6,4,3,5], 12, Rows).
false.
```

In broad terms, how can we approach this problem?

Helper **layrow**

layrow produces a sequence of bricks for a row of a given length:

```
?- layrow([3,2,7,4], 7, BricksLeft, Row).
BricksLeft = [2, 7],
Row = [3, 4] ;

BricksLeft = [3, 2, 4],
Row = [7] ;

BricksLeft = [2, 7],
Row = [4, 3] ;
false.
```

Implementation:

```
 layrow(Bricks, 0, Bricks, []).    % A row of length zero consists of no
                                   % bricks and doesn't touch the supply.

 layrow(Bricks, RowLen, Left, [Brick|MoreBricksForRow]) :-
         select(Brick, Bricks, Left0),
         RemLen is RowLen - Brick, RemLen >= 0,
         layrow(Left0, RemLen, Left, MoreBricksForRow).
```

Three rows of bricks

Let's write **lay3rows**, which is hardwired for three rows:

```
lay3rows(Bricks, RowLen, [Row1,Row2,Row3]) :-
    layrow(Bricks,      RowLen,   LeftAfter1,  Row1),
    layrow(LeftAfter1,  RowLen,   LeftAfter2,  Row2),
    layrow(LeftAfter2,  RowLen,   LeftAfter3,  Row3),
    LeftAfter3 = [].
```

Usage:

```
?- lay3rows([2,1,3,1,2], 3, Rows).
Rows = [[2, 1], [3], [1, 2]] ;
...
Rows = [[2, 1], [1, 2], [3]] ;
...
```

What is the purpose of **LeftAfter3 = []**?

How can we generalize it to N rows?

N rows of bricks

laybricks(+Bricks, +NRows, +RowLen, ?Rows) works like this:

```
?- laybricks([5,1,6,2,3,4,3], 3, 8, Rows).
Rows = [[5, 3], [1, 4, 3], [6, 2]] .

?- laybricks([5,1,6,2,3,4,3], 8, 3, Rows).
false.

?- laybricks([5,1,6,2,3,4,3], 2, 12, Rows).
Rows = [[5, 1, 6], [2, 3, 4, 3]] .

?-  laybricks([5,1,6,2,3,4,3], 4, 6, Rows).
Rows = [[5, 1], [6], [2, 4], [3, 3]] .
```

Implementation:

```
laybricks([], 0, _, []).

laybricks(Bricks, Nrows, RowLen, [Row|Rows]) :-
    layrow(Bricks, RowLen, BricksLeft, Row),
    RowsLeft is Nrows - 1,
    laybricks(BricksLeft, RowsLeft, RowLen, Rows).
```

N rows of bricks, continued

At hand:

```
laybricks([], 0, _, []).

laybricks(Bricks, Nrows, RowLen, [Row|Rows]) :-
        layrow(Bricks, RowLen, BricksLeft, Row),
        RowsLeft is Nrows - 1,
        laybricks(BricksLeft, RowsLeft, RowLen, Rows).
```

laybricks requires that all bricks be used. How can we remove that requirement?

Modify the base case:

```
laybricks(_, 0, _, []).

?- laybricks([4,3,2,1], 2, 3, Rows).
Rows = [[3], [2, 1]] .
```

We'll call this variant **laybricks2**.

Testing

Some facts for testing:

`laybricks(+Bricks, +NRows, +RowLen, ?Rows)`

```
b(1, [7,5,6,4,3,5]).
b(2, [5,1,6,2,3,4,3]).
b(3, [8,5,1,4,6,6,2,3,4,3,3,6,3,8,6,4]).   % 6x12
b(4, [8,5,1,4,6,6,2,3,4,3,3,6,3,8,6,4,1]). % 6x12 with extra 1
```

We can query **b(*N*, Bricks)** to set **Bricks** to a particular list.

```
?- b(1,Bricks), laybricks(Bricks, 2, 10, Rows).
false.

?- b(1,Bricks), laybricks2(Bricks, 2, 10, Rows). % laybricks2
Bricks = [7, 5, 6, 4, 3, 5],
Rows = [[7, 3], [5, 5]] .

?- b(3,Bricks), laybricks(Bricks,6,12,Rows).
Bricks = [8, 5, 1, 4, 6, 6, 2, 3, 4|...],
Rows = [[8, 1, 3], [5, 4, 3], [6, 6], [2, 4, 3, 3], [6, 6], [8, 4]] .
```

Testing, continued

Let's try a set of bricks that can't be laid into six rows of twelve:

```
?- b(4,Bricks), laybricks(Bricks,6,12,Rows).
...[the sound of a combinatorial explosion]...
^CAction (h for help) ? abort
% Execution Aborted

?- statistics.
8.240 seconds cpu time for 74,996,337 inferences
...
true.
```

The speed of a Prolog implementation is sometimes quoted in LIPS—logical inferences per second.

2006 numbers, for contrast:

```
?- statistics.
8.05 seconds cpu time for 25,594,610 inferences
```

The Zebra Puzzle

The Zebra Puzzle

The Wikipedia entry for "Zebra Puzzle" presents a puzzle said to have been first published in the magazine *Life International* on December 17, 1962. The facts:

- There are five houses.
- The Englishman lives in the red house.
- The Spaniard owns the dog.
- Coffee is drunk in the green house.
- The Ukrainian drinks tea.
- The green house is immediately to the right of the ivory house.
- The Old Gold smoker owns snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house.
- The man who smokes Chesterfields lives in the house next to the man with the fox.
- Kools are smoked in the house next to the house where the horse is kept.
- The Lucky Strike smoker drinks orange juice.
- The Japanese smokes Parliaments.
- The Norwegian lives next to the blue house.

The article asked readers, "Who drinks water? Who owns the zebra?"

Zebra Puzzle, continued

We can solve this problem by representing all the information with a set of goals and asking Prolog to find the condition under which all the goals are true.

A good starting point is these three facts:

- There are five houses.
- The Norwegian lives in the first house.
- Milk is drunk in the middle house.

Those facts can be represented as this goal:

```
Houses = [house(norwegian, _, _, _, _),        % First house
          _,                                    % Second house
          house(_, _, _, milk, _),              % Middle house
          _, _]                                 % 4th and 5th houses
```

Instances of **house** structures represent knowledge about a house.

house structures have five terms: nationality, pet, smoking preference (remember, it was 1962!), beverage of choice and house color.

Anonymous variables are used to represent "don't-knows".

Zebra Puzzle, continued

Some facts can be represented with goals that specify structures as members of the list **Houses**, but with unknown position:

The Englishman lives in the red house.

```
member(house(englishman, _, _, _, red), Houses)
```

The Spaniard owns the dog.

```
member(house(spaniard, dog, _, _, _), Houses)
```

Coffee is drunk in the green house.

```
member(house(_, _, _, coffee, green), Houses)
```

How can we represent *The green house is immediately to the right of the ivory house.*?

Zebra Puzzle, continued

At hand:

The green house is immediately to the right of the ivory house.

Here's a predicate that expresses left/right positioning:

```
left_right(L, R, [L, R | _]).
left_right(L, R, [_ | Rest]) :- left_right(L, R, Rest).
```

Testing:

```
?- left_right(Left,Right, [1,2,3,4]).
Left = 1,
Right = 2 ;

Left = 2,
Right = 3 ;
...
```

Goal:
```
left_right(house(_, _, _, _, ivory),
           house(_, _, _, _, green), Houses)
```

Zebra Puzzle, continued

We have these "next to" facts:

- The man who smokes Chesterfields lives in the house next to the man with the fox.
- Kools are smoked in the house next to the house where the horse is kept.
- The Norwegian lives next to the blue house.

How can we represent these?

We can say that two houses are next to each other if one is immediately left or right of the other:

```
next_to(X, Y, List) :- left_right(X, Y, List).
next_to(X, Y, List) :- left_right(Y, X, List).
```

Zebra Puzzle, continued

These "next to" facts are at hand:

- The man who smokes Chesterfields lives in the house next to the man with the fox.
- Kools are smoked in the house next to the house where the horse is kept.
- The Norwegian lives next to the blue house.

The facts above expressed as goals:

```
next_to(house(_, _, chesterfield, _, _),
        house(_, fox, _, _, _), Houses)

next_to(house(_, _, kool, _, _),
        house(_, horse, _, _, _), Houses)

next_to(house(norwegian, _, _, _, _),
        house(_, _, _, _, blue), Houses)
```

Zebra Puzzle, continued

A few more simple **house** & **member** goals complete the encoding:

- The Ukrainian drinks tea.

```
member(house(ukrainian, _, _, tea, _), Houses)
```

- The Old Gold smoker owns snails.

```
member(house(_, snails, old_gold, _, _), Houses)
```

- Kools are smoked in the yellow house.

```
member(house(_, _, kool, _, yellow), Houses)
```

- The Lucky Strike smoker drinks orange juice.

```
member(house(_, _, lucky_strike, orange_juice, _),
    Houses)
```

- The Japanese smokes Parliaments.

```
member(house(japanese, _, parliment, _, _), Houses)
```

Zebra Puzzle, continued

A rule that comprises all the goals:

```
zebra(Zebra_Owner, Water_Drinker) :-
  Houses = [house(norwegian, _, _, _, _), _,
                    house(_, _, _, milk, _), _, _],
  member(house(englishman, _, _, _, red), Houses),
  member(house(spaniard, dog, _, _, _), Houses),
  member(house(_, _, _, coffee, green), Houses),
  member(house(ukrainian, _, _, tea, _), Houses),
  left_right(house(_,_,_,_,ivory), house(_,_,_,_,green), Houses),
  member(house(_, snails, old_gold, _, _), Houses),
  member(house(_, _, kool, _, yellow), Houses),
  next_to(house(_,_,chesterfield,_,_),house(_, fox,_,_,_), Houses),
  next_to(house(_,_,kool,_,_), house(_, horse, _, _, _), Houses),
  member(house(_, _, lucky_strike, orange_juice, _), Houses),
  member(house(japanese, _, parliment, _, _), Houses),
  next_to(house(norwegian,_,_,_,_), house(_,_,_,_, blue), Houses),
  member(house(Zebra_Owner, zebra, _, _, _), Houses),
  member(house(Water_Drinker, _, _, water, _), Houses).
```

Note that the last two goals ask the questions of interest.

Zebra Puzzle, continued

The moment of truth:

```
?- zebra(_, Zebra_Owner, Water_Drinker).
Zebra_Owner = japanese,
Water_Drinker = norwegian ;
false.
```

The whole neighborhood:

```
?- zebra(Houses,_,_), member(H,Houses), writeln(H), fail.
house(norwegian,fox,kool,water,yellow)
house(ukrainian,horse,chesterfield,tea,blue)
house(englishman,snails,old_gold,milk,red)
house(spaniard,dog,lucky_strike,orange_juice,ivory)
house(japanese,zebra,parliment,coffee,green)
false.
```

Credit: The code above was adapted from **sandbox.rulemaker.net/ngps/119**, by Ng Pheng Siong, who in turn apparently adapted it from work by Bill Clementson in Allegro Prolog.

Odds and ends

Collberg's *Architecture Discovery Tool*

In the mid-1990s Dr. Collberg developed a system that is able to discover the instruction set, registers, addressing modes and more for a machine given only a C compiler for that machine.

The basic idea:

> Use the C compiler of the target system to compile a large number of small but carefully crafted programs and then examine the machine code produced for each program to make inferences about the architecture.

End result:

> A machine description that in turn can be used to generate a code generator for the architecture.

The system is written in Prolog. What makes Prolog well-suited for this task?

Paper: **http://www.cs.arizona.edu/~collberg/content/research/papers/collberg02automatic.pdf**

The Prolog 1000

The Prolog 1000 is a compilation of applications written in Prolog and related languages. Here is a sampling of the entries:

AALPS

The Automated Air Load Planning System provides a flexible spatial representation and knowledge base techniques to reduce the time taken for planning by an expert from weeks to two hours. It incorporates the expertise of loadmasters with extensive cargo and aircraft data.

ACACIA

A knowledge-based framework for the on-line dynamic synthesis of emergency operating procedures in a nuclear power plant.

ASIGNA

Resource-allocation problems occur frequently in chemical plans. Different processes often share pieces of equipment such as reactors and filters. The program ASIGNA allocates equipment to some given set of processes. (2,000 lines)

The Prolog 1000, continued

Coronary Network Reconstruction

The program reconstructs a three-dimensional image of coronary networks from two simultaneous X-Ray projections. The procedures in the reconstruction-labelling process deal with the correction of distortion, the detection of center-lines and boundaries, the derivation of 2-D branch segments whose extremities are branching, crossing or end points and the 3-D reconstruction and display.

All algorithmic components of the reconstruction were written in the C language, whereas the model and resolution processes were represented by predicates and production rules in Prolog. The user interface, which includes a main panel with associated control items, was developed using Carmen, the Prolog by BIM user interface generator.

DAMOCLES

A prototype expert system that supports the damage control officer aboard Standard frigates in maintaining the operational availability of the vessel by safeguarding it and its crew from the effects of weapons, collisions, extreme weather conditions and other calamities. (> 68,000 lines)

The Prolog 1000, continued

DUST-EXPERT

Expert system to aid in design of explosion relief vents in environments where flammable dust may exist. (> 10,000 lines)

EUREX

An expert system that supports the decision procedures about importing and exporting sugar products. It is based on about 100 pages of European regulations and it is designed in order to help the administrative staff of the Belgian Ministry of Economic Affairs in filling in forms and performing other related operations. (>38,000 lines)

GUNGA CLERK

Substantive legal knowledge-based advisory system in New York State Criminal Law, advising on sentencing, pleas, lesser included offenses and elements.

MISTRAL

An expert system for evaluating, explaining and filtering alarms generated by automatic monitoring systems of dams. (1,500 lines)

The full list is in `prolog/Prolog1000.txt`. Several are over 100K lines of code.

Lots of Prologs

For a Fall 2006 honors section assignment Maxim Shokhirev was given the task of finding as many Prolog implementations as possible in <u>one hour</u>. His results:

1. DOS-PROLOG
http://www.lpa.co.uk/dos.htm
2. Open Prolog
http://www.cs.tcd.ie/open-prolog/
3. Ciao Prolog
http://www.clip.dia.fi.upm.es/Software/Ciao
4. GNU Prolog
http://pauillac.inria.fr/~diaz/gnu-prolog/
5. Visual Prolog (PDC Prolog and Turbo Prolog)
http://www.visual-prolog.com/
6. SWI-Prolog
http://www.swi-prolog.org/
7. tuProlog
http://tuprolog.alice.unibo.it/
8. HiLog
ftp://ftp.cs.sunysb.edu/pub/TechReports/kifer/hilog.pdf
9. ?Prolog
http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/
10. F-logic
http://www.cs.umbc.edu/771/papers/flogic.pdf
11. OW Prolog
http://www.geocities.com/owprologow/
12. FLORA-2
http://flora.sourceforge.net/
13. Logtalk
http://www.logtalk.org/
14. WIN Prolog
http://www.lpa.co.uk/
15. YAP Prolog
http://www.ncc.up.pt/~vsc/Yap
16. AI::Prolog
http://search.cpan.org/~ovid/AI-Prolog-0.734/lib/AI/Prolog.pm
17. SICStus Prolog
http://www.sics.se/sicstus/
18. ECLiPSe Prolog
http://eclipse.crosscoreop.com/
19. Amzi! Prolog
http://www.amzi.com/
20. B-Prolog
http://www.probp.com/
21. MINERVA
http://www.ifcomputer.co.jp/MINERVA/
22. Trinc Prolog
http://www.trinc-prolog.com/

<u>**And 50 more!**</u>

Ruby meets Prolog

http://www.artima.com/forums/flat.jsp?forum=123&thread=182574 describes a "tiny Prolog in Ruby".

Here is **member**:

```
member[cons(:X,:Y), :X].fact
member[cons(:Z,:L), :X] <<= member[:L,:X]
```

Here's the common family example:

```
sibling[:X,:Y] <<= [parent[:Z,:X], parent[:Z,:Y], noteq[:X,:Y]]
parent[:X,:Y] <<= father[:X,:Y]
parent[:X,:Y] <<= mother[:X,:Y]

# facts: rules with "no preconditions"
father["matz", "Ruby"].fact
mother["Trude", "Sally"].fact
...

query sibling[:X, "Sally"]
# >> 1 sibling["Erica", "Sally"]
```

In conclusion...

If we had a whole semester...

- Parsing with definite clause grammars (slides 273-289 in the PDF)
- More with...
  - Puzzle solving
  - Higher-order predicates
- Expert systems
- Natural language processing
- Constraint programming
- Look at Prolog implementation with the Warren Abstract Machine.

Continued study:

More in Covington and Clocksin & Mellish.
The Craft of Prolog by O'Keefe
The Art of Prolog by Sterling and Shapiro

Parsing and grammars

Credit: The first part of this section borrows heavily from chapter 12 in Covington.

A very simple grammar

Here is a grammar for a very simple language. It has four *productions*.

```
Sentence    => Article Noun Verb

Article     => the | a

Noun        => dog | cat | girl | boy

Verb        => ran | talked | slept
```

Here are some sentences in the language:

the dog ran
a boy slept
the cat talked

the, dog, cat, etc. are *terminal symbols*—they appear in the strings of the language. Generation terminates with them.

Sentence, Article, Noun and Verb are *non-terminal symbols*—they can produce something more.

Sentence is the *start symbol*. We can generate sentences by starting with it and replacing non-terminals with terminals and non-terminals until only terminals remain.

A very simple parser

Here is a simple parser for the grammar, expressed as clauses: (**parser0.pl**)

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).

article([the| Left], Left).
article([a| Left],  Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left)   :- member(Verb, [ran,talked,slept]).
```

```
Sentence => Article Noun Verb
Article     => the | a
Noun        => dog | cat | girl | boy
Verb        => ran | talked | slept
```

Usage:

```
?- sentence([the,dog,ran]).
true .

?- sentence([the,dog,boy]).
false.

?- sentence(S).        % Generates all valid sentences
S = [the, dog, ran] ;
S = [the, dog, talked] ;
S = [the, dog, slept] ;
...
```

A very simple parser, continued

For reference:

```
sentence(Words) :-
    article(Words, Left1), noun(Left1, Left2), verb(Left2, []).

article([the|Left], Left).
article([a| Left],  Left).
noun([Noun|Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left)  :- member(Verb, [ran,talked,slept]).
```

Note that the heads for **article**, **noun**, and **verb** all have the same form.

Let's look at a clause for **article** and a unification:

```
article([the|Left], Left).

?- article([the,dog,ran], Remaining).
Remaining = [dog, ran] .
```

If **Words** begins with **the** or **a**, then **article(Words, Remaining)** succeeds and unifies **Remaining** with the rest of the list. The key idea: **article**, **noun**, and **verb** each consume an expected word and produce the remaining words.

A very simple parser, continued

```
sentence(Words) :-
    article(Words, Left1), noun(Left1, Left2), verb(Left2, []).
```

A query sheds light on how **sentence** operates:

```
    ?- article(Words, Left1), noun(Left1, Left2),
         verb(Left2, Left3), Left3 = [].
    Words = [the, dog, ran],
    Left1 = [dog, ran],
    Left2 = [ran],
    Left3 = [] .
    ?- sentence([the,dog,ran]).
    true .
```

Each goal consumes one word. The remainder is then the input for the next goal.

Why is **verb**'s result, **Left3**, unified with the empty list?

A very simple parser, continued

Here's a convenience predicate that splits up a string and calls **sentence**.

```
s(String) :-
    concat_atom(Words, ' ', String), sentence(Words).

sentence(Words) :-
    article(Words, Left1), noun(Left1, Left2), verb(Left2, []).
```

Usage:

```
?- s('the dog ran').
true .

?- s('ran the dog').
false.
```

Grammar rule notation

Prolog's *grammar rule notation* provides a convenient way to express these stylized rules. Instead of this,

```
sentence(Words) :-
     article(Words, Left0), noun(Left0, Left1), verb(Left1, []).
article([the| Left], Left).
article([a| Left],  Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

we can take advantage of grammar rule notation and say this,

```
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

This is Prolog source code, too!

Note that the literals (terminals) are specified as singleton lists.

The semicolon is an "or". Alternative: **noun --> [dog]. noun --> [cat]. ...**

Grammar rule notation, continued

```
$ cat parser1.pl
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

listing can be used to see the clauses generated for that grammar.

```
?- [parser1].
...

?- listing(sentence).
sentence(A, D) :- article(A, B), noun(B, C), verb(C, D).

?- listing(article).
article(A, B) :-
      (   A=[a|B]
      ;   A=[the|B]
      ).
```

Note that the predicates generated for **sentence**, **article** and others have an arity of 2.

Grammar rule notation, continued

At hand: (a *definite clause grammar*)

```
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].

?- listing(sentence).
sentence(A, D) :- article(A, B), noun(B, C), verb(C, D).

?- listing(article).
article(A, B) :- (A=[a|B];   A=[the|B]).

?- sentence([a,dog,talked,to,me], Leftover).
Leftover = [to, me] .

?- sentence([a,bird,talked,to,me], Leftover).
false.
```

Remember that **sentence**, **article**, **verb**, and **noun** are non-terminals. **dog**, **cat**, **ran**, **talked**, are terminals, represented as atoms in singleton lists.

Grammar rule notation, continued

Below we've added a second term to the call to **sentence**, and mixed in a regular rule for **verb** along with the grammar rule.

```
 s(String) :-                                          % parser1a.pl
      concat_atom(Words, ' ', String), sentence(Words,[]).

sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].

verb --> [ran]; [talked]; [slept].
verb([Verb|Left], Left) :- verb0(Verb).

verb0(jumped). verb0(ate). verb0(computed).

?- s('a boy computed').
true .

?- s('a boy computed pi').
false.
```

Goals in grammar rules

We can insert ordinary goals into grammar rules by enclosing the goal(s) in curly braces.

Here is a chatty parser that recognizes the language described by the regular expression a*:

```
parse(S) :- atom_chars(S,Chars), string(Chars, []). % parser6.pl

string --> as.

as --> [a], {writeln('got an a')}, as.
as --> [], {writeln('empty match')}.
```

Usage:

```
?- parse(aaa).
got an a
got an a
got an a
empty match
true .
```

```
?- parse(aab).
got an a
got an a
empty match
empty match
empty match
false.
```

What if the as clauses are swapped?

```
?- parse(aaa).
empty match
got an a
empty match
got an a
empty match
got an a
empty match
true.
```

Parameters in non-terminals

We can add parameters to the non-terminals in grammar rules. The following grammar recognizes `a*` and produces the length, too.

```
parse(S, Count) :-                         % parser6a.pl
    atom_chars(S,Chars), string(Count,Chars, []).

string(N) --> as(N).

as(N) --> [a], as(M), {N is M + 1}.
as(0) --> [].
```

Usage:

```
?- parse(aaa, N).
N = 3 .

?- parse(aaab, N).
false.
```

Parameters in non-terminals, continued

Here is a grammar that recognizes $\mathbf{a}^{N}\mathbf{b}^{2N}\mathbf{c}^{3N}$: (**parser7a.pl**)

```
parse(S,L) :- atom_chars(S,Chars), string(L, Chars, []).

string([N,NN,NNN]) -->
      as(N), {NN is 2*N}, bs(NN), {NNN is 3*N}, cs(NNN).

as(N) --> [a], as(M), {N is M+1}.
as(0) --> [].

bs(N) --> [b], bs(M), {N is M+1}.
bs(0) --> [].

cs(N) --> [c], cs(M), {N is M+1}.
cs(0) --> [].

?- parse(aabbbbcccccc, L).
L = [2, 4, 6] .

?- parse(aabbc, L).
false.
```

Can this language be described with a regular expression?

Parameters in non-terminals, continued

How could we handle $\mathbf{a}^X\mathbf{b}^Y\mathbf{c}^Z$ where X <= Y <= Z?

```
?- parse(abbbccc, L).
L = [1, 3, 3] .

?- parse(ccccc, L).
L = [0, 0, 5] .

?- parse(aaabbc, L).
false.
```

```
parse(S,L) :- atom_chars(S,Chars), string(L, Chars, []). % parser7b.pl

string([X,Y,Z]) --> as(X), bs(Y), {X =< Y}, cs(Z), {Y =< Z}.

as(N) --> [a], as(M), {N is M+1}.
as(0) --> [].

bs(N) --> [b], bs(M), {N is M+1}.
bs(0) --> [].

cs(N) --> [c], cs(M), {N is M+1}.
cs(0) --> [].
```

Accumulating an integer

Problem: Write a parser that recognizes a string of digits and creates an integer from them:

```
?- parse('4341', N).
N = 4341 .

?- parse('1x3', N).
false.
```

Solution:

```
parse(S,N) :-                                                % parser8.pl
     atom_chars(S, Chars), intval(N,Chars,[])~~, integer(N)~~.

intval(N) --> digits(Digits), { atom_number(Digits,N) }.

digits(Digit) --> [Digit], {digit(Digit)}.
digits(Digits) --> [Digit], {digit(Digit)},
         digits(More), {concat_atom([Digit,More],Digits)}.

digit('0'). digit('1'). digit('2').  ...
```

How do the **digits(...)** rules work?

A list recognizer

Consider a parser that recognizes lists consisting of positive integers and lists:

```
?- parse('[1,20,[30,[[40]],6,7],[]]').
true .

?- parse('[1,20,,[30,[[40]],6,7],[]]').
false.

?- parse('[ 1, 2 , 3 ]').  % Whitespace!  How could we handle it?
false.
```

Implementation: (**list.pl**)

```
parse(S) :- atom_chars(S, Chars), list(Chars, []).

list --> ['['], values, [']'].
list --> ['['], [']'].

values --> value.
values --> value, [','], values.

value --> digits(_).        % digits(...) from previous slide
value --> list.
```

"Real" compilation

These parsing examples are far short of what's done in a compiler. The first phase of compilation is typically to break the input into "tokens". Tokens are things like identifiers, individual parentheses, string literals, etc.

Input text like this,

```
[ 1, [30+400], 'abc']
```

might be represented as a stream of tokens with this Prolog list:

```
[lbrack, integer(1), comma, lbrack, integer(30), plus, integer(400),
rbrack, comma, atom(abc), rbrack]
```

The second phase of compilation is to parse the stream of tokens and generate code (traditional compilation) or execute it immediately (interpretation).

We could use a pair of Prolog grammars to parse source code:

- The first one would parse character-by-character and generate a token stream like the list above. (A *scanner*.)
- The second grammar would parse that token stream.

Edit this form

CSC 372 Survey / Assignment 1; due Tuesday, Jan 19 at 2:00pm

Thanks in advance for taking the time to work through all the questions below.

Several questions have a long list of checkboxes. If you click the text of your first selection, you can then use TAB to move through them, hitting a space to mark selections. Use SHIFT+TAB to move backwards. TAB can be used to move between questions, too.

Note that when you click the Submit button at the end you'll have a chance to save a link that you can use to revise your answers.

* Required

What is your name? (first last) *

What is your NetID? *

Your NetID is your @email.arizona.edu address, like "jsmith", NOT your numeric student ID.

What computer science or programming classes have you taken here or elsewhere? Use the "Other" box for any not listed or taken elsewhere. *

□ 127A

□ 127B

□ 196H

□ 227

□ 245

□ 252

□ 296

□ 335

□ 337

□ 345

□ 346

□ 352

□ 372

□ 391

□ 397C

□ 422

□ 425

□ 436

□ 437

□ 445

□ 447
□ 450
□ 452
□ 453
□ 460
□ 466
□ 473
□ 477
□ 496
□ ECE 175
□ ISTA 130
□ ISTA 303
□ ISTA 350
□ ISTA 403
□ Other: ______

Other than 372, what computer science or programming classes are you taking this semester? *
(Yes, some of these aren't offered this semester. I'm just too lazy to change this list each semester!)

□ 245
□ 252
□ 296
□ 335
□ 337
□ 345
□ 346
□ 352
□ 391
□ 397C
□ 422
□ 425
□ 433
□ 436
□ 437
□ 445
□ 452
□ 453
□ 460
□ 466
□ 473
□ 477
□ 496
□ ECE 175
□ ISTA 130

☐ ISTA 303

☐ ISTA 350

☐ No other CS courses this semester

☐ Other:

What's your preferred working environment? *

○ Windows 7

○ Windows 8

○ Windows 10

○ Mac OS X

○ Linux

○ Other:

If your preferred environment is Windows, do you have Cygwin or something similar installed?

○ yes

○ no

What text editors are in your "toolbox"? *

The editors of interest here are things like vim, Emacs, nano, Sublime, and Notepad++; NOT IDEs like Eclipse, and NOT document preparation tools like Microsoft Word.

What's your preferred option for developing solutions for this class? *

○ Work on CS department machines

○ Work on my laptop or home computer, installing whatever packages are needed

○ Some combination of the above

○ Other:

Do you have a laptop? (Count a Chromebook as "yes".) *

○ yes

○ no

Do you typically keep a laptop open during class? *

○ yes

○ no

○ it depends...

If we did in-class polls using a web app do you have a device you could easily use to participate in those polls? *

○ yes

○ no

What's your preference for the frequency of assignments? *

Assume the total amount of work required is the same for both cases.

○ An assignment every week

○ An assignment every two weeks

○ No preference

Would you rather have assignments regularly due on Thursday nights or Friday nights? *

○ Thursday nights

○ Friday nights

○ No preference

If there were *suggested* reading assignments, how likely would you be to do them? *

○ Very likely

○ Somewhat likely

○ Not very likely

What are the top three companies you'd like to work for someday? If you'd like to have your own company, say "self".

Where do you consider yourself to be "from"? *

Tell me at least three things about yourself. Some possibilities: current employment, near-term goals, post-graduation aspirations, hobbies, etc. *

Assuming you've seen the TV game show "Jeopardy!", what are three Jeopardy categories outside the realm of computer science that you'd do well at?

(In other words, what are three non-CS areas in which you consider yourself knowledgable?)

In what languages can you (the reader) write a program to print 1 through N?

Without consulting any documentation, nor asking others, nor Googling, etc., in what programming languages could you write a program to print the integers from 1 through N?

Assume N is supplied by the user in a way of your choice, such as a command line argument, read from keyboard or a file, a URL parameter, or from a GUI input element.

Assume you can edit/run/debug but there's no documentation whatsoever. Don't worry about error checking or corner cases; assume you're on the Happy Path.

NOTE: Your answer is to be a list of languages, like "Java, Python". You are NOT being asked to write any code!

*

In what languages can you (the reader) write a program to reverse the order of input lines?

Same rules as the previous question, but instead of printing 1-N, read lines from the keyboard and at end of file, print them out in reverse order: last line first, first line last (like tac(1), if you're familiar with that).

NOTE: Again, your answer is to be a list of languages, like "Java, Python". You are NOT being asked to write any code!

*

How comfortable are you with recursive functions/methods? *

○ Very comfortable. I fully understand the idea of recursive code.

○ I've written some recursive functions/methods but recursion is still a little sketchy to me.

○ I've never written a recursive function/method.

○ I don't know what "recursive function" means.

Self-rating of skills

For each of the following languages/technologies/systems/packages rate your skill on this 0-5 scale:

0: I've done nothing or almost nothing with it
1: I've read and/or experimented with it a little but spent probably less than a day with it
2: I can do a few things with it but there's lots I don't know
3: I'm in the middle, knowledge-wise
4: I am skilled with it
5: I consider myself to be an expert on it

If you used a language in the past but are rusty with it now, rate your skill as it was at your peak with it.

Programming in general *

0	1	2	3	4	5
○	○	○	○	○	○

Java *

0	1	2	3	4	5
○	○	○	○	○	○

C *

0	1	2	3	4	5
○	○	○	○	○	○

C++ (the part that's not in C) *

0	1	2	3	4	5
○	○	○	○	○	○

Haskell *

0	1	2	3	4	5
○	○	○	○	○	○

Ruby *

0	1	2	3	4	5
○	○	○	○	○	○

Prolog *

0	1	2	3	4	5
○	○	○	○	○	○

Python *

0	1	2	3	4	5
○	○	○	○	○	○

JavaScript *

0	1	2	3	4	5
○	○	○	○	○	○

PHP *

0	1	2	3	4	5
○	○	○	○	○	○

Objective-C *

0	1	2	3	4	5
○	○	○	○	○	○

Are you a "2" or better on any other programming languages?

If you'd rate yourself a 2 or better on any other programming languages, list them here and rate yourself on each.

UNIX command line tools *

0	1	2	3	4	5
○	○	○	○	○	○

Regular expressions *

0	1	2	3	4	5
○	○	○	○	○	○

Microsoft Windows (any version >= XP) *

0	1	2	3	4	5
○	○	○	○	○	○

Mac OS X *

0	1	2	3	4	5
○	○	○	○	○	○

Linux (any distro) *

0	1	2	3	4	5
○	○	○	○	○	○

Which of these programming languages have you previously heard of?

The following is a long list of programming languages. If you'd ever heard of the language BEFORE the first 372 lecture (or this survey), no matter whether you know anything at all about it, check the box for it. *

☐ ActionScript

□ Ada
□ ALGOL
□ APL
□ Arc
□ awk
□ BASIC
□ BCPL
□ "BF"
□ C
□ C++
□ C#
□ Ceylon
□ Clojure
□ COBOL
□ CoffeeScript
□ D
□ Dart
□ Delphi (Object Pascal)
□ Dylan
□ Eiffel
□ Erlang
□ F#
□ Forth
□ Fortran
□ Go
□ Groovy
□ Hack
□ Haskell
□ Icon
□ INTERCAL
□ JavaScript
□ LabVIEW
□ Leda
□ Limbo
□ Lisp
□ LOBOL
□ Lua
□ MATLAB
□ Miranda
□ ML
□ Objective-C
□ Pascal
□ perl

□ PHP

□ PL/I

□ Plankakul

□ Prolog

□ Python

□ R

□ Ratfor

□ Ruby

□ Rust

□ Scala

□ Scheme

□ Short Code

□ SIMULA

□ Smalltalk

□ SNOBOL4

□ SR

□ Swift

□ Tcl

□ WhereLQ

□ Y

□ None of the above

STRICTLY OPTIONAL, for those who have NOT had a class with me before: Tell me one thing you've heard about me. :)

If there's anything else you'd like to say at the moment, say it here! :)

Submit

Never submit passwords through Google Forms.

CSC 372, Spring 2016
Assignment 2
Due: Friday, January 29 at 23:59:59

Introduction

There's no programming at all on this assignment. It's a combination of web research (for two problems), creative thought, pondering, and a little writing.

For each problem you are to answer via a plain ASCII text file. Use an editor like Sublime, Vim, Emacs, Notepad++, etc. DO NOT submit Word documents, PDFs, Rich Text files, HTML documents, etc. As a double-check, your `.txt` files should look perfectly fine when displayed with `cat` on lectura, and the `file` command should show them as "`ASCII text`" or maybe "`ASCII English text`", or something similar.

Important: You'll be using a bash script on lectura to turn in your work. I recommend you give the submission process a try well before the deadline, in case you have trouble with it. See page 4.

Problem 1. (6 points) `morefacts.txt`

When covering slide 24 in the intro set (http://www.cs.arizona.edu/classes/cs372/spring16/intro.pdf) I said a sentence or two about various languages of interest. For this problem I'd like you to find three languages that are not mentioned on that slide and tell me a sentence or two about each.

I'll compile all the answers and post them on Piazza. Follow this format for your answers:

(1) The full response for each language should be a single line of text.
(2) Begin the line with the language's name followed by the year it appeared and then a colon, followed by one or more sentences with whatever you want to say.
(3) End each line with an attribution that is either your name or "anonymous".

As examples of both the format and the sort of thing I'm looking for, here are three of mine, one with anonymous attribution. I'll use `cat` to display my `morefacts.txt` and then `wc -l` to demonstrate it's only three lines:

```
% cat morefacts.txt
Ada 1980: The DoD's attempt to have one language for military embedded systems,
instead of 450. -- William Mitchell
Java 1995: The most rapidly adopted language of all time.   In Spring 1997 I gave
one lecture in 372 about Java as a rising language; by Fall 1998 it was being taught
in 127A. -- William Mitchell
Scala 2003: Proof that Germans should stick to beer and BMWs. -- anonymous
% wc -l morefacts.txt
3 morefacts.txt
```

Feel free to use Google, Wikipedia, etc., for research on this question but needless to say, no posts anywhere soliciting ideas.

Above I say, "the year it appeared" but that's often subject to debate. Feel free to believe Wikipedia or go with other sources.

Just to be clear, you may use anonymous attribution to keep your classmates from knowing you wrote a particular entry but what you submit must be original, not something you found on the net.

Problem 2. (6 points) `jp.txt`

Slide 35 in the intro set raises the idea of the philosophy of a language. In a nutshell, I think of the philosophy of a language as what it treats as important, or not. For this problem I'd like you to identify three elements of the philosophy of Java.

For this problem it's fine to brainstorm with CLASSMATES, Google for "what is the philosophy of java", etc., but what you submit must be stated in your own words.

A piece of "low-hanging fruit" in Java's philosophy that I'm hereby prohibiting you to use is support for object-oriented programming. If it were not prohibited, here's what you might have said about that element:

> *Java supports the object-oriented paradigm by providing classes and inheritance. The "abstract" keyword allows classes and methods to be marked as abstract. The "static" keyword, although poorly named, supports the concept of class variables and methods.*

Problem 3. (3 points) `measure.txt`

Slide 27 in the intro set cites some attempts to measure language popularity. Some use simple measures, such as new GitHub repositories and job postings. The TIOBE index is more complicated. http://www.code2015.com was a simple tweet-based survey.

For this problem I'd like you to invent another simple way to measure language popularity. For example you might say, "Stand at the intersection of Speedway and Campbell and count programming language-specific bumper stickers." That's not a bad first thought, but I'd be worried about a dearth of data points and not be inclined to award that idea full credit.

Along with describing your idea, mention any weaknesses you see with it. For example, a weakness with code2015.com was that it wasn't widely known. Another is that such polls are subject to inflation by promotion of it within user communities. (Just for fun: see if you see any code2015.com results that seem way out of line. There was a code 2014.com, too, with more participation.)

No web research or discussion with anybody else for this problem, please. It's just you and your brain on this one.

Any ideas that make me say "Wow!" will earn a point of extra credit.

If you should find yourself completely blank for an idea 48 hours prior to the deadline, ask me for a hint.

Problem 4. (2 points) `negative.txt`

Java's `String.charAt()` method allows strings to be indexed from the left end of the string but not the right end of the string—`s.charAt(0)` is the first character, but we need to write something like `s.charAt(s.length()-1)` to get the last character. In contrast, many languages interpret negative string indices as being right-relative: `-1` is the last character, `-2` is next to last, etc.

For this problem you are to imagine you're designing a new language and you are currently considering whether to support negative indexing for strings. Present an argument that's either in favor of negative indexing, or against it.

Problem 5. (1 point) `javarepl.txt`

http://www.javarepl.com is a REPL for Java. Spend a few minutes experimenting with it and write a few sentences about it. You might talk about what you liked, what you expected to work but didn't, what you'd like to change, etc.

Problem 6. (ZERO points) `quickrepl.txt`

After working with languages that have a REPL available, you may find yourself really wanting a REPL when you go to learn a language that doesn't have one. One approach to a REPL is to integrate it with a language's implementation, but that typically requires a good understanding of that implementation. Another approach is a completely standalone REPL, such as provided by javarepl.com, but that can require a lot of code—`wc` shows about 8,000 lines in Java source files for that system.

For this problem your challenge is to come up with a quick approach to write a simple REPL for a language. Your quick REPL needn't be nearly as good as `ghci` but it should be able do the sorts of things you do when learning a language, such as evaluating expressions and showing types. Think about an approach that provides a lot of functionality for relatively little effort—something you could implement in a day, and not be late for dinner.

I'm not asking you to write any code for this problem; just write a description of how you might quickly produce a simple REPL for a language of your choice.

Note: This is a hard problem that requires some creativity and cleverness, so I'm making it worth zero points, but I'm curious to see who'll do it anyway! For this problem you are free to work in any groups of any size, but for me to possibly be impressed with your answer you'll have to specifically state that you came up with whatever you did all by yourself/yourselves—no Googling.

If you work in a group, all members may submit identical copies of `quickrepl.txt`, but it should include list of group members.

Problem 7. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with "`Hours:`". There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Here's the full list of *deliverables* for this assignment

`morefacts.txt`
`jp.txt`
`measure.txt`
`negative.txt`
`javarepl.txt`
`quickrepl.txt` (zero points—it's optional!)
`observations.txt` (for extra credit)

Note that all characters in the file names are lowercase.

Do not include your name, NetID, etc. in your `.txt` files—I like reading answers without knowing who wrote them.

The bash script `/cs/www/classes/cs372/spring16/a2/turnin` (on lectura) should be used to turn in your work. It creates a time-stamped tar file that contains the expected files and then uses the system-wide `turnin` program to actually turn in that tar file.

I recommend that you create a symbolic link named `a2` that references the `/cs/www/classes/cs372/spring16/a2` directory. Here's a command that creates such a link:

```
ln -s /cs/www/classes/cs372/spring16/a2 .
```

(Yes, that final argument, a dot to specify that the symbolic link be made in the current directory, can be omitted.) If you haven't worked with symbolic links, slides 134+ in my 352 UNIX slides, on the Piazza resources page, talk about them.

Assuming you've made that symbolic link, you should see something like the following when you run `a2/turnin`:

```
% a2/turnin
Warning: quickrepl.txt not found
======== contents of a2.20160118.143419.tz ========
-rw-r--r-- whm/whm         521 2016-01-18 14:00 morefacts.txt
-rw-r--r-- whm/whm         692 2016-01-18 14:00 jp.txt
-rw-r--r-- whm/whm         535 2016-01-18 14:00 measure.txt
-rw-r--r-- whm/whm         880 2016-01-18 14:00 negative.txt
-rw-rw-r-- whm/whm         188 2016-01-18 14:00 javarepl.txt
-rw-r--r-- whm/whm         353 2016-01-18 14:00 observations.txt
======== running turnin ========
Turning in:
 a2.20160118.143419.tz -- ok
All done.
```

Note that the first line of output, "`Warning: quickrepl.txt not found`", indicates that a deliverable was not found. We'll imagine that since that `quickrepl.txt` is a zero-point problem, the student elected not to do it.

After that warning, the contents of the tar file, named `a2.20160118.143419.tz`, are shown. Finally, the system-wide `turnin` program is run and its output is shown.

You can run `a2/turnin` as often as you want. We'll grade your final non-late submission.

We can add a `-ls` option to `a2/turnin` to show what's been turned in:

```
% a2/turnin -ls
.:
total 4
-rw-rw---- 1 whm whm 1878 Jan 18 14:34 a2.20160118.143419.tz
```

If a submission is late, i.e., it shows a date of January 30 or later, it will be ignored unless you mail to `372s16` with some reason as to why it was late.

Miscellaneous

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond to 1% of your final grade in the course.

Remember that late assignments are not accepted, and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

CSC 372, Spring 2016
Assignment 3
Due: Friday, February 12 at 23:59:59

Introduction

Some of you may find this to be one of the hardest programming assignments you've ever had. Others may find it to be very easy. I recommend that you start on this assignment as soon as possible, in case you happen to fall into that first group. Remember that as the syllabus states, late work is not accepted and there are no "late days", but I will consider extensions for circumstances beyond your control.

The Haskell slides through 193 show you all the elements of Haskell you need for this assignment, but the two sections that follow in the slides, "Larger Examples" and "Errors" (ending around 230), will broaden your understanding.

I refer to assignments as "`a`*`N`*". This assignment is `a3`. This document is the "`a3` write-up".

My assignment write-ups are a combination of education, specification, and guidance. My goal is to produce write-ups that need no further specification or clarification. That goal is rarely achieved. If you have questions you can either mail to `372s16` or post to Piazza using the appropriate `aN` folder.

This assignment has a long preamble that covers a variety of topics, including the "Tester", a symlink that you need to create, some warnings about old solutions for recycled problems, a very important section on assignment-wide restrictions, and more. You might be inclined to skip all that stuff and get right to the problems, but I recommend you carefully read the preliminary material.

Use the "Tester"!

The syllabus says,

> *For programming problems great emphasis will be placed on the ability to deliver code whose output exactly matches the specification. Failing to achieve that will typically result in large point deductions, sometimes the full value of the problem.*

I'll provide a tester—a testing script—that you can use to confirm that the output of each of your solutions for the programming problems exactly matches the expected output for a number of test cases.

Don't just "eyeball" your output—use the tester! I won't have any sympathy for those who fail tests during grading simply because they didn't want to bother with the tester! However, we'll be happy to help you with using the tester and understanding its output.

The tester is described in the document *Using the Tester*, on the Piazza resources page.

Create a symbolic link to my `a3` directory

For each assignment there will be subdirectory of `/cs/www/classes/cs372/spring16` with assignment-specific files. Those directories will be named `a2`, `a3`, etc.

This write-up assumes your assignment 3 directory on lectura has an `a3` symlink (symbolic link) that references the directory `/cs/www/classes/cs372/spring16/a3`. For example, you'll run the tester with the script `a3/tester` and submit your work with `a3/turnin`.

Some of you may not have worked with symbolic links, so here's a little detail on what you need to do. First, let's imagine that you've made a `372` directory in your home directory, `/home/YOURNETID`, and that in that directory you've made an `a3` directory. (You can do both with "`mkdir -p ~/372/a3`".)

Next, go to your `372/a3` directory and then make a symlink to `spring16/a3`:

```
% cd ~/372/a3
% ln -s /cs/www/classes/cs372/spring16/a3
```

Then check the link with `ls`:

```
% ls -l a3
lrwxrwxrwx 1 whm whm 33 Jan 26 21:12 a3 -> /cw/www/classes/cs372/spring16/a3
```

That lowercase "L" at the start of the line indicates a symbolic link, and `a3 -> ...` shows what the symbolic link `a3` references. If you do "`ls a3`", "`cat a3/delivs`", etc., you'll actually be operating on `/cs/www/classes/cs372/spring16/a3` and files therein.

Let us know if you have trouble with this!

Don't post Piazza questions with pieces of solutions!

Just so there's no doubt about it, IT IS STRICTLY PROHIBITED TO POST PIECES OF SOLUTIONS, whether they work or not.

If you can boil code down to a generic question, like "Why doesn't "`f _ = _`" compile?" or "Is there a function that produces the maximum value of a list of numbers?", that's ok.

If code has any trace of the problem or it conveys or implies anything about a solution, DON'T POST IT. A good rule of thumb is this: If it's apparent what problem some code is related to, that code shouldn't be posted.

If you have even a minor concern that a Piazza post may reveal too much, mail to `372s16` instead!

Recycling of problems from past semesters

When I first started teaching here at The University of Arizona my aim was to come up with a great new set of problems for each assignment each time I taught a class. Maybe I just got old but I eventually found I couldn't keep up with that standard. I noticed that sometimes an old problem was simply better than a new one. I began to reconsider my "all new problems every semester" goal and started thinking about questions like these: Is there an ideal set of problems to teach a set of concepts? How should a teacher's time be balanced between writing new problems and other responsibilities, like working with students and improving lecture material? If creative efforts fail is it better to recycle a good problem or go with a new one that's lesser just so that students won't be tempted to consult an old solution? To make a long story short, I do recycle some problems from past semesters.

If you should come into possession of a solution from a past semester, whether it be from "homework help" site on the net, a friend who took the class, or some other source, let me encourage you to discard it! For one thing, I sometimes put *dunsels* in my solutions. I'd like to distribute solutions with perfectly clean and idiomatic code but I've found that an extra part that sticks out like a sore thumb (and that the lazy don't tend to notice!) helps me eliminate students who find and reuse my solutions. If you notice a silly extra in my code, try removing it and see if things still work. If so, you've perhaps found a dunsel!

While I'm talking about disincentives for cheating I'll also mention that I keep a copy of all former students' submissions for a given problem. We use tools like MOSS to look for suspicious similarities both in the set of this semester's submissions and in that set combined with submissions from any previous semesters when the problem was used.

Remember my cheating policy: one strike and you're out!

Don't forget what you already know about programming!

A key to success in this class is not forgetting everything you've learned about programming just because you're working in a new language. The skills you've learned in other classes for breaking problems down into smaller problems will serve you well in Haskell, too. On the larger problems, particularly `street` and `editstr`, look for small functions to write first that you can then build into larger functions. Test those functions one at a time, as they're written.

If you don't see the cause of a syntax or type error in an expression, whittle the code down until you do. Or, start with something simple and build towards the desired expression until it breaks.

You'll probably have some number of errors due to surprises with precedence. Adding parentheses to match the precedence you're assuming may reveal a problem.

I think of a bug as a divergence between expectation and reality. A key skill for programmers is being able to work backwards to find where that divergence starts. Here's an example of working backwards from an observed divergence to its source: Imagine a program whose expected output is a series of values but instead it produces no output. You then discover that it's producing no output because the count of values to print is zero. You then find that the count is zero because the argument parser is returning a zero for that argument. It then turns out that the argument parser was being passed an incorrect argument.

Once upon a time, I was stumped by a type error in this expression:

```
"#" ++ show lecNum ++ " " ++ [dayOfWeek] ++ " "
: classdays' (lecNum+1) (first+daysToNext) last pairs
```

I reduced it down to the following expression, which works:

```
"#" -- ++ show lecNum ++ " " ++ [dayOfWeek] ++ " "
: []
```

Note the use of that "`--`" (comment to end of line) just after "`#`", **temporarily hiding the rest of the line!**

I then tried advancing that "`--`" past the `show lecNum` call:

```
"#" ++ show lecNum -- ++ " " ++ [dayOfWeek] ++ " "
: []
```

It broke! I then tried a very simple equivalent expression:

```
"a" ++ "b" : []
```

It produced the original type error! I checked the precedence chart. (Slide 84.) I also used `:info` to look at the ++ and "cons" operators:

```
> :info (++)
```

```
(++) :: [a] -> [a] -> [a]
infixr 5 ++

> :info (:)
data [] a = ... | a : [a]
infixr 5 :
```

Since both ++ and `:` are **right** associative operators (`infixR`) with equal precedence (5),

```
"a" ++ "b" : []
```

grouped as

```
"a" ++ ("b" : [])
```

and that was the divergence between expectation and reality! I expected it to group as

```
("a" ++ "b") : []
```

but the reality was the opposite.

If you're puzzled by a syntax or type error, make an effort to chop down the code some before you send it to us. If you get it down as far as I did with `"a" ++ "b" : []`, then you've got a GREAT question! And, something as simple as `"a" ++ "b" : []` can clearly be posted on Piazza for all to see, without any worry about giving away part of a solution (which would cause me to give you a lot of grief!)

When you want us to take a look at a problem, send us the whole file, not just an excerpt where you think the error lies. Our first step is to reproduce the problem that you're seeing. We often can't do that without having all of your code.

ASSIGNMENT-WIDE RESTRICTIONS

There are three assignment-wide restrictions:

1. The only module you may import is `Data.Char`. You can use Prelude functions as long as they are not higher-order functions (see third restriction).

2. You may not use list comprehensions. See slide 135 for an example of one. The general form of a list comprehension is `[ `*`expression | qualifier1, ..., qualifierN`*` ]`. That vertical bar after the initial expression is the most obvious characteristic of a list comprehension.

3. You may not use any *higher-order functions*, which are functions that take other functions as arguments. We'll start talking about them in the section "Higher-order functions", probably around slide 250.

   I'd say that higher-order functions are hard to use by accident but I've been surprised before, so let me say a little bit to help you recognize them. A common example of a higher-order function is `map`:

   ```
   > :t map
   map :: (a -> b) -> [a] -> [b]
   ```

   `map` takes two arguments, the first of which has type `(a -> b)` and that's a function type. It's saying that the first argument is a function that takes one argument. Here's a usage of `map`—see if you can understand what it's doing:

```
> map negate [1..5]
[-1,-2,-3,-4,-5]
```

To be clear, the purpose of this whole section on the third restriction is to help you stay away from higher-order functions on this assignment. We'll be learning them about soon, and you'll see that we can use them instead of writing recursive functions in many cases, but **for now I want you writing recursive functions—think of it as getting good at walking before learning how to run!**

Here is, I believe, a full list of all higher-order functions in the Prelude:

```
all any break concatMap curry dropWhile either filter flip
foldl foldl1 foldr foldr1 interact iterate map mapM mapM_
maybe scanl scanl1 scanr scanr1 span takeWhile uncurry
until zipWith zipWith3
```

Try `:t` on some of them and look for argument types with `(... -> ...)`, `(... -> ... ->)`, etc.

Whenever I put restrictions in place we're happy to take a look at your code before it's due to see if there are any violations. Just mail it to `372s16`.

Strong Recommendation: Specify types for functions, but be careful!

Slides 91-93 talk about specifying types for functions but I'm going to say a little more about it here.

The following function has an error.

```
f ((x,y,z):t) index len
    | (index `mod` length) == 0 = ""
```

Here's the type that Haskell infers for that erroneous function:

```
f :: Integral ([a] -> Int) =>
     [(t1, t2, t3)] -> ([a] -> Int) -> t -> [Char]
```

Whoa! Where'd that `([a] -> Int)` for the second argument, `index`, come from?

If you look close, you'll see that instead of using the parameter `len`, the guard inadvertently uses `length`, a Prelude function. Since the type of `mod` is `Integral a => a -> a -> a`, Haskell proceeds to infer that this function needs a second argument that's an `[a] -> Int` function which is a instance of the `Integral` type class. I can't think of any use for such a thing, but **Haskell proceeds with that inferred type and bases subsequent type inferences on the assumption that the inferred type of `f` is correct**. That can produce far-flung false positives for errors in other functions.

If I simply precede the clause for `f` with a specification for the type of `f` that I intend, I get a perfect error message:

```
% cat extype.hs
f::[(Int,Int,Char)] -> Int -> Int -> String  -- The intended type
f ((x,y,z):t) index len
    | (index `mod` length) == 0 = ""
```

```
% ghci extype.hs
...
extype.hs:3:20:
  Couldn't match expected type `Int' with actual type `[a0] -> Int'
  In the second argument of `mod', namely `length'
  In the first argument of `(==)', namely `(index `mod` length)'
  In the expression: (index `mod` length) == 0
```

The downside of specifying a type for a function is that we might inadvertently make a function's type needlessly specific, perhaps by using an `Int` or `Double` when an instance of the `Num` type class would be better.

My common practice is to see what type Haskell infers for a newly written function. If it looks reasonable, I then "set it in stone" by adding a specification for that type. If an apparent type problem arises later, I might try temporarily commenting that type specification to help get a handle on the problem.

Helper functions are OK, except in `warmup.hs` and `ftypes.hs`

In general, writing helper functions to break a computation into smaller, simpler pieces is a good practice. However, the functions in `warmup.hs`, the first problem, are simple enough that you shouldn't need a helper to write any of them. The last problem on the assignment, `ftypes.hs`, has a number of problem-specific restrictions, including no helper functions.

Aside from `warmup.hs` and `ftypes.hs`, it's fine to use helper functions.

Haskell version issues

As slide 35 mentions, there are some non-trivial version issues with Haskell. We've yet to see any significant incompatibilities between lectura's 7.4.1 and the version I've suggested for your laptops (7.8.3), but your code will be tested on lectura, so how it behaves on lectura is what matters.

Prelude documentation

Prelude documentation for version 7.4.1 is here:
https://hackage.haskell.org/package/base-4.5.0.0/docs/Prelude.html. On the far right side of each entry is a link to a file with the source code for the function, but you'll find that many functions are written using elements of Haskell we haven't seen. There's a link for the above on the Piazza resources page, too—search for "Prelude".

You can see a list of all Prelude functions by using `:browse Prelude` at the `gchi` prompt.

At long last I present...

The Problems of a3!

Problem 1. (7 points) warmup.hs

The purpose of this problem is to get you warmed up by writing your own version of several simple functions from the Prelude: `last`, `init`, `replicate`, `drop`, `take`, `elem` and `++`.

The code for these functions is easy to find on the web and in books—a couple are shown as examples of recursion in chapter 4 of LYAH. Whether or not you've seen the code I'd like you first to try to write them from scratch. If you have trouble, go ahead and look for the code. Study it but then put it away and try to write the function from scratch. Repeat as needed. Think of these like practicing scales on a musical instrument.

To avoid conflicts with the Prelude functions of the same name, use these names for your versions:

Your function	Prelude function
`lst`	`last`
`initial`	`init`
`repl`	`replicate`
`drp`	`drop`
`tk`	`take`
`has`	`elem`
`concat2`	`(++)`

You should be able to write these functions using only pattern matching, comparisons in guards, list literals, cons (:), subtraction, and recursive calls to the function itself. If you find yourself about to use `if-else`, think about using a guard instead.

`concat2` is a function that's used exactly like you'd use the ++ operator as a function:

```
> concat2 "abc" "xyz"
"abcxyz"

> (++) "abc" "xyz"    -- see slides 77-78
"abcxyz"
```

You may find that `concat2` is the most difficult of the bunch—it's simple but subtle.

Experiment with the Prelude functions to see how they work. Note that `replicate`, `drop`, and `take` use a numeric count. Be sure to see how the Prelude versions behave with zero and negative values for that count. For testing with negative counts, remember that unary negation typically needs to be enclosed in parentheses:

```
> take (-3) "testing"
""
```

```
> drop (-3) "testing"
"testing"
```

You'll find that `last` and `init` throw an exception if called with an empty list. You can handle that with a clause like this one for `lst`:

```
lst [] = error "emptyList"
```

As I hope you'd assume, you can't use the Prelude function that you are recreating! **Beware that when writing these reproductions it's easy to forget and use the Prelude function by mistake**, like this:

```
drp ... = ... drop ...
```

Here's an `egrep` command you can use to quickly check for accidental use of the Prelude functions in your solution: (`a3/check-warmup`)

```
egrep -w "last|init|replicate|drop|take|elem|\+\+" warmup.hs
```

Testing note: If you run the Tester with `a3/tester warmup` you'll see that it runs tests for all of the expected functions, producing a long stream of failures for any functions you haven't completed. You can test functions one at a time by using a `-t` option following `warmup`:

```
% a3/tester warmup -t lst
...

% a3/tester warmup -t drp
...
```

Problem 2. (2 points) `join.hs`

Write a function `join separator strings` of type `[Char] -> [[Char]] -> [Char]` that concatenates the strings in `strings` into a single string with `separator` between each string. Examples:

```
> join "." ["a","bc","def"]
"a.bc.def"

> join ", " ["a", "bc"]
"a, bc"

> join "" ["a","bc","def", "g", "h"]
"abcdefgh"

> join "..." ["test"]
"test"

> join "..." []
""

> join ".." ["","","x","",""]
"....x...."

> join "-" (words "just testing this")
"just-testing-this"
```

In Java you might use a counter of some sort to know when to insert the separators but that's not the right approach in Haskell.

Problem 3. (4 points) `rme.hs`

Write a function `rme n` of type `Integral a => a -> a` that, **using an arithmetic approach**, removes the even digits from `n`, which is assumed to be greater than zero.

Examples:

```
> rme 3478
37

> rme 100100010010
1111

> rme (17^19)
39735515137153
```

Important: Your solution must be based on arithmetic operations like `*`, `-`, and `div` rather than doing something like using `show` to turn `n` into a string, and then processing that string with list operations.

A obvious difficulty is posed by numbers consisting solely of even digits, like `2468`. For such numbers, `rme` produces zero:

```
> rme 2468
0
```

Problem 4. (4 points) `splits.hs`

Consider splitting a list into two non-empty lists and creating a 2-tuple from those lists. For example, the list `[1,2,3,4]` could be split after the first element to produce the tuple `([1],[2,3,4])`. In this problem you are to write a function `splits` of type `[a] -> [([a], [a])]` that produces a list of tuples representing all the possible splits of the given list.

Examples:

```
> :type splits
splits :: [a] -> [([a], [a])]

> splits [1..4]
[([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]

> splits "xyz"
[("x","yz"),("xy","z")]

> splits [True,False]
[([True],[False])]

> length (splits [1..50])
49
```

In order to be split, a list must contain at least two elements. If `splits` is called with a list that has fewer than two elements, raise the exception `shortList`. Example:

```
> splits [1]
*** Exception: shortList
```

In case you missed it, there's an example of raising an exception in problem 1, for `lst`.

Problem 5. (7 points) `cpfx.hs`

Write a function `cpfx`, of type `[[Char]] -> [Char]`, that produces the common prefix, if any, among a list of strings.

If there is no common prefix or the list is empty, return an empty string. If the list has only one string, then that string is the result.

Examples:

```
> cpfx ["abc",  "ab", "abcd"]
"ab"

> cpfx ["abc",  "abcef", "a123"]
"a"

> cpfx ["xabc",  "xabcef", "axbc"]
""

> cpfx ["obscure","obscures","obscured","obscuring"]
"obscur"

> cpfx ["xabc"]
"xabc"

> cpfx []
""
```

Problem 6. (8 points) `paired.hs`

Write a function `paired s` of type `[Char] -> Bool` that returns `True` iff (if and only if) the parentheses in the string `s` are properly paired.

Examples with properly paired parentheses:

```
> paired "()"
True

> paired "(a+b)*(c-d)"
True

> paired "(()()(()))"
True

> paired "((1)(2)((3)))"
True

> paired "((()(()()(((())))((()))))"
True

> paired ""
True
```

Examples with improper pairing:

```
> paired ")"
False

> paired "("
False

> paired "())"
False

> paired "(a+b)*((c-d)"
False

> paired ")("
False
```

Note that you need only pay attention to parentheses:

```
> paired "a+}(/.$#${)[[["
True

> paired"a+}(/.$#$([[)["
False
```

Problem 7. (25 points) **`street.hs`**

In this problem you are to write a function `street` that prints an ASCII representation of the buildings along a street, as described by a list of `(Int, Int, Char)` tuples, each of which represents a building. The elements of the tuple represent the width, height, and character used to create the building, respectively.

Consider this example:

```
> street [(3,2,'x'), (2,6,'y'), (5,4,'z')]

   yy
   yy
   yyzzzzz
   yyzzzzz
xxxyyzzzzz
xxxyyzzzzz
----------
```

The street has three buildings. As specified by the first tuple, the first building has a width of three, a height of two, and is composed of `"x"`s. The second tuple specifies that a width of two, a height of six, and that `"y"`s be used for the second building. The third building has a width of five, a height of four, and is made of `"z"`s. Note that a blank line appears above the buildings and a line of hyphens (minus signs, not underscores) provides a foundation for the buildings.

This function does something we've only touched on lightly in class: it produces output, which being a side-effect, is a big deal in Haskell. Here's the type of `street`:

```
street::[(Int,Int,Char)] -> IO ()
```

What `street` returns is an *IO action*, which when evaluated produces output as a side effect. `putStr` is a Prelude function of type `String -> IO ()` that outputs a string:

```
> :set +t -- just to show us "it" after putStr
> putStr "hello\nworld\n"
hello
world
it :: ()
```

To avoid tangling with the details of I/O in Haskell on this assignment, make your `street` function look like this:

```
street buildings = putStr result
    where
       ...some number of expressions and helper functions that
          build up result, a string...
```

The string `result` will need to have whatever characters, blanks, and newlines are required, and that's the challenge of this problem—figuring out how to build up that multiline string!

To help, and hopefully not confuse, here's a trivial version of `street` that's hardwired for two buildings, `[(2,1,'a'), (2,2, 'b')]`:

```
streetHW _ = putStr result
    where
        result = "\n  bb\naabb\n----\n"
```

Execution:

```
> streetHW "foo"

  bb
aabb
----
```

Like I said, I hope this `streetHW` example doesn't confuse! It's intended to show the connection between (1) binding `result` to a string that represents the buildings, (2) calling `putStr` with `result`, and (3) the output being produced.

Open spaces may be placed between buildings by using buildings of zero height:

```
> street [(3,0,'a'), (3,4,'b') ,(1,0,'c'), (5,7,'d'), (2,0,'e')]

       ddddd
       ddddd
       ddddd
   bbb ddddd
   bbb ddddd
   bbb ddddd
   bbb ddddd
```

```
--------------
```

Note that the foundation (the line of hyphens) extends to the left of the "b" building and to the right of the "d" building because of the zero-height "a" and "e" buildings.

You may assume that: (1) at least one building is specified (2) a building width is always greater than zero (3) a building height is always greater than or equal to zero.

Additional examples:

```
> street [(2,5,'x')]

xx
xx
xx
xx
xx
--
> street [(5,0,'x')]

-----
```

Problem 8. (25 points) editstr.hs

For this problem you are to write a function `editstr ops s` that applies a sequence of operations (`ops`) to a string `s` and returns the resulting string. Here is the type of `editstr`:

```
> :type editstr
editstr :: [([Char], [Char], [Char])] -> [Char] -> [Char]
```

Note that `ops` is a list of tuples. One of the available operations is replacement. Here's a tuple that specifies that every blank is to be replaced with an underscore:

```
("rep", " ", "_")
```

Another operation is translation, specified with `"xlt"`.

```
("xlt", "aeiou", "AEIOU")
```

The above tuple specifies that every occurrence of `"a"` should be translated to "A", every `"e"` to `"E"`, etc. A tuple such as `("xlt", "aeiouAEIOU", "**********")` specifies that all vowels should be translated to asterisks.

Here are two cases I won't test with `xlt`:

- A duplicated "from" character, as in `("xlt", "aa", "12")`
- "from" characters appearing in the "to" string, as in `("xlt", "tab", "bats")`

Here is an example of a call that specifies a sequence of two operations, first a replacement and then a translation:

```
> editstr [("rep", " ", "_"),
           ("xlt", "aeiou", "AEIOU")] "just a test"
"jUst_A_tEst"
```

Note that for formatting purposes the example above and some below are broken across lines.

For `"rep"` (replace), the second element of the tuple is assumed to be a one-character string. The third element, the replacement, is a string of any length. For example, we can remove `"o"`s and triple `"e"`s like this:

```
> editstr [("rep", "o", ""), ("rep", "e", "eee")] "toothsomeness"
"tthsmeeeneeeess"
```

Another example:

```
> editstr [("xlt", "123456789", "xxxxxxxxx"),
           ("rep", "x", "")] "5203-3100-1230"
"0-00-0"
```

There are three simpler operations, too: length (`len`), reverse (`rev`), and replication (`x`):

```
> editstr [("len", "", "")] "testing"
"7"

> editstr [("rev", "", "")] "testing"
"gnitset"

> editstr [("x", "3", "")] "xy"
"xyxyxy"

> editstr [("x", "0", "")] "the"
""
```

Implementation note: The replication operation (`"x"`) requires conversion of a string to an `Int`. That can be done with the `read` function. Here's an example:

```
> let stringToInt s = read s::Int
> stringToInt "327"
327
```

Note that `read` does not do input! What it is "reading" from is its string argument, like `Integer.parseInt()` in Java. Because `read` is overloaded and can return values of many different types we use `::Int` to specifically request an `Int`.

Because we're using three-tuples of strings, `len`, `rev`, and `repl` leave us with one or two unused elements in the tuples.

Let's define some tuple-creating functions and simple value bindings so that we can specify operations with much less punctuation noise. **Put the following lines in your `editstr.hs`:**

```
rep from to = ("rep", from, to)

xlt from to = ("xlt", from, to)

len = ("len", "", "")

rev = ("rev", "", "")
```

```
x n = ("x", show n, "")      -- Note: show converts a value to a string
```

Recall this example above:

```
editstr [("xlt", "123456789", "xxxxxxxxx"),
          ("rep", "x", "")] "5203-3100-1230"
```

Let's redo it using the `rep` and `xlt` bindings from above.

```
>editstr [xlt "123456789" "xxxxxxxxx", rep "x" ""] "5203-3100-1230"
"0-00-0"
```

Note that instead of specifying two literal tuples as operations, we're specifying two function calls that create tuples instead. Notice what `editstr`'s first argument, a list with two expressions, turns into a list with two operation tuples:

```
> [xlt "123456789" "xxxxxxxxx", rep "x" ""]
[("xlt","123456789","xxxxxxxxx"),("rep","x","")]
```

Here's a more complex sequence of operations:

```
> editstr [x 2, len, x 3, rev, xlt "1" "x"] "testing"
"4x4x4x"
```

Operations are done from left to right. The above specifies the following steps:

1. Replicate the string twice, producing `"testingtesting"`.
2. Get the length of the string, producing `"14"`.
3. Replicate the string three times, producing `"141414"`.
4. Reverse the string, producing `"414141"`.
5. Translate `"1"`s into `"x"`s, producing `"4x4x4x"`.

Any number of modifications can be specified.

Again, it case it helps you understand what the `x`, `len`, `rev`, etc. bindings are all about, let's see what that first argument to `editstr` turns into:

```
> [x 2, len, x 3, rev, xlt "1" "x"]
[("x","2",""),("len","",""),("x","3",""),("rev","",""),("xlt","1","
x")]
```

Incidentally, this is a simple example of an *internal DSL* (Domain Specific Language) in Haskell. An expression like `[x 2, len, x 3, rev, xlt "1" "x"]` is using the facilities of Haskell to specify computation in a new language that's specialized for string manipulation. This write-up is already long enough so I won't say anything about DSLs here but you can Google and learn! What we now call Domain Specific Languages were often called "little languages" years ago.

If the list of operations is empty, the original string is returned.

```
> editstr [] "test"
"test"
```

The exception `badSpec` is raised to indicate any of three error conditions:

- An operation is something other than `"rep"`, `"xlt"`, `"rev"`, `"len"`, or `"x"`.
- For `"rep"`, the length of the string being replaced is not one.
- For `"xlt"`, the two strings are not the same length.

Here are examples of each, in turn:

```
> editstr [("foo", "the", "bar")] "test"
"*** Exception: badSpec

> editstr [("rep", "xx", "yy")] "test"
"*** Exception: badSpec

> editstr [("xlt", "abc", "1")] "test"
"*** Exception: badSpec
```

Problem 9. (5 points) `ftypes.hs`

Slides 86-89 demonstrate that Haskell infers types based on how values are used. Your task in this problem is to create a sequence of operations on function arguments that cause each of five functions, `fa`, `fb`, `fc`, `fd` and `fe` to have a specific inferred type. The functions will not be run, only loaded, and need not perform any meaningful computation or even terminate.

Here are the types, shown via interaction with `ghci`:

```
% ghci ftypes.hs
...
[1 of 1] Compiling Main             ( ftypes.hs, interpreted )
Ok, modules loaded: Main.
> :browse
fa :: Int -> Bool -> Char -> (Char, Int, Bool)
fb :: [Bool] -> [Int] -> Char -> String
fc :: (Num t1, Num t) => [(t1, t)] -> (t, t1)
fd :: (Integer, [a]) -> Int -> Bool
fe :: [[Char]] -> [[a]]
```

To make this problem challenging we need to have some restrictions:

No apostrophes (`'`), double-quotes (`"`) or decimal digits may appear in `ftypes.hs`, not even in comments. (Yes, this makes character, string, and numeric literals off-limits!)

You may not use `True` or `False`.

You may not use the `::`*`type`* specification, introduced on slide 65.

You may not define any additional functions.

You may not use the `fst` or `snd` functions from the Prelude.

You may not use "as-patterns", the `where` clause, or `let`, `do`, or `case` expressions.

You may not use guards or `if-else`.

Violation of a restriction will result in a score of a zero for that function.

Depending on your code you might end up with a type that's equivalent to a desired type but that has type variables with names that differ from those shown above. For example, `fc` is shown above as this,

```
fc :: (Num t1, Num t) => [(t1, t)] -> (t, t1)
```

but the Tester will consider the following to be correct, too:

```
fc :: (Num a, Num b) => [(a, b)] -> (b, a)
```

If you look close, you'll see that the only difference is that the former uses the type variables `t1` and `t` instead of `a` and `b`, respectively.

Similarly, the following type would also be considered correct:

```
fc :: (Num t, Num a) => [(t, a)] -> (a, t)
```

Problem 10. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a3/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `a`*`N.YYYYMMDD.HHMMSS`*`.tz` You can run `a3/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `372s16` if you need to recover a file and aren't familiar with `tar`—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a3/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
% wc $(grep -v txt < a3/delivs)
   29  115  491 warmup.hs
    3   21   81 join.hs
    8   42  215 rme.hs
    6   34  202 splits.hs
    8   38  171 cpfx.hs
    8   51  278 paired.hs
   25  143  907 street.hs
   49  220 1210 editstr.hs
   12   60  278 ftypes.hs
  148  724 3833 total
```

My code has few comments.

Miscellaneous

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) Two minus signs (`--`) is comment to end of line; `{-` and `-}` are used to enclose block comments, like `/*` and `*/` in Java.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 10 to 15 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put ten hours into this assignment and don't seem to be close to completing it, it's probably time to touch base with us. Specifically mention that you've reached ten hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment 4
Due: Friday, February 26 at 23:59:59

ASSIGNMENT-WIDE RESTRICTIONS

There are three assignment-wide restrictions:

1. Minus some exceptions that are noted for `group.hs` and `avg.hs`, the only module you may import is `Data.Char`. The purpose of this restriction is so that students don't waste time scouring dozens of Haskell packages in search of something that might be useful. `Data.Char` and the `Prelude` have all you need!

2. List comprehensions may **not** be used. They are interesting and powerful but due to time constraints we don't cover them. I want your attention focused on the elements of Haskell that we have covered.

3. Recall the idea put forth on slide 268: To build your skills with higher-order functions I want you to solve most of these problems while pretending that you don't understand recursion! Specifically, **except for problem 1, `warmup.hs`, you may not WRITE any recursive code!** Instead, use higher-order functions like `map`, `filter`, and the various folds. Those functions, and other Prelude functions, might themselves be recursive but that's no problem—it's OK to use recursive functions but you're prohibited from writing any recursive functions.

Make an `a4` symlink

Just like you did for `a3`, make an `a4` symlink:

```
$ cd ~/372/a4
$ ln -s /cs/www/classes/cs372/spring16/a4
```

Use the tester!

Just like for assignment 3, there's a tester for this assignment. **Don't just "eyeball" your output—use the tester!** I won't have any sympathy for those who fail tests during grading simply because they didn't want to bother with the tester! We'll be happy to help you with using the tester and understanding its output.

The tester is in `a4/tester`. Run it with "`a4/tester` *`PROBLEM-NAME`*". To maybe save a little typing, `a4/t` is symlinked to `a4/tester`, so you can run the tester with just `a4/t`.

Problem 1. (7 points) `warmup.hs`

This problem is like `warmup.hs` on assignment 3—I'd like you to write your own version of some functions from the Prelude: `map`, `filter`, `foldl`, `foldr`, `any`, `all`, and `zipWith`.

The code for `map`, `filter`, and `foldl` is in the slides and the others are easy to find, but I'd like you to start with a blank piece of paper and try to write them from scratch. If you have trouble, go ahead and look for the code. Study it but then put it away and try to write the function from scratch. Repeat as needed.

To avoid conflicts with the Prelude functions of the same name, use these names for your versions:

Your function	Prelude function
`mp`	`map`
`filt`	`filter`
`fl`	`foldl`
`fr`	`foldr`
`myany`	`any`
`myall`	`all`
`zw`	`zipWith`

You should be able to write these functions using only pattern matching, comparisons in guards, list literals, cons (`:`), subtraction, and recursive calls to the function itself. Experiment with the Prelude functions to see how they work.

You might find `foldl` and `foldr` to be tough. Don't get stuck on them!

Just like for `a3`'s `warmup.hs`, you can use a `-t` option with the tester to name a specific function to test. Example: "`a4/tester warmup -t mp`". Note that `-t mp` follows `warmup`.

This problem, `warmup.hs`, is the only problem on this assignment in which you can write recursive functions.

Problem 2. (3 points) `dezip.hs`

Write a function `dezip list` that separates a list of 2-tuples into two lists. The first list holds the first element of each tuple. The second list holds the second element of each tuple. **Don't overlook the additional restrictions for this problem following the examples.**

```
> :t dezip
dezip :: [(a, b)] -> ([a], [b])

> dezip [(1,10),(2,20),(3,30)]
([1,2,3],[10,20,30])

> dezip [("just","testing")]
(["just"],["testing"])

> dezip []
([],[])
```

ADDITIONAL RESTRICTIONS for `dezip.hs`:

The only functions your solution may use are `map`, `fst` , `(.)` (i.e., composition), and this function:

```
swap (x,y) = (y,x)    -- include this line in your solution
```

Remember that writing recursive functions is prohibited.

This function is just like the Prelude function `unzip`, but with a hopefully different enough name that you don't test with `unzip` by mistake!

Problem 3. (2 points) `repl.hs`

Write a function `repl` that works just like `replicate` in the Prelude.

```
> :t repl
repl :: Int -> a -> [a]

> repl 3 7
[7,7,7]

> repl 5 'a'
"aaaaa"

> repl 2 it
["aaaaa","aaaaa"]
```

ADDITIONAL RESTRICTION for `repl.hs`: You may not use `replicate`.

Remember the assignment-wide prohibition on recursion.

This is an easy problem; there are several ways to solve it using various functions from the Prelude. Just for fun you might see how many distinct solutions you can come up with.

Problem 4. (2 points) `doubler.hs`

Create a function named `doubler` that duplicates each value in a list:

```
> :t doubler
doubler :: [a] -> [a]

> doubler [1..5]
[1,1,2,2,3,3,4,4,5,5]

> doubler "bet"
"bbeett"

> doubler [[]]
[[],[]]

> doubler []
[]
```

RESTRICTION: Your solution must look like this:

```
doubler = foldr ...
```

That is, you are to bind `doubler` to a partial application of `foldr`. Think about using an anonymous function.

Replace the `...` with code that makes it work. Thirty characters should be about enough but it can be as long as you need.

Problem 5. (2 points) `revwords.hs`

Using point-free style (slides 301-303), create a function `revwords` that reverses the "words" in a string. Assume the string contains only letters and spaces.

```
> revwords "Reverse the words in this sentence"
"esreveR eht sdrow ni siht ecnetnes"

> revwords "testing"
"gnitset"

> revwords ""
""
```

You'll want to use the `words` function on this one.

Problem 6. (4 points) `cpfx.hs`

Once the `a3` deadline has passed, `a4/whm_cpfx.hs` will have my solution for `cpfx` from assignment 3. (Remind me if I forget!) The `cpfx` function is recursive. Rewrite that function to be non-recursive but still make use of my `cpfx'` function, the code for which is to be a part of your solution. Yes, `cpfx'` is recursive. That's OK.

See the assignment 3 write-up for examples of using `cpfx`.

Problem 7. (7 points) `nnn.hs`

The behavior of the function you are to write for this problem is best shown with an example:

```
> nnn [3,1,5]
["3-3-3","1","5-5-5-5-5"]
```

The first element is a `3` and that causes a corresponding value in the result to be a string of three `3`s, separated by dashes. Then we have one `1`. Then five `5`s.

More examples:

```
> :t nnn
nnn :: [Int] -> [[Char]]

> nnn [1,3..10]
["1","3-3-3","5-5-5-5-5","7-7-7-7-7-7-7","9-9-9-9-9-9-9-9-9"]

> nnn [10,2,4]
["10-10-10-10-10-10-10-10-10-10","2-2","4-4-4-4"]

> length (head (nnn [100]))
399
```

Note the math for the last example: 100 copies of `"100"` and 99 copies of `"-"` to separate them amount to 399 characters.

Assume that the values are greater than zero.

Remember: You can't write any recursive code!

Problem 8. (9 points) `expand.hs`

Consider the following two entries that specify the spelling of a word and spelling of forms of the word with suffixes:

```
program,s,#ed,#ing,'s
code,s,d,@ing
```

If a suffix begins with a pound sign (#), it indicates that the last letter of the word should be doubled when adding the suffix. If a suffix begins with an at-sign (`@`), it indicates that the last letter of the word should be dropped when adding the suffix. In all other cases, including the possessive (`'s`), the suffix is simply added. Given those rules, the two entries above represent the following words:

```
program
programs
programmed
programming
program's

code
codes
coded
coding
```

For this problem you are to write a function `expand entry` that returns a list that begins with the word with no suffix and is followed by all the suffixed forms in turn.

```
> :t expand
expand :: [Char] -> [String]

> expand "code,s,d,@ing"
["code","codes","coded","coding"]

> expand "program,s,#ed,#ing,'s"
["program","programs","programmed","programming","program's"]

> expand "adrift"                (If no suffixes, produce just the word.)
["adrift"]

> expand "a,b,c,d,e,f"
["a","ab","ac","ad","ae","af"]

> expand "a,b,c,d,@x,@y,@z,#1,#2,#3"
["a","ab","ac","ad","x","y","z","aa1","aa2","aa3"]

> expand "ab,#c,d,@e,f,::x"
["ab","abbc","abd","ae","abf","ab::x"]
```

A word may have any number of suffixes with an arbitrary combination of types. Words and suffixes may be arbitrarily long. <u>You may assume that an entry never contains a blank</u>, like `"a b,c"`.

Note that the only characters with special meaning are comma, `#`, and `@`. Everything else is just text.

<u>Assume that the entry is well-formed.</u> You won't see things like a zero-length word or suffix. Here are some examples of entries that will <u>not</u> be tested: `","`, `"test,"`, `"test,s,#,@"`

<u>Remember: You can't write any recursive code!</u>

Problem 9. (17 points) `pancakes.hs`

In this problem you are to print a representation of a series of stacks of pancakes. Let's start with an example:

```
> :t pancakes
pancakes :: [[Int]] -> IO ()

> pancakes [[3,1],[3,1,5]]
     ***
***   *
 *  *****
>
```

The list specifies two stacks of pancakes: the first stack has two pancakes, of widths `3` and `1`, respectively. The second stack has three pancakes. Pancakes are always centered on their stack. A single space separates each stack. Pancakes are always represented with asterisks.
Here's another example:

```
> pancakes [[1,5],[1,1,1],[11,3,15],[3,3,3,3],[1]]
                        ***
      *   ***********   ***
  *   *       ***       ***
***** * *************** *** *
>
```

There are opportunities for creative cooking:

```
> pancakes [[7,1,1,1,1,1],[5,7,7,7,7,5],[7,5,3,1,1,1],
[5,7,7,7,7,5], [7,1,1,1,1,1],[1,3,3,5,5,7]]
*******  *****  *******  *****  *******    *
   *    *******  *****  *******    *      ***
   *    *******   ***   *******    *      ***
   *    *******    *    *******    *     *****
   *    *******    *    *******    *     *****
   *     *****     *     *****     *    *******
>
```

Assume that there is at least one stack. Assume all stacks have at least one pancake. Assume all widths are <u>odd numbers</u> greater than zero. The smallest "order" you'll ever see is this:

```
> pancakes [[1]]
*
>
```

Like `street` on assignment 3, `pancakes` produces output. Use this structure:

```
pancakes stacks = putStr result
    where
        ...
        result = ...
```

Remember: You can't write any recursive code!

Problem 10. (17 points) `group.hs`

For this problem you are to write a program that reads a text file and prints the file with a line of dashes inserted whenever the first character on a line differs from the first character on the previous line. Additionally, the lines from the input file are to be numbered.

```
$ cat a4/group.1
able
academia
algae
carton
fairway
hex
hockshop

$ runghc group.hs a4/group.1
1 able
2 academia
3 algae
------
4 carton
------
5 fairway
------
6 hex
7 hockshop
$
```

First, note that the command `runghc`, not `ghci`, is being used.

Note also that only the lines from the input file are numbered. The separators are NOT numbered.

Lines with a length of zero (i.e., `length line == 0`) are discarded as a first step.

Another example:

```
$ cat a4/group.2
elemPos' _ [] = -1
elemPos' x ((val,pos):vps)
    | x == val = pos
    | otherwise = elemPos' x vps


f x y z = (x == chr y) == z

add_c x y = x + y


add_t(x,y) = x + y


fromToman 'I' = 1
```

```
fromRoman 'V' = 5
fromRoman 'X' = 10

p 1 (x:xs) = 10
$ runghc group.hs a4/group.2
1 elemPos' _ [] = -1
2 elemPos' x ((val,pos):vps)
------
3     | x == val = pos
4     | otherwise = elemPos' x vps
------
5 f x y z = (x == chr y) == z
------
6 add_c x y = x + y
7 add_t(x,y) = x + y
------
8 fromToman 'I' = 1
9 fromRoman 'V' = 5
10 fromRoman 'X' = 10
------
11 p 1 (x:xs) = 10
$
```

Note that when the line numbers grow to two digits the line contents are shifted a column to the right. That's ok.

If all lines start with the same character, no separators are printed.

```
$ cat a4/group.3
test
tests
testing
$ runghc group.hs a4/group.3
1 test
2 tests
3 testing
$
```

One final example:

```
$ cat a4/group.4
a
b
a
b
a
a
b
a
a
a
b
$ runghc group.hs a4/group.4
1 a
------
2 b
------
```

```
3 a
------
4 b
------
5 a
6 a
------
7 b
------
8 a
9 a
10 a
------
11 b
$
```

The separator lines are six dashes (minus signs).

Assume that there is at least one line in the input file. (A one-line file will produce no separators, of course.)

Implementation notes for `group.hs`

Unlike everything you've previously done with Haskell, this is a whole program, not just a function run at the `ghci` prompt. Follow the example of `longest` on slide 292 and have a binding for `main` that has a `do` block that sequences getting the command line arguments with `getArgs`, reading the whole file with `readFile`, and then calling `putStr` with result of a function named `group`, which does all the computation.

Your `group.hs` should look like this:

```
import System.Environment (getArgs)

main =
    do
       args <- getArgs
       bytes <- readFile $ head args
       putStr $ group bytes

...your functions here...
```

Yes, there's an `import` for something other than `Data.Char`. In this case we're asking for the `getArgs` function from the `System.Environment` module. This exception is permitted.

Note that the `$` operator, from slide 324, is being used to avoid some parentheses.

Remember: You can't write any recursive code!

Problem 11. (17 points) `avg.hs`

For this problem you are to write a Haskell program that computes some simple statistics for the hours reported in `observations.txt` submissions.

Let's use a pipeline to get a few lines of data into the file `avg.1`.

```
   $ cat {...}/observations.txt | grep -i hours > a4/avg.1
```

Here's what we got:

```
   $ cat a4/avg.1
   Hours: 3-5
   Hours: 10
   I spent 8 hours on this.
   Hours: 4-12
```

It looks like maybe somebody didn't read the instructions and wrote "`I spent...`" We'll ignore lines that don't start with "`Hours:`", case-sensitive. That leaves three lines, two with ranges. There's merit in being able to reflect uncertainty by reporting a range but we can't do simple arithmetic on a range. Let's view a range as representing three values: a low, a midpoint, and a high. Let's also view a single value as a range with low, midpoint, and high values that are equal. That gives us this view of the data:

	Low	Midpoint	High
3-5	3	4	5
10	10	10	10
4-12	4	8	12

Let's run `avg.hs` and specify only an input file:

```
   $ runghc avg.hs a4/avg.1
   n = 3
   mean = 7.333
   median = 8.000

   Ignored:
   Line 3: I spent 8 hours on this.
```

We see that:

- Three valid data points were found.
- The mean is `7.333`, which is `(4+10+8)/3` reported to three decimal places.
- The median is the middle data point in a sequence of values and in this case is `8`. As a simplification we show the median with three decimal places. (If there are an even number of values, and thus no middle value, the median is the mean of the two center-most values. For example the median of the four values `1`, `3`, `7`, `15` is `5` (`(3+7)/2`).)
- Line 3, whose contents are shown, was ignored.

If `avg.hs` is run with a `-l` (L) option, which must precede the file name, the low values (see the table) are used instead. In order, those values are `3`, `4`, `10`. We see this output:

```
   $ runghc avg.hs -l avg.1
   n = 3
   mean = 5.667
   median = 4.000
```

```
Ignored:
Line 3: I spent 8 hours on this.
```

Similarly, there's a `-h` option to compute the statistics using the high values (`5`, `10`, `12`):

```
$ runghc avg.hs -h avg.1
n = 3
mean = 9.000
median = 10.000

Ignored:
Line 3: I spent 8 hours on this.
```

Here are some points to keep in mind:

- Lines that don't start with "`Hours:`" are not included in the calculation but are reported under "`Ignored:`".

- Following "`Hours:`", discard all characters other than decimal digits, period (`.`), and dash (`-`).

  Then, ASSUME that what's left will be either a number, like `10`, or `7.5`, or a range, like `5.5-15`. (The behavior of `avg.hs` is undefined for other cases, which in practical terms means that I won't test with any such cases.) For ranges, the first value will always be less than the second.

- ASSUME that the command line arguments, which follow `runghc avg.hs`, are an optional `-l` or `-h`, followed by a file name. That amounts to three potential cases:

  ```
  runghc avg.hs FILENAME
  runghc avg.hs -l FILENAME
  runghc avg.hs -h FILENAME
  ```

  Behavior is undefined in all other cases. (Again, that means I won't test with any other cases.)

- ASSUME there will always be at least one valid data point in the input file.

- The tester's student set of tests for this problem will be the grading set of tests, too. In other words, **if the tester shows your `avg` passing all tests and you don't violate any restrictions, you are guaranteed full credit on this problem.** The final set of tests will be in place by noon on Saturday, Feburary 20.

Implementation notes for `avg.hs`

Here's a collection of implementation notes for `avg.hs`.

Some imports

In addition to the functions in the Prelude and `Data.Char`, you are permitted to use the following functions on this problem, `avg.hs`:

```
System.Environment.getArgs
Text.Printf.printf
```

and all functions in the `Data.List` module.

My solution starts by importing `getArgs` from `System.Environment`, `printf` from `Text.Printf`, and then all functions in `Data.List` and `Data.Char`:

```
import System.Environment (getArgs)
import Text.Printf (printf)
import Data.List
import Data.Char
```

See https://hackage.haskell.org/package/base-4.5.0.0/docs/Data-List.html for documentation on the functions in the `Data.List` module. Note: My solution uses only two functions from `Data.List`: `sort` and `partition`.

`main` for `avg.hs`

Just like `group.hs`, `avg.hs` does I/O. Here's the binding for `main` that I recommend you use:

```
main = do
    args <- getArgs
    bytes <- readFile $ last args
    putStr $ averages bytes $ init args
```

Like shown on slides 291-292, the `averages` function computes a string with newlines that `main` outputs with `putStr`. **Note that the $ operator, from slide 324, is being used to avoid some parentheses.**

`Double` to fixed point conversion

Use the following function to convert `Doubles` to `Strings` with three places of precision, for mean and median.

```
fmtDouble::Double -> String
fmtDouble x = printf "%.3f" x
```

Dividing a `Double` by an `Int` (or `Integer`)

You'll find that dividing a `Double` by an `Int` or an `Integer` produces a type error. A conversion can be done with the `fromIntegral` function:

```
> let sum = read "10.4"::Double
> sum / (fromIntegral $ length [1,2])
5.2
```

The type of `fromIntegral` is interesting:

```
> :t fromIntegral
fromIntegral :: (Num b, Integral a) => a -> b
```

Rather than converting an `Integral` type to a specific type, like `Double`, it's treated as a more general thing, a type that's an instance of `Num`. Then in turn, that type can be converted to a `Double`.

A starter file

As a convenience, `a4/avg_starter.hs` has the above `imports` and `main`, a stub for `averages`, and `fmtDouble` from above.

splitHours

Also in in a4/avg_starter.hs is splitHours, a function to split up specifications of hours:

```
> splitHours "10"
["10"]

> splitHours "3.4-10.3"
["3.4","10.3"]
```

Another development/debugging technique

One way to get a look at values bound in a where clause for a function is to temporarily have the function return a tuple that comprises values of interest. Look at this **mid-development snapshot** of averages:

```
averages bytes args = (validEntries, "values:", values,
            "selected:", selected, "errors:", errs, stats, errors)
    where ...
           validEntries = ...
           values = ...
           selected = ...
           errs = ...
           stats = ...
           errors = ...
           ...and more...
```

Instead of returning a fully-formatted final result, it just creates a tuple with the various intermediate bindings like validEntries, values, selected, etc.

Let's try a call to it, passing in a string with embedded newlines, which might come from a two-line file, and the list ["-h"], simulating a -h command-line argument:

```
> averages "Hours: 10\nI spent 2 hours\n" ["-h"]
([(1,True,"10")],"values:",[(10.0,10.0,10.0)],"selected:",[10.0],"er
rors:",[(2,False,"I spent 2 hours")],"n = 1\nmean = 10.000\nmedian =
10.000\n","\nIgnored:\nLine 2: I spent 2 hours\n")
```

Note that the literal strings like "values:" just serve as labels, to help us see what's what. Note also that the "stats:" and "errors:" strings are the final output, in two pieces. You can learn a few things about how I approached the problem by looking closely at that output.

Below is a main that works with the mid-development snapshot of averages above. Because the version of averages above returns a tuple and putStr wants a string, I use show to turn that tuple into a string:

```
main = do
    args <- getArgs
    bytes <- readFile $ last args
    putStr $ (show $ averages bytes (init args)) ++ "\n"
```

With the above averages and main, here's what I see with my version:

```
$ runghc avg.hs -h avg.1
([(1,True,"3-5"),(2,True,"10"),(4,True,"4-12")],"values:",
[(3.0,4.0, 5.0), (10.0,10.0,10.0), (4.0,8.0,12.0)], "selected:",
[5.0,10.0,12.0], "errors:",[(3,False,"I spent 8 hours on this.")],"n
= 3\nmean = 9.000\nmedian = 10.000\n","\nIgnored:\nLine 3: I spent 8
hours on this.\n")
```

`a4/tryall` script

`a4/tryall` is a bash script that runs `avg.hs` on a given data file using each of the low, midpoint, and high modes in turn. Do `cat a4/tryall` to get a look at it and then do this:

```
$ a4/tryall a4/avg.1
...output for each of the three modes in turn...
```

Finally, one last reminder

Remember: You can't write any recursive code!

Problem 12. (ZERO points) `rmRanges.hs`

This problem is worth no points. Try it if you wish.

Write a function `rmRanges` that accepts a list of ranges represented as 2-tuples and produces a function that when applied to a list of values produces the values that do not fall into any of the ranges. Ranges are inclusive, as the examples below demonstrate.

Note that `rmRanges` is typed in terms of the `Ord` type class, so `rmRanges` works with many different types of values.

```
> :type rmRanges
rmRanges :: Ord a => [(a, a)] -> [a] -> [a]

> rmRanges [(3,7)] [1..10]
[1,2,8,9,10]

> rmRanges [(10,18), (2,5), (20,20)] [1..25]
[1,6,7,8,9,19,21,22,23,24,25]

> rmRanges [] [1..3]
[1,2,3]

> rmRanges [('0','9')] "Sat Feb  8 20:34:50 2014"
"Sat Feb   :: "

> rmRanges [('A','Z'), (' ', ' ')] it
"ateb::"

> let f = rmRanges [(5,20),(-100,0)]

> f [1..30]
[1,2,3,4,21,22,23,24,25,26,27,28,29,30]

> f [-10,-9..21]
[1,2,3,4,21]
```

Assume for a range `(x, y)` that `x <= y`, i.e, you won't see a range like `(10,1)` or `('z', 'a')`.

As you can see above, ranges are inclusive. The range (1,3) removes 1, 2, and 3.

Just for fun...Here's an instance declaration from the Prelude:

```
instance (Ord a, Ord b) => Ord (a, b)
```

It says that if the values in a 2-tuple are orderable, then the 2-tuple is orderable. With that in mind, consider this rmRanges example:

```
> rmRanges [((3,'z'),(25,'a'))] (zip [1..] ['a'..'z'])
[(1,'a'),(2,'b'),(3,'c'),(25,'y'),(26,'z')]
```

Remember: You can't write any recursive code!

Problem 13. Extra Credit observations.txt

Submit a plain text file named observations.txt with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with "Hours:". There must be only one "Hours:" line in observations.txt. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use a4/turnin to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like a*N.YYYYMMDD.HHMMSS*.tz You can run a4/turnin as often as you want. We'll grade your final submission.

Note that each of the aN.*.tz files is essentially a backup, too, but perhaps mail to 372s16 if you need to recover a file and aren't familiar with tar—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

a4/turnin -l shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
$ wc $(grep -v txt a4/delivs)
   26  129  472 warmup.hs
    4   22  103 dezip.hs
    2   13   54 repl.hs
    2   12   58 doubler.hs
    2    9   68 revwords.hs
    7   30  131 cpfx.hs
    5   36  212 nnn.hs
   19   84  536 expand.hs
   16   81  551 pancakes.hs
   19   84  554 group.hs
   58  317 2022 avg.hs
    9   54  306 rmRange.hs
  169  871 5067 total
```

My code has few comments.

Miscellaneous

REMEMBER: Except for `warmup.hs` you are not permitted to write any recursive functions on this assignment!

This assignment is based on the material on Haskell slides 1-354.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 10 to 15 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put ten hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached ten hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment 5
Due: **Tuesday**, March 8 at 23:59:59

Game plan for the Ruby assignments and mid-term exam

Our mid-term exam will be on Thursday, March 10. It will be in our regular classroom, BIOW 208, at our regular class time, 2:00pm. It will cover the Haskell material through slide 354, and the Ruby material through what's needed for this assignment (Ruby slide 137). I'll be posting some details about the exam on Piazza in the coming week.

Due to that mid-term exam on March 10 and Spring Break the following week, I'm spreading our Ruby work across three assignments, with the following due dates:

Assignment 5	Tuesday, March 8	(date is definite)
Assignment 6	Friday, March 25	(date is tentative)
Assignment 7	Friday, April 8	(date is tentative)

I think of this assignment and the next assignment, Assignment 6, as being about a week of work each.

Restrictions on `longest.rb` and `seqwords.rb`

Problems 1 and 2 have some restrictions that will hopefully lead to some creative thinking about string-based computations. I intend them to be a challenge and a learning experience, not a frustration. I suggest that you start thinking about them as soon as possible and see what your subconscious comes up with. I believe everyone can solve these problems on their own but if you start to get frustrated, or the time you've budgeted starts to get short, ask us for some hints!

Problems 3 and 4, `minmax` and `xfield`, have no restrictions whatsoever.

Don't fall into the trap of thinking you must do these problems in sequence!

Use Ruby 2.2.4!

Ruby 2.2.4 is to be used for all Ruby assignments. Use `rvm`, as shown on Ruby slides 13-14, to select version 2.2.4 on lectura.

The Usual Stuff

Make an `a5` symlink that references `/cs/www/classes/cs372/spring16/a5`. Test using `a5/tester` (or `a5/t`).

Output from command-line examples is followed by a blank line

Most of the programming problems on the Haskell assignments were functions that you tested inside `ghci`. All of the Ruby problems are this assignment require you to create programs that can be run from the command line, like `group.hs` and `avg.hs` on assignment 4.

A strong convention in the UNIX world is that programs do not output a trailing blank line. Observe these interactions, and the absence of blank lines:

```
$ date
```

```
Wed Feb 24 09:37:27 MST 2016
$ date | wc
       1       6      29
$ ls -ld .
drwxr-xr-x 266 whm staff 9044 Feb 24 09:19 .
$ (my prompt—bash is waiting for the next command)
```

However, for ease of reading in this write-up, output from command-line examples is followed by a blank line. Instead of the above, you'll see this:

```
$ date
Wed Feb 24 09:37:27 MST 2016

$ date | wc
       1       6      29

$ ls -ld .
drwxr-xr-x 266 whm staff 9044 Feb 24 09:19 .
```

Problem 1. (6 points) `longest.rb`

Write a Ruby program that reads lines from standard input and upon end of file writes the longest line to standard output. If there are ties for the longest line, `longest` writes out all the lines that tie. If there is no input, `longest` produces no output.

Don't overlook the restrictions mentioned below.

Here are some examples.

```
$ cat a5/lg.1
a test
for
the program
here

$ ruby longest.rb < a5/lg.1
the program

$ cat a5/lg.2
xx
a
yy
b
zz

$ ruby longest.rb < a5/lg.2
xx
yy
zz
```

Let's use the null device and `wc -c` to demonstrate that if `longest` has no input, there's no output:

```
$ ruby longest.rb < /dev/null | wc -c
0
```

Let's use `longest` and some `grep`s to explore a list of words:

```
$ ruby longest.rb < /usr/share/dict/words
electroencephalograph's

$ grep ^q /usr/share/dict/words | ruby longest.rb
quadrilateral's
quadruplicate's
quadruplicating
qualification's
quartermaster's
questionnaire's

$ grep ^w /usr/share/dict/words | ruby longest.rb
whatchamacallit's
wrongheadedness's

$ grep ^wo /usr/share/dict/words | ruby longest.rb
woolgathering's
worthlessness's
```

Restrictions for `longest.rb`:

- **NO COMPARISONS, such as <, ==, `!=`, <=>, `between?`, `eql?`, `empty?`, and `String#casecmp` may be used. (As a rule, any method that ends with `?` should be viewed with suspicion.)**

- **The `case` statement may not be used.**

- **No arithmetic operations, such as addition, subtraction, negation (including negative literals such as -1), or the `pred` and `succ` methods may be used. Imagine that arithmetic was just never invented!**

- **The only types you may use are `Fixnum`, `Bignum`, `String`, along with the values `nil`, `true`, and `false` (i.e., the values of the single-value classes `NilClass`, `TrueClass`, and `FalseClass`). In particular, you may not use arrays.**

Regarding that last restriction, about types, **try to imagine that Ruby has only those six types**, so you've got to devise a solution using only values that are instances of those six types. As an example, consider this expression, where `s` is a `String`;

```
x = s.split[0]
```

Splitting a `String` produces an `Array`, which we're imagining that Ruby doesn't have, so that's a violation of the restrictions.

How about `x = "abc"; y = x[1]`? With Java in mind, you might think of `x[1]` as array indexing but x is a `String`, so `x[1]` is an operation on a `String` that produces a `String`, and that's fine.

Don't worry about what types might be used inside Ruby library method calls. For example, a poor implementation of `String`'s `reverse` method might use an array. That is of no concern!

Important: Regarding comparisons, you are permitted to employ the comparison that's implicit in control structures. For example, statements like

```
while x do ...          # OK
if f(x) then ...        # OK
```

are permitted.

However, a statement such as

```
while x > 1 do          # NOT PERMITTED!
```

is not permitted—it contains a comparison (`x > 1`).

Implementation note

The examples above show various combinations of redirection and piping with bash to supply `longest.rb` with data on "standard input" but all you need to do in `longest.rb` is read lines with `line = gets` (or whatever variable you want to use.)

Testing note

The testing machinery takes some steps to help you honor the restrictions but the approach is not foolproof. It catches a number of violations but it doesn't catch everything.

Here's an example of what the tester does with `longest.rb`:

```
(echo "load 'a5/check-longest.rb'"; cat longest.rb) >.longest-chk.rb
ruby .longest-chk.rb < a5/lg.1
```

The first line uses a *subshell* to create a new file, `.longest-chk.rb`, that's a copy of your `longest.rb` with an additional line, `"load 'a5/check-longest.rb'"`, added at the beginning. The Ruby code loaded by that additional line disables a number of disallowed operations like the < and > comparison operators on `Fixnums` and `Strings`.

The name of the new file, `.longest-chk.rb` starts with a dot, making it a hidden file, so your directory isn't cluttered with it. `ls` won't show it but `ls -a` will.

Let's imagine we've got a version of `longest.rb` that uses the < operator, which is not permitted. Here's what the failure will look like:

```
$ a5/tester longest
...
Test: 'ruby .longest-chk.rb < a5/lg.1': FAILED
Differences (expected/actual):
*** a5/master/tester.out/longest.out.01 2016-02-22
--- tester.out/longest.out.01   2016-02-22
***************
*** 1 ****
! the program
--- 1,2 ----
! (eval):1:in `<': oops -- can't use '<' for that type! (RuntimeError)
!       from .longest-chk.rb:5:in `<main>'
```

Note that `"...oops -- can't use '<' for that type!"`. The traceback shows the error occurred at line 5 in `.longest-chk.rb`, so that corresponds to line 4 in `longest.rb`.

Important: The code in `a5/check-longest.rb` will catch common violations of the restrictions but it's not foolproof. The "safe harbor" is to mail your code to `372s16` and ask for a manual inspection.

Problem 2. (7 points) **`seqwords.rb`**

For this problem you are to write a Ruby program that reads a series of words from standard input, one per line, and then prints lines with the words sequenced according to a series of specifications, also one per line and read from standard input.

Don't overlook the restriction mentioned below.

Here is an example with four words and three sequencing specifications, which are simply integers, one per line.

```
$ cat a5/sw.1
one
two
three
four
.
1
2
3
.
3
2
1
1
2
3
.
4
1

$ ruby seqwords.rb < a5/sw.1
one two three
three two one one two three
four one
```

Note that lines containing only a period (.) end the word list and also separate specifications. For output, words are separated with a single blank. Here's another example:

```
$ cat a5/sw.2
tick
.
1
.
1
1

$ ruby seqwords.rb < a5/sw.2
tick
tick tick
```

Assume that there is at least one word and at least one sequencing specification. Assume that each sequencing specification has at least one number. Assume that all entries in the sequencing specifications are integers and in range for the list of words. Because periods separate specifications, the input will never end with a period.

Here is a key simplification: Assume that words are between 1 and 100,000 characters in length, inclusive. (Note that `100_000` is a valid `Fixnum` literal in Ruby.)

Restriction: The only types you may use are `Fixnum`, `Bignum`, and `String`, along with the values `nil`, `true`, and `false` (i.e., the values of the single-value classes `NilClass`, `TrueClass`, and `FalseClass`). In particular, you may not use arrays.

The tester uses an approach like that used for `longest.rb` to look for violations of the restrictions, combining `a5/check-seqwords.rb` with your `seqwords.rb` to create `.seqwords-chk.rb`, which is then run. As with `longest.rb`, the process is not foolproof; don't hesitate to ask for a manual inspection of your `seqwords.rb`.

Problem 3. (8 points) `minmax.rb`

Write a Ruby program that reads lines from standard input and determines which line(s) are the longest and shortest lines in the file. The minimum and maximum lengths are output along with the line numbers of the line(s) having that length.

```
$ cat a5/mm.1
just a
test
right here
x

$ ruby minmax.rb < a5/mm.1
Min length: 1 (4)
Max length: 10 (3)
```

The output indicates that the shortest line is line 4; it is one character in length. The longest line is line 3; it is ten characters in length.

Another example:

```
$ cat a5/mm.2
xxx
xx
yyy
yy
zzz
qqq

$ ruby minmax.rb < a5/mm.2
Min length: 2 (2, 4)
Max length: 3 (1, 3, 5, 6)
```

In this case, lines 2 and 4 are tied for being the shortest line. Four lines are tied for maximum length.

If the input file is empty, the program should output a single line that states "Empty file":

```
$ ruby minmax.rb < /dev/null
Empty file
```

Here are some examples with `/usr/share/dict/words`:

```
$ ruby minmax.rb < /usr/share/dict/words
Min length: 1 (1, 1229, 2448, 3799, 4500, 5076, 5514, 6213, 6951,
7266, 7757, 8316, 9129, 10654, 11149, 11482, 12369, 12426, 13108,
14502, 15288, 15415, 15729, 16171, 16207, 16346, 16484, 21151,
26000, 34170, 39292, 42567, 46256, 49013, 52079, 55431, 56202,
56806, 59393, 63790, 65313, 67250, 74022, 74432, 79106, 89039,
93338, 95154, 96416, 98713, 98730, 99012)
Max length: 23 (39886)

$ ruby minmax.rb < /usr/share/dict/words | wc -l
2

$ grep ^q /usr/share/dict/words | ruby minmax.rb
Min length: 1 (1)
Max length: 15 (27, 49, 52, 86, 162, 251)
```

Although output is shown wrapped across several lines the first invocation above produces only two lines of output, demonstrated by piping that same output into `wc -l`.

IMPORTANT: DO NOT assume any maximum length for input lines.

You might be inclined to have some repetitious code in your `minmax.rb`, such as an if-then-else that handles minimum lengths and a nearly identical if-then-else that handles maximum lengths. There are no extra points for it for but I challenge you to produce a solution in which there is no repetitious code.

I consider even something like the following, albeit short, to be repetitious:

```
mins = [1]
maxs = [1]
```

If you think your solution has no repetition, include the following comment and I'll see if I agree.

```
# Look! No repetition!
```

Problem 4. (16 points) `xfield.rb`

For this problem you are going to create your own version of a Ruby tool that I use every day: `xfield`. `xfield` was inspired by the UNIX `cut` command but its behavior differs in various ways.

Here's a `man`-style description of `xfield`. Detailed examples follow.

SYNOPSIS:

```
xfield [-dC] [-sSEPARATOR] [FIELDNUM|TEXT]...
```

`xfield` extracts fields of data from standard input. Field numbers are one-based and may be negative to specify counting from the right. If a field number is out of bounds, "`<NONE>`" is output in place of actual data. Unlike `cut(1)`, `xfield` allows fields to be specified in any order and appear more than once.

Fields are delimited by one or more spaces by default but an alternate character to delimit fields can be specified with `-dC`. Tabs separate output fields by default but `-s`*`SEPARATOR`* can be used to specify an alternate separator, which must be at least one character in length. There may be multiple `-d` and `-s` specifications, in any order, but they must appear before any field number or text specifications. If there are multiple specifications for either `-d` or `-s`, the last one of each "wins".

If a textual argument (not a number) falls between two field specifications (two numbers), that text is used instead of the separator.

Input lines are assumed to end with a newline. If there are no input lines, `xfield` produces no output.

If no fields are specified, the message "`xfield: no fields specified`" is printed, and `xfield` calls `exit 1` to terminate execution. (`exit` is a `Kernel` method.)

`xfield` is able to handle any number of input lines. (Hint: Don't do something like read all the input lines into memory and then process them—they might not fit! Just process lines one at a time.)

The behavior of `xfield` is undefined in cases that are not specifically addressed by this write-up or exercised with the tester. That means that any non-malicious behavior is ok—run-time errors, curious results, etc., are not a surprise if the user misuses `xfield`.

Some detailed examples of `xfield` in operation follow. Here is an input file:

```
$ cat a5/xf.1
one     1   1.0
two     2   2.0
three   3   3.0
four    4   4.0
twenty  20  20.0
```

The English text and the real numbers can be extracted like this:

```
$ ruby xfield.rb 1 3 < a5/xf.1
one     1.0
two     2.0
three   3.0
four    4.0
twenty  20.0
```

`xfield` can be used to reorder fields:

```
$ ruby xfield.rb 3 2 1 < xf.1
1.0     1       one
2.0     2       two
3.0     3       three
4.0     4       four
20.0    20      twenty
```

`xfield` supports negative indexing, just like Ruby arrays:

```
$ ruby xfield.rb -1 1 < a5/xf.1
1.0     one
2.0     two
3.0     three
4.0     four
20.0    twenty
```

```
$ ruby xfield.rb -1 1 2 -2 < a5/xf.1
1.0     one     1       1
2.0     two     2       2
3.0     three   3       3
4.0     four    4       4
20.0    twenty  20      20
```

If a field reference is out of bounds, the string `"<NONE>"` is output:

```
$ ruby xfield.rb 1 10 2 < a5/xf.1
one     <NONE>  1
two     <NONE>  2
three   <NONE>  3
four    <NONE>  4
twenty  <NONE>  20
```

The `-s` flag specifies an output separator to use instead of tab.

```
$ ruby xfield.rb -s... 1 3 1 < a5/xf.1
one...1.0...one
two...2.0...two
three...3.0...three
four...4.0...four
twenty...20.0...twenty
```

To extract login ids and real names (and room/phone) from `a5/oldpasswd`, an excerpt from an ancient `/etc/passwd`, one might use `-d:` to specify that a colon is the delimiter:

```
$ ruby xfield.rb -d: 1 5 < a5/oldpasswd
wnj     Bill Joy,457E,7780
dmr     Dennis Ritchie
ken     Ken Thompson
mike    Mike Karels
carl    Carl Smith,508-21E,6258
joshua  Josh Goldstein
```

Note that the `-s` and `-d` options are single arguments—there's no space between `-s` or `-d` and the following string.

Non-numeric arguments other than the `-s` and `-d` flags are considered to be text to be included in each output line. If a textual argument (not a number) falls between two field specifications (two numbers), that text is used instead of the separator:

```
$ ruby xfield.rb int= 2 ", real=" 3 ", english=" 1 < a5/xf.1
int=1, real=1.0, english=one
int=2, real=2.0, english=two
int=3, real=3.0, english=three
int=4, real=4.0, english=four
int=20, real=20.0, english=twenty
```

Note the use of quotation marks to form an argument that contains blanks. The shell strips off the quotation marks so that the resulting arguments passed to the program do not have quotes. See the *Implementation notes for `xfield`* section below for more on this.

Here's that text-argument-overrides-separator rule again:

If a textual argument (not a number) falls between two field specifications (two numbers), that text is used instead of the separator:

Here are some more examples showing that rule in action:

```
$ cat a5/xf.2
one two three four

$ ruby xfield.rb -s. 1 2 3 < a5/xf.2
one.two.three

$ ruby xfield.rb -s. A 1 2 3 B C < a5/xf.2
Aone.two.threeBC

$ ruby xfield.rb -s. A 1 B C 2 3 D  < a5/xf.2
AoneBCtwo.threeD

$ ruby xfield.rb -s. A 1 B C 2 D 3 E  < a5/xf.2a5
AoneBCtwoDthreeE
```

Below are some cases that bring all the elements into play.

```
$ cat a5/xf.3
xxxxxxxAxxxxxxxxxxBxC
DxExF
xG1xG2
xxxxHIxxxJKxxxLMNxxxOPQRSx

$ ruby xfield.rb -dx -s- 1 2 3 < a5/xf.3
A-B-C
D-E-F
G1-G2-<NONE>
HI-JK-LMN

$ ruby xfield.rb -dx -s- -1 ... -2 -3 @ < a5/xf.3
C...B-A@
F...E-D@
G2...G1-<NONE>@
OPQRS...LMN-JK@

$ ruby xfield.rb -s/ -de 1 2 < a5/xf.1
on/     1   1.0
two     2   2.0/<NONE>
thr/   3   3.0
four    4   4.0/<NONE>
tw/nty  20  20.0
```

If there are no input lines, `xfield` produces no output:

```
$ ruby xfield.rb -s/ -d: 1 x 2 3 < /dev/null
```

Implementation notes for `xfield`

`gets` vs. `STDIN.gets`

The `gets` method does a little more than simply reading lines from standard input: If command line arguments are specified, `gets` will consider those arguments to be file names. It will then try to open those files and produce the lines from each in turn. That's really handy in some cases but it gets in the way for `xfield`. To avoid this behavior, **don't use "`line = gets`" to read lines**. Instead, do this:

```
while line = STDIN.gets do
```

That limits `gets` to the contents of standard input.

Delimiter-specific behavior in `String#split`

I'm astounded by it but the fact is that `split` behaves differently when the delimiter is a space:

```
>> " a  b  c ".split(" ")
=> ["a", "b", "c"]

>> ".a..b..c.".split(".")
=> ["", "a", "", "b", "", "c"]
```

Command line argument handling

The command line arguments specified when a Ruby program is run are made available as strings in `ARGV`, an array. Here is `echo.rb`, a Ruby program that prints the command line arguments:

```
printf("%d arguments:\n", ARGV.length)
for i in 0...ARGV.length do      # 0...N is 0..N-1
    printf("argument %d is '%s'\n", i, ARGV[i])
end
```

Execution:

```
$ ruby echo.rb -s -s2 -abc x y
5 arguments:
argument 0 is '-s'
argument 1 is '-s2'
argument 2 is '-abc'
argument 3 is 'x'
argument 4 is 'y'
```

Unescaped quotes and backslashes specified on the command line are processed and fully consumed by the shell; the program doesn't "see" them. Example:

```
$ ruby echo.rb int= 2 ", real=" 3 ", english="
5 arguments:
argument 0 is 'int='
argument 1 is '2'
argument 2 is ', real='
argument 3 is '3'
argument 4 is ', english='

$ ruby echo.rb "     "        '  x '   \ \y\  ""
4 arguments:
argument 0 is '     '
argument 1 is '  x '
argument 2 is ' y '
argument 3 is ''
```

The shell does provide some mechanisms to allow quotes and backslashes to be transmitted in arguments:

```
$ ruby echo.rb '"'  \\x\\
2 arguments:
argument 0 is '"'
argument 1 is '\x\'
```

Additionally, the consumes I/O redirections with < and >—the program never sees those operators or their accompanying filename argument. Here's some evidence of that:

```
$ ruby echo.rb 1 2 3 < lg.1
3 arguments:
argument 0 is '1'
argument 1 is '2'
argument 2 is '3'

$ ruby echo.rb 1 2 3 < lg.1 > out

$ cat out
3 arguments:
argument 0 is '1'
argument 1 is '2'
argument 2 is '3'
```

The above examples were produced with a UNIX shell; but you'll see similar behavior when working on the Windows command line, although backslashes are handled differently.

BOTTOM LINE: Don't add code to your solution that attempts to process those shell metacharacters—that's the job of the shell, not your program!

An admonishment/HINT about argument handling

I've seen many students turn command line argument handling into an incredibly complicated mess. Don't do that! Here's an easy way to process arguments in `xfield`: Iterate over the elements in `ARGV`. If the argument starts with `"-s"` or `"-d"` then save the rest of the string for later use. If `argument.to_i` produces something other than zero, then add the value (as an integer) to an array that specifies what's to be printed for each line. If `argument.to_i` produces zero, add `argument` to that same array. That's about 15 lines of simple code.

Note that Ruby's `ARGV` is the counterpart to `args` in a Java declaration like `void main(String args[])`, but unlike Java, the name `ARGV` is fixed.

A HINT on handling the textual argument/separator rule

One way to handle the textual argument/separator rule is to simply make a pass over the argument array and if two consecutive numbers are encountered, put the separator between them, as if the user had done that in the first place. For example, if the separator is `"."`, the specification

```
1 3 x 4 x -2 1
```

would be transformed into

```
1 . 3 x 4 x -2 . 1
```

A hint in a hint: Think about representing the above specification with this Ruby array:

```
[0, ".", 2, "x", 3, "x", -2, ".", 0]
```

Note the combination of `Fixnum`s and `String`s.

A lesson in LHtLaL

Recall that `xfield`'s output fields are separated by a single tab. (Use `"\t"`.) Let's demonstrate that by using a couple of `ruby` command line options:

```
% ruby xfield.rb 1 3 < a5/xf.1 | ruby -n -e 'puts $_.inspect'
"one\t1.0\n"
"two\t2.0\n"
"three\t3.0\n"
"four\t4.0\n"
"twenty\t20.0\n"
```

Use `man ruby` to learn about those `-n` and `-e` options! `$_` is a predefined global variable that holds "The last string read by the `Kernel` methods `gets` and `readline`." (from RPL)

Here's the lesson in LHtLaL: I could have used `cat -A` to see those tabs but I chose to build my Ruby skills by learning about `-n`, `-e`, and `$_`.

MID-TERM HINT: I'll ask you what LHtLaL stands for. Answer: Learning How to Learn a Language.

Problem 5. (3 points) `hudak.txt`

In *The Haskell School of Expression* the late Paul Hudak wrote,

> "The best news is that Haskell's type system will tell you if your program is well-typed before you run it. This is a big advantage because most programming errors are manifested as typing errors."

Do agree or disagree with his claim that "most programming errors are manifested as typing errors"? For ths problem you are to present an argument based on your full experience as a programmer that either supports his claim or refutes it. (Do not argue both sides!) Take all your programming experience into account, not just 372!

As usual, I'm looking for quality, not quantity but as a ballpark figure I imagine 200-400 words, as reported by `wc -w`, will be needed for a thoughtful answer.

As always, the `.txt` suffix on `hudak.txt` should be enough to tell you that I'm wanting a plain ASCII text file, not a Word document, PDF, etc.

Problem 6. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
```

```
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a5/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `aN.YYYYMMDD.HHMMSS.tz` You can run `a4/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `372s16` if you need to recover a file and aren't familiar with `tar`—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a5/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, here's what I see as of press time:

```
$ wc $(grep -v txt a5/delivs)
  13   36  237 longest.rb
  19   47  295 seqwords.rb
  37   89  760 minmax.rb
  60  150 1272 xfield.rb
 129  322 2564 total
```

My code has few comments.

Miscellaneous

Restrictions not withstanding, you can use any elements of Ruby that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Ruby slides 1-137.

If you're worried about whether a solution meets the restrictions, mail it to `372s16`—we'll be happy to look it over. But don't wait too long; there may be a crunch at the end. Remember: only `longest.rb` and `seqwords.rb` have restrictions.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) A # is comment to end of line, unless in a string literal or regular expression. There's no Ruby analog for `/* ... */` in Java and `{- ... -}` in Haskell but you can comment out multiple lines by making them an *embedded document*—lines bracketed with `=begin`/`=end` starting in column 1. Example:

```
=begin
    Just some
  comments here.
=end
```

Silly-looking, huh? I agree! (It looks like a construction that escaped from the 1960s.)

RPL 2.1.1 has more on comments.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 5 to 7 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put five hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached five hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment 6
Due: Friday, March 25 at 23:59:59

The Usual Stuff

Make an `a6` symlink that references `/cs/www/classes/cs372/spring16/a6`. Test using `a6/tester` (or `a6/t`). Use Ruby 2.2.4.

A note about problems 1-4

Each of the first four problems ask you to write an iterator, which is a single method, rather than a program. The write-up for the first iterator, `eo`, shows a couple of ways to approach edit-run cycle.

Each of the iterators has a "duck typing" specification that describes what the iterator requires of its argument(s), such as being subscriptable or having a `size` or `each` method.

IMPORTANT: The first test for each iterator is a Ruby program with a name like *`ITERATOR`*`-ap.rb`. The "ap" stands for **all points**—when grading, all the points for the problem will be based on whether you pass the "ap" test. The "ap" tests use dumbed-down classes that provide only the capabilities that the iterator is supposed to require. For example, `eo-ap.rb` uses a class named `Dumb` that provides only subscripting with `x[n]` and a `size` method. If you're passing some cases for an iterator but failing the "ap" test, then your implementation is requiring more capabilities of its argument(s) than it should be, according to the duck typing specification.

Problem 1. (2 points) eo.rb

Write a Ruby iterator `eo(x)` that yields every other element in `x`. `eo(x)` returns `x`.

Duck typing: `eo` only requires `x` to be subscriptable with `x[n]` and have a `size` method.

Usage:

```
>> eo([10,20,30,40]) {|v| puts v }
10
30
=> [10, 20, 30, 40]

>> eo("testing") {|c| print "'#{c}' "}
't' 's' 'i' 'g' => "testing"

>> sum = 0; eo((1..10).to_a) {|v| sum += v}; sum
=> 25

>> eo([]) {|v| puts v }
=> []
```

Here's a way to approach the edit-run cycle with `eo` and the three iterators that follow, `tkcycle`, `vrepl` and `mirror`:

Add the following line to your `~/.irbrc`:

```
$eo="eo.rb"
```

After restarting irb you can do this:

```
$ irb
>> load $eo
=> true
>> eo("abcd") {|c| puts c}
a
c
=> "abcd"
```

That works because load is a Ruby function. That contrasts with :load in ghci, which is an operation provided by the REPL, not the Haskell language.

You can edit in another window and then do load $eo to load up the latest.

Another angle to save a little typing is add a method like this to to your .irbrc:

```
def ld s
    load s.to_s + ".rb"
end
```

Use it like this:

```
$ irb
>> ld :eo
=> true
```

By giving ld a symbol, it's sort of like using a string literal that needs a quote on only one end. (*Atoms* in Lisp are similar.) ld converts its argument to a string, tacks on the ".rb" suffix, and calls load with the result. Again, we get this sort of flexibility because load is a Ruby method, but on the other hand, I still miss the filename completion that ghci provides.

Problem 2. (4 points) tkcycle.rb

Write a Ruby iterator tkcycle(x, sizes) that yields consecutive "slices" of x based on the integers in sizes. tkcycle cycles through the sizes in sizes until it runs out of elements in x. tkcycle always returns nil.

Duck typing: tkcycle only requires x to support a "slice" operation with x[start,length] and have a size method. sizes is expected to be a non-empty array of integers.

Examples:

```
>> tkcycle((1..10).to_a,[1,2]) {|s| p s}
[1]
[2, 3]
[4]
[5, 6]
[7]
[8, 9]
[10]
=> nil

>> tkcycle("just a test", [3]) {|s| puts "slice = #{s}"}
```

```
slice = jus
slice = t a
slice =  te
=> nil

>> tkcycle("just a test", [30]) {|s| puts "slice = #{s}"}
=> nil

>> tkcycle("just a test", [3,0,2]) {|s| puts ">#{s}<"}
>jus<
><
>t <
>a t<
><
>es<
```

Note that **tkcycle("just a test", [3])** above demonstrates that `tkcycle` never yields a partial result: After the third block invocation, with `s = " te"`, only `"st"` remains, and that's not enough for the next three-element yield.

Problem 3. (4 points) vrepl.rb

Write a Ruby iterator `vrepl(x)` that produces an array consisting of varying numbers of repetitions of values in `x`. The number of repetitions for an element is determined by the result of the block when the iterator yields that element to the block.

Duck typing: `vrepl` only requires `x` to have an `each` method.

```
>> vrepl(%w{a b c}) { 2 }
=> ["a", "a", "b", "b", "c", "c"]

>> vrepl(%w{a b c}) { 0 }
=> []

>> vrepl((1..10).to_a) { |x| x % 2 == 0 ? 1 : 0 }
=> [2, 4, 6, 8, 10]

>> i = 0
=> 0

>> vrepl([7, [1], "4"]) { i += 1 }
=> [7, [1], [1], "4", "4", "4"]
```

If the block produces a negative value, zero repetitions are produced:

```
>> vrepl([7, 1, 4]) { -10 }
=> []
```

Problem 4. (4 points) mirror.rb

Write a Ruby iterator `mirror(x)` that yields a "mirrored" sequence of values based on the values that `x.each` yields.

Duck typing: `mirror` only requires that `x` implement the iterator `each`.

The value returned by `mirror` is always `nil`.

```
>> mirror(1..3) { |v| puts v }
1
2
3
2
1
=> nil

>> mirror([1, "two", {a: "b"}, 3.0]) { |v| p v }
1
"two"
{:a=>"b"}
3.0
{:a=>"b"}
"two"
1
=> nil

>> mirror({:a=>1, :b=>2, :c=>3}) {|x| p x}
[:a, 1]
[:b, 2]
[:c, 3]
[:b, 2]
[:a, 1]
=> nil

>> mirror([]) { |v| puts v }
=> nil
```

Like the previous three iterators, `mirror` is a freestanding method.

Problem 5. (12 points) `calc.rb`

Write in Ruby a simple four-function line-oriented calculator that evaluates expressions composed of integer literals and variables, providing addition, subtraction, multiplication, and division. All operators have equal precedence. Evaluation is done in a strict left to right order. Control-D exits the program. Here are examples of expressions involving integer literals:

```
$ ruby calc.rb
? 3+4
7
? 3*4+5
17
? 3+4*5               Note that the addition is done first because it is the leftmost operator.
35
? 1/2*3+4
4
? 5/3
1
? 14324324324324242323*342343443234324
49038385111943393068867603094652
? ^D
$
```

Variables are created with assignments. Variables begin with a letter and are followed by zero or more letters or digits. Variables have a default value of zero. The result of an assignment is the value assigned.

```
$ ruby calc.rb
? x=7
7
? yval=10
10
? z
0
? x=x+yval+z
17
? yval=x+yval
27
? yval
27
? big=1111111111111111*1111111111111111
123456790123456787654320987654321
? big=big/big
1
```

Assignments only appear as the first operation on a line and consist of a variable followed by an equals sign followed by an expression. You won't see something like `x=y=3` or `x+y=0`.

Note that while the arithmetic operators are done in strict left-to-right order, the assignment, if any, is done last.

Input lines consist solely of letters, digits, and these five symbols: `+*-/=`. Assume all expressions are well formed; you won't see something like `x==3` or `+10/5+`. If a string starts with a letter, it is a variable; you won't see something like `15x`. There is no negation; you won't see something like `x=-10` or `3*-4`. Division by zero is not supported. There will be no empty lines in the input.

*Implementation note*s

Regular expressions are handy in a couple of places in `calc.rb`. As of press time we're just getting into regular expressions, so I'm going to give a couple of pieces of code to use as-is. The first is a method that can be used to see if a string is a non-negative integer:

```
def isInt(s)
    !!(s =~ /^\d+$/)
end
```

The match simply requires that `s` consist of nothing but digits. I chose to use a couple of "not"s to produce `true` or `false` rather than a truthy match position or a falsy `nil`, mainly for a prettier example below.

The second piece of regular expression code is a particular invocation of `String#scan`, to break up an input line:

```
>> line = "x2=3*val+40-500"

>> line.scan(/\w+|\W+/)
=> ["x2", "=", "3", "*", "val", "+", "40", "-", "500"]
```

Just for fun, let's combine that with `isInt`:

```
>> line.scan(/\w+|\W+/).map {|s| isInt(s)}
=> [false, false, true, false, false, false, true, false, true]
```

WARNING: `Kernel#eval` might look like a quick shortcut to a solution for this problem but using it can lead to some headaches. My recommendation is that you avoid `eval` for this problem. However, I do recommend that you use `Object#send`! It works like this:

```
>> 4.send("+", 3)
=> 7

>> 10.send("-", 4)
=> 6
```

Problem 6. (25 points) `switched.rb`

The U.S. Social Security Administration makes available yearly counts of first names on birth certificates back to 1885. Over time, some names change from predominantly male to predominantly female or vice-versa. For this problem you are to create a Ruby program `switched.rb` to look for names that change from predominantly male to predominantly female in given spans of years.

`switched.rb` takes two command-line arguments: a starting year and an ending year. Here's a run:

```
% ruby switched.rb 1951 1958
             1951   1952   1953   1954   1955   1956   1957   1958
Dana         1.19   1.20   1.26   1.29   1.00   0.79   0.67   0.64
Jackie       1.40   1.29   1.14   1.13   1.11   0.94   0.72   0.57
Kelly        4.23   2.74   3.73   2.10   2.32   1.77   0.98   0.51
Kim          2.58   1.82   1.47   1.08   0.61   0.30   0.17   0.12
Rene         1.43   1.32   1.15   1.24   1.13   0.88   0.87   0.89
Stacy        1.06   0.81   0.62   0.47   0.44   0.36   0.29   0.21
Tracy        1.51   1.14   1.02   0.73   0.56   0.55   0.59   0.59
```

First, note that all numbers in the leftmost column are greater than one and all numbers in the rightmost column are less than one.

The 1.19 for Dana in 1951 indicates that in 1951 there were 1.19 times as many male babies named Dana as there were female babies named Dana. We can see that in `a6/yob/1951.txt`, which has the 1951 data:

```
$ grep Dana, a6/yob/1951.txt
Dana,F,1076
Dana,M,1277
```

The format of the `a6/yob/`*`YEAR`*`.txt` dat files is simple: each line has the name, sex, and the associated count, separated by commas.

Note that the argument to `grep`, "`Dana,`" has a trailing comma so that "Danae" doesn't turn up, too.

By 1958 things had changed—there were only .64 males named Dana for every female named Dana:

```
$ grep Dana, a6/yob/1958.txt
```

```
   Dana,F,2388
   Dana,M,1531
```

`switched.rb` reads the `a6/yob/YEAR.txt` files for all the years in the range specified by the command line arguments and looks for names for which the male/female ratio is > 1.0 in the first year and < 1.0 in the last year. For all the names it finds, it prints the male/female ratio for all the years from the first year through the last year. Names are printed in alphabetical order.

As a specific example, Dana is included in the list for 1951 through 1958 because males/females in 1951 was 1.19 (> 1.0) and males/females in 1958 was 0.64 (<1.0). The ratios for the middle years are not examined to decide whether to include a name; they are shown only to provide a more complete picture of the data between the endpoints.

Note that there's a big shift for Kim from 1954 through 1957. I wonder if that's because the actress Kim Novak had a breakout role in 1955's *Picnic*.

If no names meet the criteria, `switched` prints "`no names found`" and exits by calling `exit`.

```
   $ ruby switched.rb 2011 2012
   no names found
```

IMPORTANT: To eliminate the less significant results, a name is included only if both the male and female counts in both the first and last year are greater than or equal to 100. By that criteria the name Lavern is not included for 1949-1951, and no other names turn up, either:

```
   % ruby switched.rb 1949 1951
   no names found
```

Here's the underlying data:

```
   $ grep Lavern, a6/yob/yob1949.txt
   Lavern,F,93
   Lavern,M,121

   $ grep Lavern, a6/yob/yob1951.txt
   Lavern,F,95
   Lavern,M,86
```

There was a M/F shift from 121/93 in 1949 to 86/95 in 1951 but because not all four of those counts are >= 100, Lavern is not included. There's no `grep` shown above for 1950 because that data is inconsequential: inclusion is determined solely based on counts for the first and last year.

It's interesting to combine `switched` with a `bash` `for`-loop that runs the program with a gradually shifting range. When your `switched.rb` is done, try this:

```
   for i in $(seq 1940 2005); do ruby switched.rb $i $(($i+9)); echo ===; done
```

Two obvious extensions to `switched` would be command-line options to adjust the 100-baby minimum and to look for female to male flips for a name. You might find those interesting to implement and experiment with, but neither are required.

`switched.rb` does no error handling whatsoever. Behavior is only defined in the case of being given two command line arguments in the range of 1885 to 2014, and the first must be less than the second.

Implementation notes for `switched`

I intend this problem to be an exercise in using the `Hash` class. I encourage you to devise a data structure yourself but in case you run into trouble, here are a few thoughts on my approach to the problem: http://www.cs.arizona.edu/classes/cs372/spring16/a6/switched-hint.html

It's easy to drown in the data on a problem like this. You might start by having your code that reads the `YEAR.txt` files discard data for everybody but "Dana" and then use "`p x`" (equivalent to "`puts x.inspect`") to dump out your data structure. Then try adding in a male-only and a female-only name, like "Delbert" and "Mimi" in 1951. Alternatively, you might edit down some data files to just a few lines of interest. (After copying any of the `a6/yob` files into your directory, use `chmod 600 FILE` so that you can edit it; `cp` will have left it as mode `444`—read only.)

Watch out for bugs related to integer division. (Use `.to_f` to get a `Float` when needed.)

Use `File.open` to produce a `File` object whose `gets` method can be used to read lines. Example:

```
$ cat fileio.rb
year = ARGV[0]

f = File.open("a6/yob/#{year}.txt")

count = 0
while line = f.gets
    count += 1
end

f.close

puts "read #{count} lines"

$ ruby fileio.rb 2001
read 30258 lines
```

Alternatively, you could use `f.readlines()` to produce an array of all the lines in the file with a single call or `f.each { ... }` to process each line with the associated block.

The M/F ratios are formatted using a `%7.2f` format with `printf`, demonstrated on the command line with `ruby -e`:

```
$ ruby -e 'printf("%7.2f\n", 1277.0/1076.0)'
   1.19
```

Names are left-justified in a 10-wide field using a `printf` format of `%-10s`.

You may have questions about the data files. Before mailing to us or posting on Piazza, take a look at the data files and see if you can answer the question yourself. The files are in `a6/yob`. Those same files will be used for testing when grading.

You can download http://www.cs.arizona.edu/classes/cs372/spring16/a6/yob.zip for testing on your own machine. In the same directory as your `switched.rb`, make a directory named `a6` and then unzip `yob.zip` in that directory to produce a structure compatible with the `File.open` above.

Problem 7. (ZERO points) `pancakes.rb`

Let's see who will write a Ruby version of the Haskell pancake printer from assignment 4 for zero points!

The Ruby version is a program that reads lines from standard input, one order per line, echoes the order, and then shows the pancakes.

Example:

```
$ cat a6/pancakes.1
3 1 / 3 1 5
3 1 3
1 5/          1 1 1/11 3 15     /3 3 3     3/1
1
$ ruby pancakes.rb < a6/pancakes.1
Order: 3 1 / 3 1 5

     ***
***   *
 *  *****

Order: 3 1 3

***
 *
***

Order: 1 5/          1 1 1/11 3 15     /3 3 3     3/1

                        ***
      *   ***********   ***
  *   *       ***       ***
***** * *************** *** *

Order: 1

*

$
```

A blank line is printed after the `Order:` line and again after the stacks.

Assume that input lines consist exclusively of integers, spaces, and slashes, which separate stacks. Assume that there is at least one stack. Assume all stacks have at least one pancake. Assume all widths are greater than zero. Assume the input is well-formed—you won't see something like "`1 / / 3`" or "`/ 3 /`". Assume there are no empty lines in the input.

If an order specifies an even-width pancake, the message shown below is printed. Processing then continues with the next order in the input, if any.

```
$ ruby pancakes.rb < a6/pancakes.2
Order: 1 3 1 / 1 2 3

Even-width pancake.  Order ignored.

Order: 51 49
```

```
**************************************************
 ************************************************

$
```

In case you want to play "Beat the Teacher", I'll tell you that it took me about 25 minutes to write `pancakes.rb`, sketching on paper included. If you care to, let me know how long it takes you. Think about it all you want to but start the clock the moment a tangible artifact is produced, like a mark on a piece of paper.

Problem 8. Extra Credit **observations.txt**

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a6/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `aN.`*`YYYYMMDD.HHMMSS`*`.tz` You can run `a6/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `372s16` if you need to recover a file and aren't familiar with `tar`—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a6/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, with comments stripped, here's what I see as of press time:

```
$ wc $(grep -v txt a6/delivs)
   8   17   93 eo.rb
  19   37  332 tkcycle.rb
   8   29  158 vrepl.rb
  17   23  205 mirror.rb
  35   75  667 calc.rb
  57  142 1241 switched.rb
  39  112  851 pancakes.rb
 183  435 3547 total
```

Miscellaneous

You can use any elements of Ruby that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Ruby slides 1-200.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 7 to 9 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put seven hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached seven hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment 7
Due: Friday, April 8 at 23:59:59

Option: Make Your Own Assignment!

If you've got an idea for something you'd like to write in Ruby, you can propose that as a replacement for some or all of the problems on this assignment. To pursue this option send me mail with a brief sketch of your idea. We'll negotiate on points and details.

The Usual Stuff

Make an `a7` symlink that references `/cs/www/classes/cs372/spring16/a7`. Test using `a7/tester` (or `a7/t`). Use Ruby 2.2.4.

Don't hesitate to dig into the test programs

Most of the tests for the problems on this assignment are in the form of Ruby programs that exercise a number of cases. For example, there's only one test for `label.rb`. It's this: `ruby a7/label1.rb`

The tests for `vstring.rb` look like this:

```
ruby a7/vstring1.rb 1r
ruby a7/vstring1.rb 1m
...
```

The command line argument specifies the block of tests to run.

In some cases you may need to dig around in those test programs to figure out exactly what code is being executed for a failing test. In some cases it may be useful to copy the code into your directory and hack it up, maybe adding `Kernel#p` calls or varying the amount of loop iterations to help narrow down a bug.

In various places in the output for the `label` and `VString` tests you'll see something like this:

```
Line 35 in a7/label1.rb:
label([a,b,c]):
   a1:[a2:[],a3:[a2,a2],a4:[a3,a3]]
```

That first line tells us that the test originates at line 35 in `a7/label1.rb`. Here's that line:

```
sv("label([a,b,c])", bb)
```

`sv`, standing for "show values" is a helper method that uses `eval` to evaluate the expression specified by the first argument. The second argument, `bb`, is the current set of variable bindings, which are used by `eval`.

Problem 1. (12 points) `label.rb`

`Array#inspect`, which is used by `Kernel#p` and by `irb`, does not accurately depict an array that contains multiple references to the same array and/or references itself. Example:

```
>> a = []
>> b = [a,a]

>> p b
[[], []]
```

By simply examining the output of `p b` we can't tell whether `b` references two distinct arrays or has two references to the same array.

Another problem is that if an array references itself, Ruby "punts" and shows "`[...]`":

```
>> a = []

>> a << a

>> p a
[[...]]
```

The problems continue if we then get a `Hash` involved:

```
>> h = {}

>> h[a] = h

>> p h
{[[...]]=>{...}}
```

For this problem you are to write a method `label(x)` that produces a labeled representation of x, where x is a string, number, array or hash. Arrays and hashes may in turn reference other arrays and hashes, and be cyclic.

Here's what `label` shows for the first case above.

```
>> a = []; b = [a,a]

>> puts label(b)
a1:[a2:[],a2]
```

The outermost array is labeled as `a1`. Its first element is an empty array, labeled `a2`. The second element is a reference to that same empty array. Its contents are not shown, only the label `a2`. For reasons described later, we'll use `puts label(...)` to show the string that `label` produces.

Here's another step, and the result:

```
>> c = [b,b]

>> puts label(c)
a1:[a2:[a3:[],a3],a2]
```

Note that the label numbers are not preserved across calls. The array that this call labels as `a3` was labeled as `a2` in the previous example.

To explore relationships between the contents of `a`, `b`, and `c` we could wrap them in an array:

```
    >> puts label([a,b,c])
    a1:[a2:[],a3:[a2,a2],a4:[a3,a3]]
```

Here's a simple cyclic case. The third element in `a` is a reference to `a`; representing that reference with a label lets us see the cycle.

```
>> a = []

>> a = [10,20]

>> a << a

>> puts label(a)
a1:[10,20,a1]
```

Let's try a simple hash:

```
>> h = {}

>> h["one"] = 1

>> h[2] = "two"

>> puts label(h)
h1:{"one"=>1,2=>"two"}
```

Note that hashes are labeled as "`hN`". The key/value pairs are shown with "thick" arrows. Pairs are separated with commas, and curly braces surround the list of pairs.

Let's add some arrays into the mix:

```
>> h = {}

>> a = [2,2,2]

>> a << a

>> h["twos"] = a

>> h["words"] = %w{just a test}

>> puts label(h)
h1:{"twos"=>a1:[2,2,2,a1],"words"=>a2:["just","a","test"]}
```

Note that arrays and hashes have separate numbering: the above labeling shows both `h1` and `a1`, for example.

Let's try `h[h] = h`:

```
>> h[h] = h

>> puts label(h)
h1:{"twos"=>a1:[2,2,2,a1],"words"=>a2:["just","a","test"],h1=>h1}
```

`label(h)` eventually reaches the key/value pair made by `h[h] = h`. Because `h` has already been labeled as `h1`, the presence of that key/value pair is shown as `h1=>h1`.

Another example:

```
>> a = [1,2,3]

>> a << [[[a,[a]]]]

>> a << a

>> puts label(a)
a1:[1,2,3,a2:[a3:[a4:[a1,a5:[a1]]]],a1]
```

One more example:

```
>> a = [1,2,3]

>> b = {"lets" => "abc", 1 => a}

>> 3.times { a << b }

>> a << "end"

>> puts label([a,b,{100=>200}])
a1:[a2:[1,2,3,h1:{"lets"=>"abc",1=>a2},h1,h1,"end"],h1,h2:{100=>200}]
```

The trivial case for `label(x)` is that `x` is not an array or hash. If so, `label` returns `x.inspect`:

```
>> puts label(4)
4

>> puts label("testing")
"testing"
```

Here are some simple cases that are good for getting started:

```
>> puts label([7])
a1:[7]

>> puts label([[7]])
a1:[a2:[7]]

>> puts label([[70],[80,"90"]])
a1:[a2:[70],a3:[80,"90"]]
```

Keep in mind that your solution must be able to accommodate an arbitrarily complicated structure but the only types you'll encounter are numbers, strings, arrays and hashes.

This routine is a simplified version of the `Image` procedure from the Icon library. I've looked around and asked around for something similar in Ruby. I haven't found anything yet but it may well exist. If you find such a routine, which would trivialize this problem, you may not use it. However, you may study it and then, based on what you've learned, create your own implementation. And, tell me about your discovery!

The above examples use `puts label(...)` rather than showing the result of `label(...)`. Let's look at a `label` call without `puts`:

```
>> label([1,"two",["3"]])
```

```
=> "a1:[1,\"two\",a2:[\"3\"]]"
```

`label` returns a string and `irb` uses `inspect` to show an unambiguous representation of results, so any quotes in the result are escaped. Using `puts` lets us avoid that clutter:

```
>> puts label([1,"two",["3"]])
a1:[1,"two",a2:["3"]]
=> nil
```

Implementation notes

I think of this as a fairly hard problem to solve given only the above and what you've seen in class, but for those who wish to have a challenge, I'll say nothing here about how to approach it. If you have trouble getting started, however, see http://www.cs.arizona.edu/classes/cs372/spring16/a7/label-hint.html

IMPORTANT: You must match the sequence of label numbers that my solution produces. That essentially requires you to traverse the structure in the same order I do, which is depth-first. Here's an example that evidences that depth-first traversal:

```
>> puts label([[[10]],[[21,22]]])
a1:[a2:[a3:[10]],a4:[a5:[21,22]]]
```

Note that the deeply nested `[10]` was labeled with `a3` before the second element of the top-level list was labeled with `a4`.

Also, note that I iterate through key/value pairs in a hash using `h.each {|k,v| ... }`

Problem 2. (15 points, unevenly distributed across four sub-problems) `re.rb`

In this problem you are to write several methods. **Each returns a regular expression that matches the specified strings (no more, no less) and, in some cases, creates named groups.**

Here is an example specification:

> *Write a method `letsdigs_re` that produces a regular expression that matches strings that consist of one or more letters followed by three or more digits. The named groups `lets` and `nums` are set to the letters and digits, respectively. A dash may optionally appear between the two. If so, the named group `dash` is non-empty (i.e., not `nil` and not the empty string).*

Here's a solution: (See slide 231 if you don't recognize the `{3,}` construct.)

```
def letsdigs_re
    /^(?<lets>[a-z]+)(?<dash>-?)(?<digs>\d+{3,})$/
end
```

Because the regular expression is the last expression in `letsdigs_re`, it is the return value.

`spring16/ruby/smg.rb` has a variant of `show_match` (slide 213) that also shows named groups. Maybe add that method to your `~/.irbrc`. Let's test `letsdigs_re` with `smg`:

```
>> smg(letsdigs_re, "abc123")
"<<abc123>>"
lets = <<abc>>, dash = <<>>, digs = <<123>>

>> smg(letsdigs_re, "abc-12")
```

```
no match

>> smg(letsdigs_re, "abc-1234")
"<<abc-1234>>"
lets = <<abc>>, dash = <<->>, digs = <<1234>>
```

A great resource for developing regular expressions is rubular.com. Here's the example above on Rubular: http://rubular.com/r/vPnFMgzPoB. Note that the "Your test string:" window has three test cases, one per line. There is a little bad news: Rubular doesn't seem to provide any way to build up a regular expression like shown on slide 246.

Here's an older resource: a video example of using `show_match` to gradually develop a regular expression: http://screencast.com/t/FO7OOIScCR39. (Yes, I should produce a version of it that uses Rubular!)

The set of tests used for grading `re.rb` will be the set of tests used by `a7/tester`, and that set will be finalized by noon on March 26.

Here are the methods you are to write for this problem:

(a) (1 point) Write a method `phone_re` that produces a regular expression that matches strings that are phone number in any of these forms: `555-1212`, `800-555-1212`, and `(800) 555-1212`.

(b) (2 points) Write a method `sentence_re` that produces a regular expression that matches sentences, as follows: Sentences must begin with a capital letter. Sentences are composed of one or more words. Words are separated by exactly one blank. The sentence must end with a period, question mark, exclamation mark, or one of the two strings `"!?"` and `"?!"`.

Two good sentences: `"I shall test this!"`, `"Xserwr AAA x."`

A bad sentence: `"it works!"` (Doesn't start with a capital.)

See http://www.cs.arizona.edu/classes/cs372/spring16/a7/sentence-hint.html if you have trouble getting started with this problem.

(c) (3 points) Write a method `hours_re` that produces a regular expression that matches a specification of one or more office hours periods, like "`MWF 12:00-13:00`", or "`T-H 9:30-12:30, F 19:00-0:00`". Days are specified with one or more letters in the set `[MTWHF]`, or a range from one day to another. Hours are in the range `0:00-23:45`, with 5-minute resolution. Times before `10:00` can optionally specified with a leading zero, like `09:45`. Multiple periods are separated by a comma and a space.

(d) (3 points) Here are some examples of `ls -l` output:

```
-rw-r-----   1 whm whm    543 Mar 14 18:19 ttl.rb
-rw-r--r--   1 whm whm   6555 Dec 10  2009 w.dat
lrwxrwxrwx   1 whm whm      6 Mar 19 20:56 a7/t -> tester
drwx------ 183 whm empty 1306 Mar 21 20:21 /home/whm/.
drwx--x--x  51 whm wheel  318 Jan 30 22:13 /x/closed
```

The first character indicates file type—`d` for directory, `l` for symbolic link, `-` for regular files. There are other types, too.

The next nine characters, which are three three-character groups, show the permissions. The first group of three characters is "user" permissions—what the owner of the file is permitted to do with the file. The next three characters are the group permissions. The third group of three is "other" permissions—what users who aren't the owner nor are in the appropriate group can do with the file. Ignoring some special cases, the first letter of the three-character groups is always either 'r' or '-'; the second is always 'w' or '-'; and the third is always 'x' or '-'. You'll never see something like 'rrw' or 'w-w'.

For this problem you're to write a method perms_re that produces a regular expression that matches lines of ls -l output for files whose "other" permissions (the third group of three) is not "---" **and** the file is not a symbolic link. In the lines above, w.dat and /x/closed meet that criteria.

When the match is successful, the named group perms should contain the three characters of "other" permissions (like "r--", "rwx", or "--x"), and the named group name should contain the file name, like "w.dat".

Assume that file names do not contain blanks, but you might find it interesting to consider handling that case, too.

Note that the format of the first ten characters never varies for ls -l output, but field widths for the rest of the line can vary. The upshot of this is that you can't assume that the file name begins in any particular column.

(e) (3 points) For this problem you're to write a method vr_re that matches a string if it is a Vim range specification. A Vim range specification can be a single line specification or two line specifications separated by a comma.

We'll handle the following line specifications

A simple line number, like 15
A dot (.) for the current line, or $ for the last line
A dot or $ followed by +N or -N, where N is a positive integer
A regular expression enclosed in slashes, like /abc/ or /^x/. **Assume** the regular expression doesn't contain any escaped slashes, like in /a\/b/.

Here are some valid range specifications:

```
10
10,20
.,20
20,$
/abc/,.
.-5,.+10
/begin/,/end/
```

When a match succeeds, the named groups from and to should be set, if two line specifications are present. If there's only one, then $~["to"] should be nil or the empty string.

(f) (3 points) Write a method path_re that produces a regular expression that matches UNIX paths and sets the named group dir to the directory name, base to the filename, minus extension, and ext to the extension, which is defined as everything in the filename to the right of the leftmost dot. If an element is not present, then $~[*whatever-group*] should be nil or the empty string.

Examples: (excerpt of output from `ruby a7/re1.rb path < a7/re.path`)

```
path: 'longest.java', dir = '', base = 'longest', ext = 'java'

path: '/etc/passwd', dir = '/etc/', base = 'passwd', ext = ''

path: '/cs/www/classes/cs372/spring16/tester/Test.rb',
dir = '/cs/www/classes/cs372/spring16/tester/', base = 'Test',
ext = 'rb'

path: '/home/whm/.bashrc', dir = '/home/whm/', base = '', ext =
'bashrc'
```

The above is skimpy on examples but you'll find plenty in `a7/re.*`. Those are input files for `a7/re1.rb`.

To be clear, `re.rb`, should consist of six methods. It should look something like this:

```
def phone_re
     ...
end

...four more here...

def path_re
    ...
end
```

<u>A note on testing `re.rb`</u>

For testing, "`a7/t re`" will test all the regular expressions but you can use the tester's `-t` option to test just one of the regular expressions. Example:

```
$ a7/t re -t phone
```

Along with "`phone`" there's "`sen`", "`hours`", "`perms`", "`vr`", and "`path`".

Problem 3. (22 points) `vstring.rb`

For this problem you are implement a hierarchy of four Ruby classes: `VString`, `ReplString`, `MirrorString`, and `IspString`. `VString` stands for "virtual string"—**<u>these classes create the illusion of very long strings but relatively little data is needed to create that illusion</u>**.

`VString` serves as an abstract superclass of `ReplString`, `MirrorString`, and `IspString`; it simply provides some methods that are used by those subclasses.

`ReplString` represents strings that consist of zero or more replications of a specified string. Example:

```
$ irb
>> load "vstring.rb"

>> s1 = ReplString.new("abc", 2)
=> ReplString("abc",2)
```

Note that `irb` used `s1.inspect` to produce the string '`ReplString("abc",2)`', that shows the type, base string, and replication count.

VString subclasses support only a handful of operations: size, [n], [n,m], inspect, to_s, and each. The semantics of [n] and [n,m] are the same as for Ruby's String class with one exception, described below.

Here are some examples:

```
>> s1.size           => 6

>> s1.to_s           => "abcabc"

>> s1.to_s.class     => String

>> s1[0]             => "a"

>> s1[2,4]           => "cabc"

>> s1[-5,2]          => "bc"

>> s1[-3,10]         => "abc"

>> s1[10]            => nil
```

A ReplString can represent a *very* long string:

```
>> s2 = ReplString.new("xy", 1_000_000_000_000)
=> ReplString("xy",1000000000000)

>> s2.size                => 2000000000000

>> s2[-1]                 => "y"

>> s2[-2,2]               => "xy"

>> s2[1_000_000_000]      => "x"

>> s2[1_000_000_001]      => "y"

>> s2[1_000_000_001,7]    => "yxyxyxy"
```

Some operations are impractical on very long strings. For example, s2.to_s would require a vast amount of memory; but if the user asked for it, we'd let it run. Similarly, s2[n,m] has the potential to produce an impractically large result.

A MirrorString represents a string concatenated with a reversed copy of itself. Examples:

```
>> s3 = MirrorString.new("1234")       => MirrorString("1234")

>> s3.size                             => 8

>> s3.to_s                             => "12344321"

>> s3[-1]                              => "1"
>> s3[2,4]                              => "3443"
```

An IspString represents a string with instances of another string interspersed between each character:

```
>> s4 = IspString.new("xyz", "...")    => IspString("xyz","...")

>> s4.to_s                             => "x...y...z"

>> s4[0]                               => "x"

>> s4[1]                               => "."

>> s4[s4.size-1]                       => "z"
```

In the above examples the strings used in the subclass constructors are instances of String but they can be an arbitrary nesting of VStrings, too:

```
>> s5 = IspString.new(MirrorString.new("abc"),
ReplString.new(".",3))
=> IspString(MirrorString("abc"),ReplString(".",3))

>> s5.to_s
=> "a...b...c...c...b...a"

>> s6 = ReplString.new(s5,1_000_000_000_000_000)
=>ReplString(IspString(MirrorString("abc"),ReplString(".",3)),100000
0000000000)

>> s6.size
=> 21000000000000000

>> s6[s6.size-20,100]
=> "...b...c...c...b...a"

>> s6[(s5.size)*(1_000_000_000_000_000-2),50]
=> "a...b...c...c...b...aa...b...c...c...b...a"
```

<u>To emphasize that the "V" in VString stands for "virtual", consider this interaction:</u>

```
>>s=MirrorString.new(ReplString.new("abc",10000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00))
=>MirrorString(ReplString("abc",100000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000))

>> s.size
=>6000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000

>> s2 = IspString.new(s,s)
=>IspString(MirrorString(ReplString("abc",10000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00)),MirrorString(ReplString("abc",100000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000)))

>> s2.size.to_s.size
=> 328
```

The iterator `each` is available for `ReplString`, `MirrorString`, and `IspString`. It produces all the characters in turn, as one-character strings. It returns the `VString` it was invoked on.

```
>> s1 = ReplString.new("abc",2)  => ReplString("abc",2)

>> s1.each {|x| puts x}
a
b
c
a
b
c
=> ReplString("abc",2)
```

__**Be sure to '`include Enumerable`' in `VString`**__, so that methods like `map`, `sort`, and `all?` work:

```
>> MirrorString.new("abc").map { |c| c }
=> ["a", "b", "c", "c", "b", "a"]

>> MirrorString.new(ReplString.new("abc",2)).sort
=> ["a", "a", "a", "a", "b", "b", "b", "b", "c", "c", "c", "c"]

>> MirrorString.new(
             ReplString.new("a", 100000)).all? { |c| c == "a" }
=> true
```

__**Using `Ranges` to subscript `VStrings` is not supported:**__

```
>> s1 = MirrorString.new("ab")   => MirrorString("ab")
>> s1[0..-1]
NoMethodError: ...
```

I won't test with ranges.

__**TL;DR**__

Here's a quick summary of what's required of the `VString` subclasses and their constructors:

- `ReplString.new(s, n)` where `s` is a non-empty `String` or a `VString` and `n > 0`.
- `MirrorString.new(s)` where `s` is a non-empty `String` or a `VString`.
- `IspString.new(s1,s2)` where `s1` and `s2` are non-empty `Strings` or are `VStrings`.
- `size`, `inspect`, and `to_s` methods
- Subscripting with `[n]` and `[start,len]`, with the same semantics as for Ruby's `String` class.
- Subscripting with `[`*`Range`*`]` is __**not**__ supported.
- `VString` implements the iterator `each` and `includes` `Enumerable`.

__There is one exception to matching the semantics of `String` when subscripting with `[start,len]`.__ Note this behavior of `String`:

```
>> s="abc"          => "abc"

>> s[3]             => nil

>> s[3,1]           => ""

>> s[4,1]           => nil
```

You might find it interesting to think about why `String`s have that behavior but we won't match it with `VString`. Instead, if `start` is out of bounds, `nil` is produced:

```
>> s = ReplString.new("abc",2)    => ReplString("abc",2)

>> s[5,10]                        => "c"

>> s[6,10]                        => nil
```

Implementation Notes

VERY IMPORTANT: *Implement as much functionality as possible in* `VString`

It may help to imagine that there will eventually be dozens of `VString` subclasses instead of just the three specified here. Having that mindset, and wanting to write as little code as possible to implement those dozens of subclasses, should motivate you to **do as much as possible in `VString` and as little as possible in the subclasses**.

My implementations of `ReplString`, `MirrorString`, and `IspString` have only **FOUR** methods: `initialize`, `size`, `inspect`, and `char_at(n)`. All of those methods are tiny—one or two short lines of code (with one exception).

Implementation of subscripting

`VString` itself should have this method:

```
def [](start, len = 1)
    ...
end
```

That defaulted second argument allows this one method to handle both `s[n]` and `s[start,len]`.

Implement this method in terms of calls to `size` and `char_at`, which in turn will resolve to the implementation of those methods in the three subclasses.

With deeply nested constructions of `VString` subclasses there's a potential of producing exponential behavior with subscripting if your `[]` method contains more than one call to `char_at`. Limit your `[]` method to one `char_at` call. (Save the value with `c = char_at(...)` if you need to use it in multiple places.)

Don't get tangled up considering various cases of nesting

Watch out for thinking like this: "What if I've got a `ReplString` that holds a `MirrorString` of a `MirrorString` of a `ReplString` of an `IspString` made of ..."

Instead, just keep in mind that what you can count on is that the values used in the VString constructors support size, s[n], s[start,len], and inspect. Have a duck-typing mindset and write code that uses those operations. (Yes, there's to_s, too, but it's problematic with long strings; avoid using it.)

Those double-quotes in inspect

Getting the double-quotes right for inspect can be a little tricky. Start by getting the following examples working:

```
>> s = ReplString.new("a\nb", 2)
=> ReplString("a\nb",2)

>> s2 = MirrorString.new("x\ty")
=> MirrorString("x\ty")

>> s3 = ReplString.new(s2,10)
=> ReplString(MirrorString("x\ty"),10)
```

When grading, tests will only exercise the VString subclasses; VString will not be tested directly. (Note that none of the examples above do anything with VString itself.)

Saving a little typing when testing

To save myself some typing when testing, I've got these lines at the end of my vstring.rb:

```
if !defined? RS
    RS=ReplString
    MS=MirrorString
    IS=IspString
end
```

With those in place, I can do something like this:

```
>> s = IS.new(MS.new(RS.new("abc",2)),"-")
=> IspString(MirrorString(ReplString("abc",2)),"-")
```

Implementing each

In retrospect, I believe the slides don't say enough how to put an each method in a class like VString. I considered dropping it altogether but it's a useful example for VString in general, so here is VString#each, for you to drop into your vstring.rb as-is:

```
def each
    for i in 0...size
        yield self[i]
    end
    self
end
```

Note that self[i] will call VString#[], which will in turn use the char_at implementation in the subclass for the instance at hand.

Put all four classes in vstring.rb

Put the code for all four classes in `vstring.rb`. `VString` needs to be first.

Problem 4. (12 points) `gf.rb`

Here's something I saw in a book:

```
class Fixnum
    def hours; self*60 end  # 60 minutes in an hour
end

>> 2.hours       => 120

>> 24.hours      => 1440
```

You are to write a Ruby method `gf(spec)` that dynamically adds (i.e., "monkey patches") a number of such methods into `Fixnum`, as directed by `spec`. Example:

```
gf("foot/feet=1,yard(s)=3,mile(s)=5280")
```

Using `Kernel#eval`, the above call to `gf` adds nine methods to `Fixnum`. Here are six of them: `foot`, `feet`, `yard`, `yards`, `mile`, `miles`. Respectively, on a pair-wise basis, those methods produce the `Fixnum` (which is `self`) multiplied by `1`, `3`, and `5280`.

```
% irb -r ./gf.rb
>> gf("foot/feet=1,yard(s)=3,mile(s)=5280")       => true

>> 1.foot          => 1

>> 10.feet         => 10

>> 5.yards         => 15

>> 3.miles         => 15840

>> 8.mile          => 42240

>> 1.feet          => 1
```

It would perhaps be useful to detect plurality mismatches like `8.mile` and `1.feet` and produce an error but that is not done.

In addition to the six methods mentioned above, these three are added to `Fixnum`, too: `in_feet`, `in_yards`, and `in_miles`:

```
>> (30.feet+10.yards).in_yards   => 20.0

>> 10_000.feet.in_miles          => 1.89393939393939
```

Two more examples:

```
>> gf("second(s)=1,minute(s)=60,hour(s)=3600,day(s)=86400")=> true

>> (12.hours+30.minutes).in_days       => 0.520833333333333

>> gf("inch(es)=1,foot/feet=12")       => true
>> 18.inches.in_feet                    => 1.5

>> 1.foot                               => 12

>> 1.foot.in_inches                     => 12.0
```

Note that methods later generated by `gf` simply replace earlier methods of the same name. After the two calls `gf("foot/feet=1")` and `gf("foot/feet=12")`, `1.foot` is `12`.

An individual mapping must be in the form *`singular/plural=integer`* or *`singular(pluralSuffix)=integer`*. None of the parts may be empty. Mappings are separated by commas. Only lowercase letters are permitted in the names. No whitespace is allowed. If any part of a specification is invalid, a message is printed and `false` is returned; but the result is otherwise undefined. Here is an example of the output in the case of an error:

```
>> gf("foot/feet=1,yards=3")
bad spec: 'foot/feet=1,yards=3'
=> false
```

Note that the error is not pin-pointed—the specification as a whole is cited as being invalid.

Here are more examples of errors:

```
gf("foot/feet=1,")           # trailing comma
gf("foot/feet=1.5")          # non-integer
gf("foot/=1")                # empty plural
gf("inch()=12")              # empty plural suffix
gf("foot/feet=1,Yard(s)=3")  # capital letter
```

This is NOT a restriction but to get more practice with regular expressions I recommend that your solution not use any string comparisons; use matches (=~) to break up the specification. And, using regular expressions will probably increase the likelihood that you accept exactly what's valid.

For this problem you are to use `eval` (slide 261+) to add the methods to `Fixnum`, but as the slides show, using `eval` to generate code based on data supplied by another person can be perilous! `eval` is a powerful tool but you've got to be careful when using it. Googling for `ruby eval` turns a lot of discussion about it, but be wary of those who say, "Never use `eval`!" Instead, recognize that `eval` is a powerful tool and use it with caution, weighing risks and benefits, and **ALWAYS** being careful to consider the source of all data that can possibly contribute directly or indirectly to a string that is passed to `eval`.

Keep in mind that you're writing a method named `gf`, not a program. Helper methods are permitted.

In assignment 3's write-up for `editstr` it was mentioned that the bindings for `x`, `len`, etc. are the beginnings of a simple internal DSL (Domain Specific Language). The list `[x 2, len, x 3, rev, xlt "1" "x"]` uses the facilities of Haskell to specify computation in a new language that's specialized for string manipulation. Similarly, this problem provides another example of an internal DSL by using the facilities of Ruby to express computations involving unit conversions.

Problem 5. (24 points) `optab.rb`

When writing this problem one of my favorite quotes came to mind:

> "Far better is it to dare mighty things, to win glorious triumphs, even though checkered by failure, than to take rank with those poor spirits who neither enjoy much nor suffer much because they live in the gray twilight that knows neither victory nor defeat."—Theodore Roosevelt

One way to learn about a language is to manually create tables that show what type results from applying a binary operator to various pairs of types. For this problem you are to write a Ruby program, `optab.rb`, that generates such tables for Java, Ruby, and Haskell.

Here's a run of `optab.rb`:

```
% ruby optab.rb ruby "*" ISA
 * | I  S  A
---+---------
 I | I  *  *
 S | S  *  *
 A | A  S  *
```

The first argument, `ruby`, specifies Ruby as the language of interest for this run.

The second argument, `"*"`, specifies the operator of interest. We'll make a practice of putting quotes around the operator because some operators, like `*` and <, are shell metacharacters. `\*` would work, too.

The third argument, `ISA`, specifies types of interest. The letters `I`, `S`, and `A` stand for `Fixnum` (`I` for integer), `String`, and `Array`, respectively.

`optab`'s output is a table showing the type that results from applying the operator to various pairs of types. The row headings on the left specify the type of the left-hand operand. The column headings along the top specify the type of the right-hand operand.

The upper-left entry, an `I`, shows that `Fixnum * Fixnum` produces a `Fixnum`. (Remember that we're using `I`, not `F`, to stand for integers.) The lower-left entry, `A`, shows that `Array * Fixnum` produces an `Array`. The `S` in the bottom of the middle row shows that `Array * String` produces a `String`.

The `*`'s indicate that `Fixnum * String`, `String * String`, and three other type combinations produce an error.

Here's an example with Java:

```
% ruby optab.rb java "*" IFDCS
 * | I  F  D  C  S
---+---------------
 I | I  F  D  I  *
 F | F  F  D  F  *
 D | D  D  D  D  *
 C | I  F  D  I  *
 S | *  *  *  *  *
```

`I`, `F`, `D`, `C`, and `S` stand for `int`, `float`, `double`, `char`, and `String`, respectively.

Here's how optab is intended to work:

> For the specified operator and types, try each pairwise combination of types with the operator by executing that expression in the specified language and seeing what type is produced, or if an error is produced. Collect the results and present them in a table.

The table just above was produced by generating and then running each of twenty-five different Java programs and analyzing their output. Here's what the first one looked like:

```
% cat checkop.java
public class checkop {
    public static void main(String args[]) {
        f(1 * 1);
        }
    private static void f(Object o) {
        System.out.println(o.getClass().getName());
        }
    }
```

Note the third line, f(1 * 1); That's an int times an int because the first operation to test is I * I.

Remember: Ruby code wrote that Java program!

In Ruby, the expression `` `some-command-line` `` is called *command expansion*. It causes the shell to execute that command line. The complete output of the command is collected, turned into a string, possibly with many newlines, and is the result of `` `...` ``. (Note that `` ` `` is a "back quote".)

Here's a demonstration of using `` `...` `` to compile and execute checkop.java:

```
>> result = `bash -c "javac checkop.java && java checkop" 2>&1`
=> "java.lang.Integer\n"
```

The extra stuff with bash -c ... 2>&1 is to cause error output, if any, to be collected too.

Here's the checkop.java that's generated for I * S:

```
public class checkop {
    public static void main(String args[]) {
        f(1 * "abc");
        }
    private static void f(Object o) {
        System.out.println(o.getClass().getName());
        }
    }
```

Note that it is identical to the checkop.java generated for I * I with one exception: the third line is different: instead of being 1 * 1 it's 1 * "abc".

Let's try compiling and running the checkop.java just above, generated for I * S:

```
% irb
>> result = `bash -c "javac checkop.java && java checkop" 2>&1`
=> "checkop.java:3: error: bad operand types for binary operator
'*'\n                    f(1 * \"abc\");\n
^\n  first type:  int\n  second type: String\n1 error\n"
```

javac detects incompatible types for * in this case and notes the error. java checkop is not executed because the shell conjunction operator, &&, requires that its left operand (the first command) succeed in order for execution to proceed with its right operand (the second command).

That output, "checkop.java:3: ..." can be analyzed to determine that there was a failure. Then, code maps that failure into a "*" entry in the table.

Let's try Haskell with the / operator. "D" is for Double.

```
% ruby optab.rb haskell "/" IDS
 / | I  D  S
---+---------
 I | *  *  *
 D | *  D  *
 S | *  *  *
```

For the first case, I / I, Ruby generated this file, checkop.hs:

```
% cat checkop.hs
(1::Integer) / (1::Integer)
:type it
```

Note that just a plain 1 was good enough for Java since the literal 1 has the type int but with Haskell we use (1::Integer) to be sure the type is Integer. (Yes; Integer, not Int.)

Let's try running it. For Java we used javac and java. We'll use ghci for Haskell and redirect from checkop.hs. Be sure to specify the **-ignore-dot-ghci** option, too!

```
% irb
>> result = `bash -c "ghci -ignore-dot-ghci < checkop.hs" 2>&1`
=> "GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for
help\n[lots more]... linking ... done.\nLoading package base ...
linking ... done.\nPrelude> \n<interactive>:2:14:\n    No instance
for (Fractional Integer)\n      arising from a use of `/'\n[lots
more]\n"
```

Ouch—an error! That's going to be a "*".

Here's the checkop.hs file generated for D * D:

```
% cat checkop.hs
(1.0::Double) * (1.0::Double)
:type it
```

Let's try it:

```
>> result = `bash -c "ghci < checkop.hs" 2>&1`
>> result = `bash -c "ghci -ignore-dot-ghci < checkop.hs" 2>&1`
=> "GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for
help\nLoading package ghc-prim ... linking ... done.\nLoading
package integer-gmp ... linking ... done.\nLoading package base ...
linking ... done.\nPrelude> 1.0\nPrelude> it :: Double\nPrelude>
Leaving GHCi.\n"
```

If we look close we see `it`, with a type: **`it :: Double`**

In pseudo-code, here's what `optab` needs to do:

```
  For each pairwise combinations of types specified on the command line...

        Generate a file in the appropriate language to test the combination at hand.

        Run the file using command expansion (`...`).

        Analyze the command expansion result, determining either the type produced or that an error was produced.

        Add an appropriate entry for the combination to the table—either a single letter for the type or an asterisk to indicate an error.
```

The examples above show Java and Haskell testing programs and their execution. You'll need to figure out how to do the same for Ruby, but let us know if you have trouble with that. The obvious route with Ruby is creating and running a file but you can use `Kernel#eval` instead. If you take the `eval` route, you'll probably need to do a bit of reading and figure out how to catch a Ruby exception using `rescue`.

I chose the names `checkop.java` and `checkop.hs` but you can use any names you want.

Below is an example of a complete program that generates a file named `hello.java` and runs it. Note that the program's command-line argument is interpolated into the "here document", which is a multi-line string literal. (See slide 79.)

```
% cat a7/mkfile.rb
prog = <<X
public class hello {
    public static void main(String args[]) {
        System.out.println("Hello, #{ARGV[0]}!");
        }
    }
X
# IMPORTANT: That X just above MUST BE IN THE FIRST COLUMN!

f = File.new("hello.java","w")
f.write(prog)
f.close
result = `bash -c "javac hello.java && java hello" 2>&1`
puts "Program output: (#{result.size} bytes)", result

% ruby mkfile.rb whm
Program output: (12 bytes)
Hello, whm!
```

Here's the file that was created:

```
% cat hello.java
public class hello {
    public static void main(String args[]) {
        System.out.println("Hello, whm!");
        }
    }
```

Copy a7/mkfile.rb into your a7 directory on lectura and try it, to help you get the idea of generating a program, running it, and then doing something with its output.

If you like to be tidy, you can use File's delete class method to delete hello.java:
File.delete("hello.java")

Here's a table that shows what types must be supported in each language, and a good expression to use for testing with that type.

Letter	Haskell	Java	Ruby
I	(1::Integer)	1	1
F	(1.0::Float)	1.0F	1.0
D	(1.0::Double)	1.0	not supported
B	True	true	true
C	'c'	'c'	not supported
S	"abc"	"abc"	"abc"
O	not supported	new Object()	not supported
A	not supported	not supported	[1]

optab.rb is not required to do any error checking at all. It assumes the first argument is haskell, java, or ruby. It assumes the second argument is a valid operator in the language specified. It assumes the third argument is a string of single-letter type specifications and that those types are supported for the language at hand. Behavior is undefined for all other cases. I won't test any error cases.

I hope that everybody recognizes that there needs to be language-specific code for running the Java, Haskell, and Ruby tests but ONE body of code can be used to process command-line arguments, launch the language-specific tests, and build the result table.

Think in terms of an object-oriented solution, perhaps with an OpTable class that handles the language-independent elements of the problem. OpTable would have subclasses JavaOpTable, HaskellOpTable, and RubyOpTable with language-specific code.

For example, my solution has a method OpTable#make_table that handles table generation. It calls a subclass method tryop(op, lhs, rhs) to try an operator with a pair of operands and report what's produced (a type or an error). With Java a call might be tryop("+", "1", '"abc"'); it would return "S". In contrast, RubyOpTable#tryop("+", "1", '"abc"') produces "*".

Note that testing the Java cases can be slow. With my version, ruby optab.rb java "*" IFDCS takes almost 30 seconds to run on lectura. The same test for Haskell takes about seven seconds.

Ruby's command expansion (`...`) works on Windows but I haven't tried to work out command lines that'll behave as well as the examples above, which were done on lectura. The bottom line is that you'll probably need to do much of your testing on lectura. However, if you use an eval-based approach for Ruby, you can easily get that working on Windows. If you want to write code that runs on both Windows and UNIX, you can use RUBY_PLATFORM as a simple way to see what sort of system you're running on.

For three points of extra credit per language, have your `optab.rb` support up to three additional languages of your choice. PHP, Python, and Perl come to mind as easy possibilities. (For Python, you can do either Python 2 or Python 3, but not both for credit.) At least three types must be supported for each language. You may introduce types in addition to those shown above. Submit a **plain text file** `optab.txt`, that shows your extended version in action. Demonstrate at least three operators for each language. The burden of proof for this extra credit is on you, not me!

Problem 6. Extra Credit `vstring-extra.txt`

For three points of extra credit, devise and implement another subclass for `VString`. For example, a fourth subclass I considered having you implement for `VString` was to be created like this:

```
XString.new("a","bb", 10)
```

It represents the following sequence of characters, which ends with 10 "a"s and 20 "b"s:

```
abbaabbaaabbb...aaaaaaaaaabbbbbbbbbbbbbbbbbbbb
```

You can't implement `XString` for credit but I'll be impressed if you do it for fun.

Add whatever new `VString` subclass you come up with to your `vstring.rb` and create `vstring-extra.txt`, a plain text file that talks about your creation and shows it action with `irb`.

Turning in your work

Use `a7/turnin` to submit your work.

To give you an idea about the size of my solutions, here's what I see as of press time, with comments stripped:

```
$ wc $(grep -v txt a7/delivs)
  21   53  573 label.rb
  29   47  599 re.rb
 104  216 1821 vstring.rb
  35   88  870 gf.rb
 148  328 3440 optab.rb
 337  732 7303 total
```

Miscellaneous

You can use any elements of Ruby that you desire, but the assignment is written with the intention that it can be completed easily using only the material in the full set of Ruby slides.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 10 to 15 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put ten hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached ten hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment 8
Due: Friday, April 15 at 23:59:59

Game plan for the Prolog assignments

Our work with Prolog will be distributed across three assignments, with the following due dates:

Assignment 8	Friday, April 15
Assignment 9	Friday, April 22
Assignment 10	**Wednesday**, May 4

Remember that the video assignment is also due on May 4.

The Usual Stuff

Make an `a8` symlink that references `/cs/www/classes/cs372/spring16/a8`. Test using `a8/tester` (or `a8/t`).

Use SWI Prolog!

We'll be using SWI Prolog for the Prolog assignments. On lectura that's `swipl`.

About the `if-then-else` structure (`->`) and disjunction (`;`)

To encourage thinking in Prolog, you are strictly prohibited from using the `if-then-else` structure, which is represented with `->`. (Section 4.7 in Covington talks about it.)

Disjunction, represented with a semicolon (`;`), is occasionally very appropriate but it's easy to misuse and make a mess. Section 1.10 in Covington talks about it. Here's the rule for us: If you think you've found a good place to use disjunction, ask me about it; but unless I grant you a specific exemption, you are not allowed to use disjunction. (My general rule is this: don't use disjunction unless it avoids significant repetition.)

Easy Money!

Due to the time frame for this assignment and not wanting to underweight problems on assignments 9 and 10, I think you'll find that the time required to do this assignment is a bit low with respect to the points assigned.

Problem 1. (7 points, ½ point each) `queries.pl`

For this problem you are to write a number of queries, packaged up as rules. Some are easier than their half-point and some are harder, but they're all worth a half-point.

`a8/queries-starter.pl` starts like this:

```
% What foods are green?
q0(Food) :- thing(Food,green,yes).

% What are all the things?
```

```
q1(Thing) :- true.

% What are the colors of non-foods?
q2(Color) :- true.
```

q0 above is a completed example. The comment just prior specifies a question, "What foods are green?" Following that comment is a query that will answer that question. Let's load up the file and try q0:

```
$ swipl a8/queries-starter.pl
[...lots of singleton warnings due to the uncompleted queries...]
...
?- q0(F).
F = broccoli ;
F = lettuce.
```

Your task is to replace the dummy bodies (just true) for all the rules. The first few use the facts in a8/things.pl; the rest use the facts in a8/fcl.pl. Begin by copying a8/queries-starter.pl to queries.pl, and then edit queries.pl.

When your queries.pl is complete you should see behavior like this:

```
$ swipl queries.pl
...

?- q1(T).
T = apple ;
T = broccoli ;
...
T = stopsign ;
T = bagel.

?- q2(NF).
NF = brown ;
NF = green ;
NF = blue ;
NF = red.
```

Leave the sample rule q0 in place—the tester uses it.

The :-QUERY. construct

You'll see that a8/queries-starter.pl ends like this:

```
:-[a8/fcl].
:-[a8/things].
```

When consulting a file, Prolog assumes that it contains clauses that constitute the knowledgebase but sometimes we want to execute queries when consulting a file. The construct **:-*QUERY*.** indicates that *QUERY* is to be performed.

The two lines above cause a8/fcl.pl and a8/things.pl to be consulted, providing the facts to be used by this problem's queries.

Grading

When grading I'll use altered versions of `a8/things.pl` and `a8/fcl.pl`, with facts added, deleted, and changed. Write queries to be general, rather than "wired" for current data. For example, if something involves the cost of an orange, use a goal like `cost(orange, OC)` to get the cost of an orange rather than visually inspecting `a8/fcl.pl`, seeing `cost(orange,3)`, and using `3` for the cost of an orange.

Important: The usual guarantee of 75% of the points for passing all supplied test cases does not apply for this problem.

A note about the tester

You'll see that the tester uses a Prolog query with several goals for the rules in `queries.pl`:

```
findall(X,q1(X),L), sort(L,Results), writeln('Results:'),
member(X,Results), writeln(X), fail.
```

We'll be learning about `findall`, `sort`, and `member` soon, but briefly, here is what's happening: `findall` makes a list of all results produced by `q1(X)` and then `sort` sorts them, removing duplicates. The `member(X,Results), writeln(X), fail` sequence causes the results to be written out, one per line.

Problem 2. (1 point) `altrules.pl`

The first examples we saw with Prolog involved `food/1` and `color/2` facts. Then on slide 48 we saw an alternate representation of the same data using `thing/3`.

For this problem you are to implement `food/1` and `color/2` as rules that use the `thing/3` facts.

Your solution should look like this:

```
:-[a8/things].

food(F) :- ...
color(T,C) :- ...
```

The first line consults `a8/things.pl`.

Your task is to simply fill in the bodies for the `food` and `color` rules.

Usage:

```
$ swipl altrules.pl
...
?- food(X).
X = apple ;
X = broccoli ;
...

?- color(apple,red).
true.

?- color(F,green).
F = broccoli ;
```

```
F = grass ;
F = lettuce.

?-
```

Problem 3. (2 points) `sequence.pl`

Write a predicate `sequence/0` that outputs the sequence below.

```
?- sequence.
10101000
10101001
10101010
10101011
10111000
10111001
10111010
10111011
true.
```

Be sure that `sequence` produces `true` when done, as shown above.

Two notes: (1) Don't over think this one. (2) **Don't just "wire-in" the output verbatim**, like `writeln(10101000), writeln(10101001), ...`—**that'll be a zero!**

Problem 4. (7 points) `rect.pl`

In this problem you are to implement several simple predicates that work with `rect(width,height)` structures that represent position-less rectangles having only a width and height.

`square(+Rect)` asks whether a rectangle is a square.

```
?- square(rect(3,4)).
false.

?- square(rect(5,5)).
true.
```

`landscape(+Rect)` is true iff (if and only if) a rectangle is wider than it is high. `portrait` tests the opposite—whether a rectangle is higher than wide. A square is neither landscape nor portrait.

```
?- landscape(rect(16,9)).
true.

?- landscape(rect(3,4)).
false.

?- portrait(rect(3,4)).
true.

?- portrait(rect(10,1)).
false.

?- landscape(rect(3,3)).
false.
```

```
?- portrait(rect(3,3)).
false.
```

`classify(+Rect,-Which)` instantiates `Which` to `portrait`, `landscape` or `square`, depending on the width and height. If `Rect` is not a two-term `rect` structure, then `Which` is instantiated to `wat`.

```
?- classify(rect(3,4),T).
T = portrait.

?- classify(rect(10,1),T).
T = landscape.

?- classify(rect(3,3),T).
T = square.

?- classify(rect(3),T).
T = wat.

?- classify(10,T).
T = wat.
```

You may need to use some cuts (slide 109+) to prevent `classify` from producing bogus alternatives. **Here is an example of BUGGY behavior:**

```
?- classify(rect(5,7),T).
T = portrait ;    First answer is correct but there should be no alternatives!
T = square ;
T = wat.
```

Needless to say, use your `portrait/1`, `landscape/1`, and `square/1` predicates to write `classify/2`.

`rotate(?R1,?R2)` has three distinct behaviors:

(1) If `R1` is instantiated and `R2` is not, `rotate` instantiates `R2` to the rotation of `R1`.
(2) If `R2` is instantiated and `R1` is not, `rotate` instantiates `R1` to the rotation of `R2`.
(3) If both are instantiated, `rotate` succeeds iff `R1` is the rotation of `R2`.

Examples:

```
?- rotate(rect(3,4),R).
R = rect(4, 3).

?- rotate(R,rect(3,4)).
R = rect(4, 3).

?- rotate(rect(5,7),rect(7,5)).
true.

?- rotate(rect(3,3),R).
R = rect(3, 3).
```

`rotate` should also handle cases like these:

```
?- rotate(rect(3,4),rect(W,H)).
```

```
W = 4,
H = 3.

?- rotate(rect(3,X),rect(Y,4)).
false.
```

`smaller(+R1,+R2)` succeeds iff both the width and height of `R1` are respectively less than the width and height of `R2`. Rotations are not considered.

```
?- smaller(rect(3,5), rect(5,7)).
true.

?- smaller(rect(3,5), rect(7,5)).
false.
```

`add(+R1, +R2, ?RSum)` follows the idea of "adding" rectangles that was shown on the Ruby slides on operator overloading.

```
?- add(rect(3,4),rect(5,6),R).
R = rect(8, 10).

?- add(rect(3,4),rect(5,6),rect(W,H)).
W = 8,
H = 10.

?- add(rect(3,4),rect(5,6),rect(10,10)).
false.

?- X = 10, add(rect(3,4),rect(5,6),rect(X,X)).
false.
```

Assume both terms of `rect` structures are non-negative integers.

If you need more than ten mostly short lines of Prolog to implement all the above, you're probably not making good use of unification.

Problem 5. (3 points) `consec.pl`

Write a predicate `consec(?A, ?B, ?C)` that expresses the relationship that `A`, `B`, and `C` are consecutive integers. `A`, `B`, and `C` can be any combination of integers and uninstantiated variables.

Examples:

```
?- consec(6,B,C).
B = 7,
C = 8.

?- consec(3,4,5).
true.

?- consec(X,Y,-3).
X = -5,
Y = -4.

?- consec(X,0,Y).
```

```
X = -1,
Y = 1.

?- consec(A,2,A).
false.
```

If none of the three terms are instantiated, we see this:

```
?- consec(A,B,C).
A = 1,
B = 2,
C = 3.
```

Implementation notes

`integer/1` can be used to see if a term is an integer or uninstantiated:

```
?- integer(5).
true.

?- integer(A).
false.

?- A = 5, integer(A).
A = 5.
```

Also, note this behavior of `is/2`:

```
?- X = 2, 3 is X + 1.
X = 2.

?- X = 2, 3 is X + 10.
false.
```

My solution has four clauses.

Problem 6. (3 points) `bases.pl`

Write a predicate `bases/2` such that `bases(+Start,+End)` prints the integers from `Start` through `End` in decimal, hex, and binary. Assume that `Start` is non-negative and that `End` is greater than `Start`. Examples:

```
$ swipl bases.pl
...

?- bases(0,5).
 Decimal     Hex        Binary
     0         0             0
     1         1             1
     2         2            10
     3         3            11
     4         4           100
     5         5           101
true.

?- bases(1022,1027).
```

```
 Decimal     Hex         Binary
  1022       3FE    1111111110
  1023       3FF    1111111111
  1024       400   10000000000
  1025       401   10000000001
  1026       402   10000000010
  1027       403   10000000011
true.
```

Be sure that your predicate succeeds, showing `true`, not `false`.

Below is a predicate `fmttest/0` that shows almost exactly the specifications to use with `format/2`. However, you'll need to do `help(format/2)` and figure out how to output numbers in hex and binary.

```
?- listing(fmttest).
fmttest :-
        format('~tDecimal~t~10|~tHex~t~20|~tBinary~t~35|\n'),
        format('~t~d~6|~t~d~16|~t~d~30|\n', [10, 20, 30]).

true.

?- fmttest.
 Decimal     Hex         Binary
    10        20            30
true.
```

Problem 7. (12 points) `grid.pl`

Write a predicate `grid(+Rows,+Cols)` that prints an ASCII representation of a grid based on a specification of rows and columns in English.

Here's an example of a grid with three rows and four columns:

```
?- grid(three,four).
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
|  |  |  |  |
+--+--+--+--+
true.
```

The grid is built with plus signs, minus signs, vertical-bars ("or" bars), and spaces. Lines have no trailing whitespace.

Unless a specification is invalid, `grid` always succeeds, producing the `true` that follows the output.

Here are two more examples:

```
?- grid(three,twenty-one).
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
true.

?- grid(one,one).
+--+
|  |
+--+
true.
```

Widths and heights, in English, from `one` through `ninety-nine` are recognized; numbers are one word or two hyphen-separated words.

If a number is used for either dimension instead of an English specification, the user is reminded to use English:

```
?- grid(3,four).
Use English, please!
true.
```

Hint: Use `number/1` to see if a value is a number rather than a structure.

Invalid specifications produce `Huh?`:

```
?- grid(testing,this).
Huh?
true.

?- grid(one-hundred,twenty-five).      one-hundred is out of range
Huh?
true.

?- grid(---,+++).
Huh?
true.
```

Be careful not to accept invalid combinations of words representing numbers, like `ten-four`, `twenty-twenty`, and `one-fifty`; they, too, should produce the `Huh?` diagnostic. Example:

```
?- grid(ten-four,twenty-twenty).
Huh?
true.
```

`a8/grid-hint.html` shows a solution for a simplified version of this problem, a predicate `box` that prints a rectangle of asterisks. To provide a little extra challenge for those who want it, I'm not showing that code here but please don't hesitate to take a look if you're stumped by `grid`.

Note that terms like `ninety-nine`, `thirty-seven`, `fifty-two` are simply two-atom structures with the functor `'-'`. Here's a predicate that prints the terms of such a structure:

```
parts(First-Second) :-
     format('First word: ~w; second word: ~w\n', [First,Second]).

?- parts(twenty-one).
First word: twenty; second word: one
```

```
true.
```

a8/numbers.txt might save you a little typing. (Think about using a keyboard macro/keystroke recorder in your editor or maybe a Ruby program to turn the text in that file into Prolog facts.)

Problem 8. (4 points) rsg.pl

"rsg" stands for "random sentence generator".

Overall, this problem has three parts:

(1) Decide what sort of "sentences" you'd like to generate.
(2) Create a predicate rsg that outputs a single random sentence.
(3) Create a predictate rsg(+N) that calls rsg/0 N times.

Here's an example of an rsg that generates trivial English sentences:

```
?- rsg.
The boy sat.
true.

?- rsg.
A girl ran.
true.

?- rsg.
The girl spoke.
true.
```

Here's the Prolog code for the rsg above:

```
rsg :- article, b, noun, b, verb, write(.), !.

article :- p(0.5), w('The').
article :- p(_),   w('A').

noun :- p(0.33), w('boy').
noun :- p(0.5), w('girl').
noun :- p(_), w('dog').

verb :- p(0.4), w('ran').
verb :- p(0.66), w('sat').
verb :- p(_), w('spoke').

w(X) :- write(X).

b :- w(' ').

p(P) :- number(P), !, random(100) < P * 100.
p(_).
```

w/1 and b/0 are convenience predicates that let us save a little typing.

p(X) is a predicate that succeeds with a probability equal to X. For example, p(0.5) succeeds half the time, on average. If p is not called with a number, it succeeds.

Let's consider the procedure (the clauses) for `article`, which output `The` or `A` with equal probability:

```
article :- p(0.5), w('The').
article :- p(_), w('A').
```

The first clause succeeds half the time. If the first clause fails, which it will half the time, the second clause is tried, and it always succeeds. Effectively, its probability is also 0.5 but it's important that one always succeeds, so we'll use the convention of using `p(_)` on the last clause to stand for "otherwise". (Yes, we could just omit it, too, but having a `p` goal on each clause seems more aesthetically appealing at the moment.)

And now, a confession: I made a dopey mistake when writing this problem. Here was my first version of `noun`:

```
noun :- p(0.33), w('boy').
noun :- p(0.33), w('girl').
noun :- p(_), w('dog').
```

My thinking was that I'd get an even three-way distribution between `boy`, `girl`, and `dog` but for 10,000 calls to `noun` I found that I was getting counts like these:

```
3297 boy
4524 dog
2179 girl
```

What's happening is that a third of the time, the first clause succeeds and we get `boy`. In the two-thirds of the time that the first clause fails, we then pick `girl` one third of the time, which is 2/9 overall. We reach the always succeed `p(_)` case 4/9 of the time overall and produce `dog`. Oops!

My press deadline was looming and I couldn't think of a simple way to produce an even distribution with the Prolog we've seen. (In particular, without using lists and/or some higher-order predicates.) I thought about dropping this problem altogether but I like it because it gives you a chance to be creative. Thus it remains, along with this story.

To get an even three-way split, we can do this:

```
noun :- p(0.33), w('boy').
noun :- p(0.5), w('girl').
noun :- p(_), w('dog').
```

One third of the time we get `boy`. In half of the remaining two-thirds, we get `girl`. In the remaining third overall, we get `dog`.

If you do the math for the `verb` procedure shown above, I believe you'll find that we should get `ran` 40% of the time, `sat` 40% of the time, and `spoke` 20% of the time.

If you want to work through the math or maybe write a Ruby method to generate `p` values that produce a particular distribution, that's fine, but if you want to just plop in some numbers and see if they produce results you like, that's fine, too!

(End of confession and emergency problem salvage operation.)

Let's use a flexible definition for "sentence" and now say that a sentence is an array of integers and nested arrays of integers. Here are some random "sentences" of that sort:

```
?- rsg.
[[17,91,30,31,38,40],85,48,96,[79,62]]
true.

?- rsg.
[61]
true.

?- rsg.
[[3,58,[1,[21,95,6,85,9,92,38,79,27,24,2,10,47,[6,[96,58,62],57,56]
],44],[83,48,14,60,79,[29,6,49,93,55,24]],2,96,23,64,69,97,[[37,32,
66,12],41,9],36,[[33,76],45],74,37,[5],72,55,86,[16,9,30],41],[[[92
,3],90,39],64,18,22,19],47,65,57,49]
true.
```

A fourth result, that's not shown above, was 23,424 characters long.

Here's the code that generated the arrays above:

```
rsg :- array, !.

array :- w('['), elems, w(']').

elems :- p(0.2), elem.
elems :- p(_), elem, w(','), elems.

elem :- p(0.8), X is random(100), w(X).
elem :- p(_), array.
```

Note the procedure for `elems`. 20% of the time it outputs a single element. 80% of the time it outputs an element (`elem`) followed by a comma and more elements (`elems`—a recursive call).

The procedure for `elem` outputs a random number between 0 and 99 on 80% of the calls, but on the remaining 20%, it outputs an array.

Note that the body for `rsg` should end with a cut, to prevent backtracking that would in turn produce some malformed results.

Possibilities for "sentence"

If you want to stick to English sentences, you might have some fun with a Mad Libs style involving popular culture or current events, especially the election season.

There are lots of possibilities in the symbolic realm, like expressions for some language you know, or even simple whole programs. Various possibilities with ASCII pictures come to mind. You might consider an HTML document to be a sentence. It's fine to have multi-line "sentences".

I hope I'll be dazzled by some creativity but something as simple as English sentences with three or more fields with varying content is sufficient for full credit.

I'll compile a sampling of random sentences generated by all solutions and post it on Piazza. I won't show authorship.

The approach shown above lets you be fairly creative using only material presented on slides 1-131 but you are free to implement `rsg/0` in any way you want. In particular, when we get into lists you'll see additional ways to randomly choose from several things.

Incidentally, another of the many things I learned from Ralph Griswold is that processing machine-generated input, like the random arrays above, often turns up bugs that have long been dormant while processing human-generated input.

Along with `rsg/0` you'll need to write `rsg(+N)`, which generates `N` random sentences using `rsg/0`. `N` is assumed to be an integer greater than zero. A line with three dashes is output after each sentence.

```
?- rsg(3).
[69,72]
---
[55,11,[18,83,80,57,1,54,13,57,[59],68,25],1]
---
[4,[[47,41,98,96]]]
---
true.
```

Testing note

Because of the nature of this problem there's no simple way to test `rsg/0` in an automated fashion. For this problem the tester only confirms that `rsg/1` produces the right number of "`---`" lines. Passing that simple test doesn't guarantee 75% of the points on this problem.

Problem 9. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a8/turnin` to submit your work.

Line counts are often good for ballpark measurements of program size for many languages but they're sometimes misleading with Prolog. For example, I'll sometimes write procedures with one goal per line. With Prolog I'm going to give you a different measure of my solutions: the number of left parentheses and commas that appear. I'll use this bash script:

```
$ cat a8/plsize
for i in $*
do
    echo $i: $(tr -dc "(," < $i | wc -c)
done
```

Here's what I see as of press time, with comments stripped:

```
$ a8/plsize $(grep -v txt a8/delivs)
queries.pl: 129
altrules.pl: 9
sequence.pl: 17
rect.pl: 46
consec.pl: 24
bases.pl: 13
grid.pl: 131
rsg.pl: 39
```

Miscellaneous

Aside from `->` and `;` you can use any elements of Prolog that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Prolog slides 1-131.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) In Prolog, `%` is comment to end of line. Comments with `/* ... */`, just like in Java, are supported, too.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 4 to 6 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put six hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached six hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment 9
Due: Friday, April 22 at 23:59:59

The Usual Stuff

Make an `a9` symlink that references `/cs/www/classes/cs372/spring16/a9`. Test using `a9/tester` (or `a9/t`). Use SWI Prolog—`swipl` on lectura.

About the `if-then-else` structure (`->`) and disjunction (`;`)

The rules about using `if-then-else` and disjunction are the same as for assignment 8.

`once/1`

Various tests run by the tester use the higher-order predicate `once(Goal)`, which limits `Goal` to producing one result. Example:

```
?- once(nth0(Pos, [a,b,a,c,a], a)).
Pos = 0.
```

Potential alternatives that don't materialize

The tester is not sensitive to potential alternatives that don't materialize. Consider this behavior for `last`, from the first problem below.

```
?- last([1,2,3],X).
X = 3 ;
false.
```

It prompts because somewhere there's at least one more clause that can be tried. The tester won't distinguish between the above behavior and the following behavior:

```
?- last([1,2,3],X).
X = 3.
```

Problem 1. (5 points) `append.pl`

Note: This problem has restrictions!

The purpose of this problem is to help you see that `append/3` can be used for lots more than concatenating two lists. You are to write several simple predicates that heed this **restriction: the only predicates you can use are `length/2` and `append/3`**. For some predicates, like `head`, a single `append` goal will be all you need:

```
head(List,Elem) :- append(...).
```

Additionally, you **may** use the `[E1, E2, ..., EN]` list syntax, like in `append([X,Y],[],Z)`, but you **may not use** the `[E1, E2, ..., EN | Tail]` notation, introduced on slide 192.

Here are the predicates you are to implement:

`head(?List, ?Elem)` expresses the relationship that the first element of `List` is `Elem`. It fails if the list is empty.

```
?- head([a,b,c],H).
H = a.

?- head([a,b,c],x).
false.
```

`last(?List, ?Elem)` expresses the relationship that the last element of `List` is `Elem`. It fails if the list is empty.

```
?- last([1,2,3],X).
X = 3 ;
false.

?- last(L,4).
L = [4] ;
L = [_G2199, 4] ;
L = [_G2199, _G2205, 4] ;
...keeps going...
```

`init(?List, ?Init)` expresses the relationship that `Init` is all but the last element of `List`. It fails if `List` is empty.

```
?- init([a,b,c,d],I).
I = [a, b, c] ;
false.
```

`tail(?List, ?Tail)` expresses the relationship that `Tail` is all but the first element of `List`. It fails if `List` is empty.

```
?- tail([a,b,c],T).
T = [b, c].
```

`min2(+List)` fails if `List` is not at least two elements long.

```
?- min2([1]).
false.

?- min2([1,2]).
true.
```

`mem(?Elem, ?List)` behaves just like the built-in `member/2`:

```
?- mem(2,[1,2,3]).
true ;
false.

?- mem(E,[1,2,3]).
E = 1 ;
E = 2 ;
E = 3 ;
false.
```

`contains(+List,?SubList)` expresses the relationship that `List` contains `SubList`.

```
?- contains([1,2,3,4,5],[3,4,5]).
true ;
false.

?- contains([1,2,3,4,5],[10,20]).
false.

?- contains([1,2,3],S), S \== [].
S = [1] ;
S = [1, 2] ;
S = [1, 2, 3] ;
S = [2] ;
S = [2, 3] ;
S = [3] ;
false.
```

`firstlast(?List,?FL)` expresses the relationship that `FL` is a list containing the first and last elements of `List`. It fails if `List` is empty.

```
?- firstlast(L,[1,2]).
L = [1, 2] ;
L = [1, _G3410, 2] ;
L = [1, _G3410, _G3416, 2] ;
...keeps going...
```

`halves(?List, ?First, ?Last)` expresses the relationship that the first half of `List` is `First` and the second half is `Last`. It fails if the length of `List` is odd.

```
?- halves([1,2,3,4],F,S).
F = [1, 2],
S = [3, 4] ;
false.

?- halves([1,2,3,4,5],F,S).
false.

?- halves([],F,S).
F = S, S = [] ;
false.

?- halves(L,[a,b],S).
L = [a, b, _G4635, _G4638],
S = [_G4635, _G4638].
```

Remember the restriction: you may only use `append` and `length` in these predicates!

Testing note

Use `a9/t append.pl -t` *`PREDICATE`* to test an individual predicate. As you've seen before, the `-t` follows the problem name.

Problem 2. (2 points) middle.pl

Write a predicate `middle(+List,+N,?Middle)` that expresses the relationship that the middle N elements of `List` are `Middle`.

Restriction: Just like problem 1, you may only use `append` and `length` in this predicate.

```
?- middle([a,b,c,d,e],3,M).
M = [b, c, d] ;
false.

?- middle([a,b,c,d,e],5,M).
M = [a, b, c, d, e] .

?- middle([a,b,c,d,e],2,M).
false.

?- middle([a,b,c,d,e,f],2,M).
M = [c, d] ;
false.

?- middle([a,b,c],1,M).
M = [b] ;
false.

?- middle([a,b,c],2,M).
false.
```

As the above examples imply, an even-length list has an even-length middle, and an odd-length list has an odd-length middle.

Assume that `N` (the length of the middle) is greater than zero.

Don't forget the restriction!

Problem 3. (2 points) splits.pl

This problem reprises `splits.hs` from assignment 3. In Prolog it is to be a predicate `splits(+List,-Split)` that unifies `Split` with each "split" of `List` in turn. Example:

```
?- splits([1,2,3],S).
S = [1]/[2, 3] ;
S = [1, 2]/[3] ;
false.
```

Note that `Split` is not an atom. It is a structure with the functor `/`. Observe:

```
?- splits([1,2,3], A/B).
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
false.
```

Here are additional examples. Note that splitting a list with less than two elements fails.

```
?- splits([],S).
false.

?- splits([1],S).
false.

?- splits([1,2],S).
S = [1]/[2] ;
false.

?- atom_chars('splits',Chars), splits(Chars,S).
Chars = [s, p, l, i, t, s],
S = [s]/[p, l, i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p]/[l, i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l]/[i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l, i]/[t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l, i, t]/[s] ;
false.
```

My solution uses only two predicates: `append` and `\==`.

Problem 4. (3 points) `posints.pl`

Write a predicate `posints(+List)` that succeeds iff all elements in `List` are positive (>`0`) integers.

Restriction: Your solution must not be recursive!

```
?- posints([1,2,3,4,5]).
true.

?- posints([1,2,3,-4,5]).
false.

?- posints([1,xyz,3,4,5]).
false.

?- posints([]).
true.
```

I hope you'll ponder this one for a little while but here's a hint:
http://www.cs.arizona.edu/classes/cs372/spring16/a9/posints-hint.html

Problem 5. (3 points) `repl.pl`

Write a predicate `repl(?E, +N, ?R)` that unifies `R` with a list that is `N` replications of `E`. If `N` is less than `0`, `repl` fails.

```
?- repl(x,5,L).
L = [x, x, x, x, x].
```

```
?- repl(1,3,[1,1,1]).
true.

?- repl(X,2,L), X=7.
X = 7,
L = [7, 7].

?- repl(a,0,X).
X = [].

?- repl(a,-1,X).
false.
```

Problem 6. (3 points) `pick.pl`

Write a predicate `pick(+From, +Positions, -Picked)` that unifies `Picked` with an atom consisting of the characters in `From` at the zero-based, non-negative positions in `Positions`.

```
?- pick('testing', [0,6], S).
S = tg.

?- pick('testing', [1,1,1], S).
S = eee.

?- pick('testing', [10,2,4], S).
S = si.

?- between(0,6,P), P2 is P+1, pick('testing', [P,P2], S),
writeln(S), fail.
te
es
st
ti
in
ng
g
false.

?- pick('testing', [], S).
S = ''.
```

If a position is out of bounds, it is silently ignored. My solution uses `atom_chars`, `findall`, `member`, and `nth0`.

Problem 7. (5 points) `polyperim.pl`

Write a predicate `polyperim(+Vertices,-Perim)` that unifies `Perim` with the perimeter of the polygon described by the sequence of Cartesian points in `Vertices`, a list of `pt` structures.

```
?- polyperim([pt(0,0),pt(3,4),pt(0,4),pt(0,0)],Perim).
Perim = 12.0.

?- polyperim([pt(0,0),pt(0,1),pt(1,1),pt(1,0),pt(0,0)],Perim).
Perim = 4.0.
```

```
?- polyperim([pt(0,0),pt(1,1),pt(0,1),pt(1,0),pt(0,0)],Perim).
Perim = 4.82842712474619.
```

There is no upper bound on the number of points but at least four points are required, so that the minimal path describes a triangle. (Think of it as ABCA, with the final A "closing" the path.) If less than four points are specified, `polyperim` fails with a message:

```
?- polyperim([pt(0,0),pt(3,4),pt(0,4)],Perim).
At least a four-point path is required.
false.
```

The last point must be the same as the first. If not, `polyperim` fails with a message:

```
?- polyperim([pt(0,0),pt(3,4),pt(0,4),pt(0,1)],Perim).
Path is not closed.
false.
```

Note: check first for the minimum number of points and then a closed path.

This is not a course on geometric algorithms so keep things simple! Calculate the perimeter by simply summing the lengths of all the sides; don't worry about intersecting sides, coincident vertices, etc.

Be sure that `polyperim` produces only one result.

Problem 8. (14 points) `switched.pl`

This problem is a reprise of `switched.rb` from assignment 6. `a9/births.pl` has a subset of the baby name data, represented as facts. Here are the first five lines:

```
% head -5 a9/births.pl
births(1950,'Linda',f,80437).
births(1950,'Mary',f,65461).
births(1950,'Patricia',f,47942).
births(1950,'Barbara',f,41560).
births(1950,'Susan',f,38024).
```

`births.pl` holds data for only 1950-1959. Names with less than 70 births are not included.

Your task is to write a predicate `switched(+First,+Last)` that prints a table much like that produced by the Ruby version. To save a little typing, `switched` assumes that the years specified are in the 20th century.

```
?- switched(51,58).
            1951  1952  1953  1954  1955  1956  1957  1958
Dana        1.19  1.20  1.26  1.29  1.00  0.79  0.67  0.64
Jackie      1.40  1.29  1.14  1.13  1.11  0.94  0.72  0.57
Kelly       4.23  2.74  3.73  2.10  2.32  1.77  0.98  0.51
Kim         2.58  1.82  1.47  1.08  0.61  0.30  0.17  0.12
Rene        1.43  1.32  1.15  1.24  1.13  0.88  0.87  0.89
Stacy       1.06  0.81  0.62  0.47  0.44  0.36  0.29  0.21
Tracy       1.51  1.14  1.02  0.73  0.56  0.55  0.59  0.59
true.
```

If no names are found, `switched` isn't very smart; it goes ahead and prints the header row:

```
?- switched(52,53).
           1952  1953
true.
```

If you want to make your `switched` smarter, that's fine—I won't test with any spans that produce no names. Also, I'll only test with spans where the first year is less than the last year.

Names are left-justified in a ten-wide field. Below is a `format` call that does that. Note that the dollar sign is included only to clearly mark the end of the output.

```
?- format("~w~t~10|$", 'David').
David     $
true.
```

Outputting the ratios is a little more complicated. I use `sformat`, like this:

```
?- sformat(Out, '~t~2f~6|', 2.32754), write(Out).
  2.33
Out = "  2.33".
```

The call above instantiates `Out` to a six-character **string** (a list of integers that are character codes) and `write(Out)` outputs it.

See `help(format/2)` if you're curious about the details of using `~t` but the essence is that you can use `~N|` to specify that a field extend to column `N`, and then put a `~t` on the left, right, or both sides of a specifier like `~w` or `~f` to get right, left, or center justification, respectively.

To consult `a9/births.pl` when you consult `switched.pl`, put the following line in your `switched.pl`.

```
:-[a9/births].
```

As mentioned in the assignment 8 write-up for `queries.pl`, that construction, `:-` followed by a query, causes the query to be executed when the file is consulted.

Note that `:-[a9/births.pl].` fails with an error. For an extra point on this assignment, add a note to `observations.txt` with a speculative but sound explanation of why `[a9/births]` works but `[a9/births.pl]` doesn't. No Googling, etc., please!

You might also use that `:- ...` mechanism to cause a couple of tests to be run when the file is loaded. Below I define `test/0` to do a couple of `switched` queries, putting a line of dashes between them. I then invoke it with `:-test.`

```
test :- switched(51,58), writeln('-----'), switched(51,52).

:-test.
```

That invocation of `test` must follow the definition of `switched` and consulting `a9/births.pl`.

You'll see that with `swipl switched.pl` the output from `test` appears before "`Welcome to SWI-Prolog...`"

Be sure to comment out lines like `:-test.` before turning in your solution. (That output will cause a9/tester failures, too, as you'd expect.)

My Prolog solution is significantly smaller than my Ruby solution but it's easy to get sideways on this problem if you don't come up with a good set of helper predicates. I suggest that you give it a try on your own but if it starts to get ugly, http://www.cs.arizona.edu/classes/cs372/spring16/a9/switched-hints.pdf shows how I broke it down. The points assigned to this problem are based on the assumption that you will take a look at the hints. Without the hints, and based on your current level of Prolog knowledge, I might assign this problem 25-30 points.

Problem 9. (14 points) iz.pl

In this problem you are to write a predicate `iz/2` that evaluates expressions involving atoms and a set of operators and functions. Let's start with some examples:

```
?- S iz abc+xyz.        % + concatenates two atoms.
S = abcxyz.

?- S iz (ab + cd)*2.  % *N  produces N replications of the atom.
S = abcdabcd.

?- S iz -cat*3.         % – is a unary operator that produces a reversed copy of the atom.
S = tactactac.

?- S iz -cat+dog.
S = tacdog.

?- S iz abcde / 2.      % / N  produces the first N characters of the atom.
S = ab.

?- S iz abcde / -3.     % If N is negative, / produces last N characters
S = cde.

?- N is 3-5, S iz (-testing)/N.
N = -2,
S = et.
```

The functions `len` and `wrap` are also supported. `len(E)` evaluates to an **atom** (not a number!) that represents the length of `E`.

```
?- N iz len(abc*15).
N = '45'.

?- N iz len(len(abc*15)).
N = '2'.
```

`wrap` adds characters to both ends of its argument. If `wrap` is called with two arguments, the second argument is concatenated on both ends of the string:

```
?- S iz wrap(abc, ==).
S = '==abc=='.

?- S iz wrap(wrap(abc, ==), '*' * 3).
S = '***==abc==***'.
```

If `wrap` is called with three arguments, the second and third arguments are concatenated to the left and right ends of the string, respectively:

```
?- S iz wrap(abc, '(', ')').
S = '(abc)'.

?- S iz wrap(abc, '>' * 2, '<' * 3).
S = '>>abc<<<'.
```

It is important to understand that `len(xy)`, `wrap(abc, ==)`, and `wrap(abc, '(', ')')` are simply structures. If `iz` encounters a two-term structure whose functor is `wrap` (like `wrap(abc, ==)`) its value is the concatenation of the second term, the first term, and the second term. i**z** evaluates `len` and `wrap` like i**s** evaluates `random` and `sqrt`.

The atoms `comma`, `dollar`, `dot`, and `space` do not evaluate to themselves with `iz` but instead evaluate to the atoms `','`, `'$'`, `'.'`, and `' '`, respectively. (They are similar to `e` and `pi` in arithmetic expressions evaluated with `is/2`.) In the following examples note that `swipl` (not me!) is adding some additional wrapping on the comma and the dollar sign that stand alone. That adornment disappears when those characters are used in combination with others.

```
?- S iz comma.
S = (',').

?- S iz dollar.
S = ($).

?- S iz dot.
S = '.'.

?- S iz space.
S = ' '.

?- S iz comma+dollar*2+space+dot*3.
S = ',$$ ...'.

?- S iz wrap(wrap(space+comma+space,dot),dollar).
S = '$. , .$'.

?- S iz dollarcommadotspace.
S = dollarcommadotspace.
```

The final example above demonstrates that these four special atoms don't have special meaning if they appear in a larger atom.

Here is a summary for `iz/2`:

> `-Atom iz +Expr` unifies `Atom` with the result of evaluating `Expr`, a structure representing a calculation involving atoms. The operators (functors) are as follows:
>
> `E1+E2` Concatenates the atoms produced by evaluating `E1` and `E2` with `iz`.
>
> `E*N` Concatenates `E` (evaluated with `iz`) with itself `N` times. (Just like Ruby.) `N` is a term that can be evaluated with `is/2` (repeat, i**s**/2). Assume `N >= 0`.

E/N Produces the first (last) N characters of E if N is greater than (less than) 0. If N is zero, an empty atom is produced. (An empty atom is shown as two single quotes with nothing between them.) N is a term that can be evaluated with is/2. The behavior is undefined if abs(N) is greater than the length of E.

-E Produces reversed E.

len(E) Produces an **atom**, not a number, that represents the length of E.

wrap(E1,E2) Produces E2+E1+E2.

wrap(E1,E2,E3) Produces E2+E1+E3.

The behavior of iz is undefined for all cases not covered by the above. Things like 1+2, abc*xyz, etc., simply won't be tested.

Here are some cases that demonstrate that the right-hand operand of * and / can be an arithmetic expression:

```
?- X = 2, Y= 3, S iz 'ab' * (X+Y*3).
X = 2,
Y = 3,
S = abababababababababab .

?- S = '0123456789', R iz S + -S, End iz R / -(2+3).
S = '0123456789',
R = '01234567899876543210',
End = '43210' .
```

Implementation notes

One of the goals of this problem is to reinforce the idea that for an expression like -a+b*3 Prolog creates a tree structure that reflects the precedence and associativity of the operators. is/2 evaluates a tree as an arithmetic expression. iz/2 evaluates a tree as a "string expression". Note the contrast when the same tree is evaluated by is and iz:

```
?- X is pi + e*3.          % using is
X = 11.296438138966929.

?- X iz pi + e*3.          % using iz
X = pieee .
```

It's important to understand Prolog itself parses the expression and builds a corresponding structure that takes operator precedence into account. display/1 shows the tree:

```
?- display(pi + e*3).
+(pi,*(e,3))
true.
```

Processing of syntactically invalid expressions like abc + + xyz never proceeds as far as a call to iz.

Below is some code to get you started. It fully implements the + operation.

```
% cat a9/iz0.pl
:-op(700, xfx, iz).   % Declares iz to be an infix operator. (Remember that leading :-
                      % causes the code to be evaluated as a goal, not consulted as a
                      % clause.)

A iz A :- atom(A), !.
R iz E1+E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).
```

Here are examples that use the version of `iz` just above.

```
?- [a9/iz0].          % Note: no ".pl"
true.

?- X iz abc+def.
X = abcdef.

?- X iz abc+def, Y iz X+'...'+X.
X = abcdef,
Y = 'abcdef...abcdef'.

?- X iz a+b+(c+(de+fg)+hij+k)+l.
X = abcdefghijkl.
```

Let's look at the code provided above. Let's first talk about the second clause:

```
R iz E1 + E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).
```

The first thing you may notice is that the head (`R iz E1 + E2`) doesn't match the *`functor(term,term,...)`* form we've always seen. That's because the `op(700, xfx, iz)` call above lets us use `iz` as an infix operator, and that applies in both the head and body of a rule. Here is a **completely equivalent version** that doesn't take advantage of the `op` specification:

```
iz(R,E1+E2) :- iz(R1,E1), iz(R2,E2), atom_concat(R1, R2, R).
```

With that equivalence explained, lets focus on the version in `a9/iz0.pl`, which uses the infix form:

```
R iz E1 + E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).
```

Consider the goal "`X iz ab+cd`". That goal unifies with the head of the above rule like this:

```
?- (X iz ab+cd) = (R iz E1+E2).
X = R,
E1 = ab,
E2 = cd.
```

If `E1` is instantiated to `ab` then the first goal in the body of the rule would be equivalent to `R1 iz ab`. That goal unifies with the head of this rule:

```
A iz A :- atom(A), !.
```

This rule represents the base case for the recursive evaluation performed by `iz`. It says, "If `A` is an atom then the result of evaluating that atom is `A`." Another way to read it is, "An atom evaluates to itself." The result is that "`R1 iz ab`" instantiates `R1` to `ab`. Note that atoms always lie at the leaves of the expression's tree.

It's important to recognize that because the `iz(R, E1+E2)` rule is recursive, it'll handle every tree composed of + operations.

Here are the heads (and necks) for all the `iz` rules that I've got in my solution:

```
R iz E1+E2 :-

R iz E1*NumExpr :-

R iz -E :-

R iz E1 / NumExpr :-

R iz len(E) :-

R iz wrap(E,Wrap) :-

R iz wrap(E,First,Last) :-
```

Via recursion, those heads handle all possible combinations of operations, like this one:

```
?- X iz wrap((-(ab+cde*4)/6+xyz), 'Start>','<'+(end*3+zz*2)).
X = 'Start>edcedcxyz<endendendzzzz'.
```

If you find yourself wanting to add a bunch of rules like

```
R iz (E1+E2+E2) :- ...

R iz (E1*E2) / NumExpr :- ...
```

then STOP! You're not recognizing that a set of rules based on the heads above will cover ALL the operations.

atomic_list_concat

`iz0.pl` above uses `concat_atom` but `atomic_list_concat(+List, -Atom)` is a more general predicate:

```
?- atomic_list_concat([ab,c,def,ghij], S).
S = abcdefghij.
```

We'll later see in the slides that it can be used to split atoms, too.

A minor parsing problem

On the slides I fail to mention that Prolog requires some sort of separation between operators. Consider this:

```
?- X iz abc+-abc.     % No space between + and -
ERROR: Syntax error: Operator expected
ERROR: X iz abc
ERROR: ** here **
```

```
ERROR: +-abc .
```

To make it work, add a space or parenthesize:

```
?- X iz abc+ -abc.
X = abccba.

?- X iz abc+(-abc).
X = abccba.
```

This issue only arises with unary operators, of course.

Problem 10. Extra Credit `iz-extra.txt`

For up to three points of extra credit, devise and implement three more operations for `iz` to handle. They can be operators or functions (like `len` and `wrap`). Perhaps use `op/3` to define a new operator!

Include the required clauses in `iz.pl` and create `iz-extra.txt`, a plain text file that talks about your creation(s) and also shows them in action with `swipl`.

Problem 11. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with `"Hours:"`. There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a9/turnin` to submit your work.

Just like on assignment 8 I'll use my `plsize` script to show you the "sizes" of my solutions. (Recall that the script counts left parentheses and commas, which I claim is a reasonable proxy for program size for Prolog source code.)

Here's what I see as of press time, with comments stripped:

```
$ a9/plsize $(grep -v txt a9/delivs)
append.pl: 65
middle.pl: 19
splits.pl: 7
repl.pl: 15
posints.pl: 9
pick.pl: 19
polyperim.pl: 53
switched.pl: 106
iz.pl: 76
```

Miscellaneous

Aside from `->` and `;`, and any per-problem restrictions notwithstanding, you can use any elements of Prolog that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Prolog slides 1-185.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) In Prolog, `%` is comment to end of line. Comments with `/* ... */`, just like in Java, are supported, too.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 5 to 7 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put seven hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached seven hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment 10
Due: **Wednesday**, May 4 at 23:59:59

The Usual Stuff

Make an `a10` symlink that references `/cs/www/classes/cs372/spring16/a10`. Test using `a10/tester` (or `a10/t`). Use SWI Prolog—`swipl` on lectura. The rules about using `if-then-else` and disjunction (`;`) are the same as for assignments 8 and 9.

General advice

If you think you need to use arithmetic or something like `between` on `rotate`, `outin`, and/or `btw`, problems you're probably not understanding how Prolog naturally generates alternatives. The slides have code for a number of predicates that generate alternatives but `sf_gen` on 201-202 was included specifically to help with `outin` and `btw`.

`pipes.pl` is the only problem on which using `assert` and `retract` is appropriate.

Problem 1. (2 points) `rotate.pl`

Write a Prolog predicate `rotate(+L,?R)` that instantiates `R` to each unique list that is a left rotation of `L`. For example, the list `[1,2,3]` can be rotated left to produce `[2,3,1]` which in turn can be rotated left again to produce `[3,1,2]`.

```
?- rotate([1,2,3],L), writeln(L), fail.
[1,2,3]
[2,3,1]
[3,1,2]
false.

?- rotate([a,b,c,d],R).
R = [a, b, c, d] ;
R = [b, c, d, a] ;
R = [c, d, a, b] ;
R = [d, a, b, c] ;
false.

?- rotate([1], R).
R = [1] ;
false.

?- rotate([], R).
false.
```

Additionally, `rotate` can be asked whether the second term is a rotation of the first term:

```
?- rotate([a,b,c],[c,a,b]).
true ;
false.

?- rotate([a,b,c],[c,b,a]).
false.
```

Problem 2. (3 points) `ints.pl`

Write a Prolog predicate `ints(-L)` that instantiates `L` to successively longer lists of the integers.

```
?- ints(L).
L = [0] ;
L = [0, 1] ;
L = [0, 1, -1] ;
L = [0, 1, -1, 2] ;
L = [0, 1, -1, 2, -2] ;
L = [0, 1, -1, 2, -2, 3] ;
L = [0, 1, -1, 2, -2, 3, -3] ;
...
```

Problem 3. (3 points) `outin.pl`

Write a Prolog predicate `outin(+L,?R)` that generates the elements of the list `L` in an "outside-in" sequence: the first element, the last element, the second element, the next to last element, etc. If the list has an odd number of elements, the middle element is the last one generated.

Restriction: You may not use `is/2`.

```
?- outin([1,2,3,4,5],X).
X = 1 ;
X = 5 ;
X = 2 ;
X = 4 ;
X = 3 ;
false.

?- outin([1,2,3,4],X).
X = 1 ;
X = 4 ;
X = 2 ;
X = 3 ;
false.

?- outin([1],X).
X = 1 ;
false.

?- outin([],X).
false.
```

Problem 4. (3 points) `btw.pl`

Write a Prolog predicate `btw(+L,+X,?R)` that instantiates `R` to copies of `L` with `X` inserted between each element in turn.

Restriction: You may not use `append` or `between`.

```
?- btw([1,2,3,4,5],---,R).
R = [1, ---, 2, 3, 4, 5] ;
R = [1, 2, ---, 3, 4, 5] ;
R = [1, 2, 3, ---, 4, 5] ;
```

```
R = [1, 2, 3, 4, ---, 5] ;
false.

?- btw([1,2],***,R).
R = [1, ***, 2] ;
false.

?- btw([1],***,R).
false.

?- btw([],x,R).
false.
```

Problem 5. (20 points) `fsort.pl`

Imagine that you have a stack of pancakes of varying diameters that is represented by a list of integers. The list `[3,1,5]` represents a stack of three pancakes with diameters of 3", 1" and 5" where the 3" pancake is on the top and the 5" pancake is on the bottom. If a spatula is inserted below the 1" pancake (putting the stack `[3,1]` on the spatula) and then flipped over, the resulting stack is `[1,3,5]`.

In this problem you are to write a predicate `fsort(+Pancakes,-Flips)` that instantiates `Flips` to a sequence of flip positions that will order `Pancakes`, an integer list, from smallest to largest, with the largest pancake (integer) on the bottom (at the end of the list). `fsort` stands for "flip sort".

`fsort` does not produce a sorted list—its only result is the flip sequence.

The flip position is defined as the number of pancakes on the spatula. In the above example the flip position is `2`. `Flips` would be instantiated to `[2]`.

Below are some examples. Note the use of a set of `case` facts to show a series of examples with one query.

```
% cat fsortcases.pl
case(a, [3,1,5]).
case(b, [5,4,3,2,1]).
case(c, [3,4,5,1,2]).
case(d, [5,1,3,1,4,2]).
case(e, [1,2,3,4]).
case(f, [5]).

% swipl
...
?- [fsortcases,fsort].
% fsortcases compiled 0.00 sec, 7 clauses
% fsort compiled 0.00 sec, 10 clauses
true.

?- case(_,L), fsort(L,Flips).
L = [3, 1, 5],
Flips = [2] ;

L = [5, 4, 3, 2, 1],
Flips = [5] ;

L = [3, 4, 5, 1, 2],
Flips = [3, 5, 2] ;

L = [5, 1, 3, 1, 4, 2],
Flips = [6, 2, 5, 2, 4, 3] ;

L = [1, 2, 3, 4],
Flips = [] ;

L = [5],
Flips = [].
```

Your solution needs only to produce a sequence of flips that results in a sorted stack; the sequences it produces do NOT need to match the sequences shown above. There are some requirements on the flips, however: (1) All flips must be between `2` and the number of pancakes, inclusive. (2) There must be no consecutive identical flips, like `[5,3,3,4]`. (3) `fsort` must always generate exactly one solution.

You may assume that stacks always have at least one pancake and that pancake sizes are always greater than zero.

"Pancake sorting" is a well-known problem. I first encountered it in 1993's Internet Programming Contest. There's even a Wikipedia article about pancake sorting. (Read it!) I debated whether to go with this problem because it's so well known but it's a fun problem and it's interesting to solve in Prolog, so here it is. I did Google up one "solution" in Prolog but it's got some issues! I strongly encourage you to build your Prolog skills by solving this problem without Google's assistance.

The clauses in my current solution have a total of seventeen goals. I don't use `is/2` at all. I do use `max_list`. You might find `nth0` to be useful; if you look at slide 151 closely you'll see it can be used to both extract values and find values, among other things. There's an `nth1`, too, if you find one-based thinking to be a better choice.

I've placed this problem, `fsort.pl`, early in the line-up in hopes of getting you thinking about it early but don't get hung up on it.

Problem 6. (15 points) `pipes.pl`

In this problem you are to write a simple command interpreter to perform manipulations on a set of pipes. These are pipes like you buy at Home Depot, not UNIX pipes! Each pipe has a name, length, and diameter.

The commands for the interpreter are in the form of Prolog terms. The calculator shown on slide 219 and following is a good starting point for this problem.

The interpreter provides the following commands:

`pipes` Show the current set of pipes. The pipes are shown in alphabetical order by name.

`weld(A, B)`
The pipe named `B` is welded onto the pipe named `A`. `A` and `B` must have the same diameter. After welding, `A` has the combined length of `A` and `B`. Pipe `B` no longer exists.

`cut(A, L, B)`
A section of length `L` is cut from the pipe named `A`. The section cut off becomes a pipe named `B` having the same diameter as `A`. `L` must be less than the length of `A`.

`trim(A,L)` A section of length `L` is cut from the pipe named `A` and discarded. `L` must be less than the length of `A`.

`help` Print a help message with a brief summary of the commands.

`echo` Toggle command echo and prompting. (See below for details on this.)

Assume that all lengths are integers.

Use a predicate such as `setN` to establish a set of pipes to work with:

```
set1 :-
    retractall(pipe(_,_,_)),
    assert(pipe(a,10,1)),
    assert(pipe(b,5,1)),
    assert(pipe(c,20,2)).
```

The command interpreter is started with the predicate `run/0`.

A session with the interpreter is shown below. No blank lines have been inserted or deleted.

```
% swipl -l pipes
...

?- set1.
true.

?- run.

Command? help.
pipes -- show the current set of pipes
weld(P1,P2) -- weld P2 onto P1
cut(P1,P2Len,P2) -- cut P2Len off P1, forming P2
trim(P,Length) -- trim Length off of P
echo -- toggle command echo
help -- print this message
q -- quit

Command? pipes.
a, length: 10, diameter: 1
b, length: 5, diameter: 1
c, length: 20, diameter: 2

Command? weld(a,b).
b welded onto a

Command? pipes.
a, length: 15, diameter: 1
c, length: 20, diameter: 2


Command? trim(a,12).
12 trimmed from a

Command? pipes.
a, length: 3, diameter: 1
```

```
c, length: 20, diameter: 2

Command? cut(c,10,d).
10 cut from c to form d

Command? pipes.
a, length: 3, diameter: 1
c, length: 10, diameter: 2
d, length: 10, diameter: 2


Command? cut(c,6,c1).
6 cut from c to form c1

Command? pipes.
a, length: 3, diameter: 1
c, length: 4, diameter: 2
c1, length: 6, diameter: 2
d, length: 10, diameter: 2

Command? weld(d,c1).
c1 welded onto d

Command? pipes.
a, length: 3, diameter: 1
c, length: 4, diameter: 2
d, length: 16, diameter: 2

Command? q.
true.
```

Your implementation must handle four errors:

Cutting or welding a pipe that doesn't exist.

Cutting with a result pipe that does exist.

Cutting the full length (or more) of a pipe with cut or trim.

Welding pipes with differing diameters.

If an error is detected, the pipes are unchanged. Here are examples of error handling:

```
Command? pipes.
a, length: 10, diameter: 1
b, length: 5, diameter: 1
c, length: 20, diameter: 2

Command? cut(x,10,y).
x: No such pipe

Command? weld(x,y).
x: No such pipe

Command? weld(a,x).
x: No such pipe
```

```
Command? cut(a,5,b).
b: pipe already exists

Command? cut(a,10,a2).
Cut is too long!

Command? trim(a,15).
Cut is too long!

Command? weld(a,c).
Can't weld: differing diameters

Command? pipes.
a, length: 10, diameter: 1
b, length: 5, diameter: 1
c, length: 20, diameter: 2
```

Prompting, and the `echo` command

Don't use `write('\nCommand? ')` to prompt the user. Instead, use the built-in `prompt/2` to set the prompt, like this: `prompt(_, '\nCommand? ').` Then when `read(X)` is called, `'\nCommand? '` will automatically be printed first.

To make `tester` output more usable there is an `echo` command. By default, the `Command?` prompt is printed and the command entered is not echoed. The `echo` command toggles both behaviors: entering `echo` causes prompting to be turned off and echo to be turned on. A subsequent `echo` command reverts to the default behavior. In the following example, the text typed by the user is in bold and underlined:

```
?- run.

Command? cut(a,1,a2).
1 cut from a to form a2

Command? echo.
Echo turned on; prompt turned off
cut(a,1,a3).

Command: cut(a,1,a3)
1 cut from a to form a3
pipes.

Command: pipes
a, length: 8, diameter: 1
a2, length: 1, diameter: 1
a3, length: 1, diameter: 1
b, length: 5, diameter: 1
c, length: 20, diameter: 2
weld(a,a2).

Command: weld(a,a2)
a2 welded onto a
echo.

Command: echo
Echo turned off; prompt turned on
Command? q.
```

```
true.

?-
```

Implementing the toggling of echoing is a little tricky. Here's a sketchy hint, for students who want a challenge:

> Manipulate an `echo/0` fact with `assert(echo)` and `retract(echo)`. To produce no prompt at all, use the built-in `prompt/2` like this: `prompt(_, '')`.
>
> **Important**: to allow that manipulation of `echo/0` with `assert` and `retract` you'll need to declare `echo` as *dynamic*. Have the following line as the first line in your `pipes.pl`:
>
> ```
> :-dynamic(echo/0).
> ```

A more detailed hint is in http://www.cs.arizona.edu/classes/cs372/spring16/a10/echo-hint.html

TL;DR

The built-in help provides a quick summary:

```
?- run.

Command? help.
pipes -- show the current set of pipes
weld(P1,P2) -- weld P2 onto P1
cut(P1,P2Len,P2) -- cut P2Len off P1, forming P2
trim(P,Length) -- trim Length off of P
echo -- toggle command echo
help -- print this message
q -- quit
```

And, handle these errors:
Cutting or welding a pipe that doesn't exist.
Cutting with a result pipe that does exist.
Cutting the full length (or more) of a pipe with cut or trim.
Welding pipes with differing diameters.

I won't test with cases involving multiple errors, like a too-long cut that names an existing pipe as the result.

Problem 7. (20 points) `connect.pl`

In this problem you are to write a predicate `connect` that finds and displays a suitable sequence of cables to connect two pieces of equipment that are some distance apart. Each cable is specified by a three element list. Here is a list that represents a twelve-foot cable with a male connector on one end and a female connector on the other:

```
[m,12,f]
```

Let's consider an example of using `connect` to produce a sequence of cables. Imagine that to your left is a piece of equipment with a male connector. On your right, fifteen feet away, is a piece of equipment with a female connector. To connect the equipment you have two cables:

- A ten-footer with a male connector on one end and a female on the other.

- A seven-footer with a female connector on one end and a male on the other.

The following query represents the situation described above.

```
?- connect([ [m,10,f], [f,7,m] ], m, 15, f).
```

`connect`'s first argument is a list with the two cables. The second, third, and fourth arguments respectively represent the gender of the connector of the equipment on the left (male—`m`), the distance between the equipment (15 feet), and the gender of the connector of the equipment on the right (female—`f`).

Here's the query and its result:

```
?- connect([ [m,10,f], [f,7,m] ], m, 15, f).
F----------MF-------M
true.
```

We see that a connection is possible in this case; a valid sequence of connections is shown. Observe that the first cable was reversed to make the connection. The number of dashes is the length of the cable. There's some slack in the connection—only fifteen feet needs to be spanned but the total length of the tables is seventeen feet. That's fine.

Note that the output has no representation of the pieces of equipment on the left and right that we're connecting with the cables.

Only male/female connections are valid in the world of `connect.pl`.

In some cases a connection cannot be made, but `connect` always succeeds:

```
?- connect([[m,10,f], [f,7,m] ], m, 25, f).
Cannot connect
true.

?- connect([[m,10,f], [f,7,m] ], m, 15, m).
Cannot connect
true.
```

More examples:

```
?- connect([[m,1,m],[f,1,f],[m,10,m],[f,5,f],[m,3,f]], m, 20, f).
F-FM-MF-----FM----------MF---M
true.

?- connect([[m,1,m],[f,1,f],[m,10,m],[f,5,f],[m,3,f]], m, 20, m).
Cannot connect
true.

?- connect([[m,1,m],[f,1,f],[m,10,m],[f,5,f],[m,3,f]], m, 10, f).
F-FM-MF-----FM----------M
true.

?- connect([[m,10,f]], m, 1, f).
F----------M
true.
```

IMPORTANT: The ordering of cables your solution produces for a particular connection need NOT match that shown above. Any valid ordering is suitable. (A Ruby program, `a10/pc.rb`, analyzes the output.)

Assume the arguments to `connect` are valid—you won't see two-element lists, non-numeric or non-positive lengths, ends other than `f` and `m`, etc. Assume that all lengths are integers. Assume that the distance to span is greater than zero.

You can approach this problem using an approach similar to that in the pit-crossing example in the slides.

Note that you do not need to use all the cables or exactly span the distance.

My current solution is around 25 goals; about a third of those are related to producing the required output.

Problem 8. (8 points) `buy.pl`

In this problem the task is to print a bill of sale for a collection of items. Several predicates provide information about the items. The first is `item/2`, which associates an item name with a description:

```
item(toaster, 'Deluxe Toast-a-matic').
item(antfarm, 'Ant Farm').
item(dip, 'French Onion Dip').
item(twinkies, 'Twinkies').
item(lips, 'Chicken Lips').
item(hamster, 'Hamster').
item(rocket, 'Model rocket w/ payload bay').
item(scissors, 'StaySharp Scissors').
item(rshoes, 'Running Shoes').
item(tiger, 'Sumatran tiger').
item(catnip, '50-pound bag of catnip').
```

The second predicate is `price/2`, which associates an item name with a price in dollars:

```
price(toaster, 14.00).
price(antfarm, 7.95).
price(dip, 1.29).
price(twinkies, 0.75).
price(lips, 0.05).
price(hamster, 4.00).
price(rocket, 12.49).
price(scissors, 2.99).
price(rshoes, 59.99).
price(tiger, 749.95).
```

The third is `discount/2`, which associates a discount percentage with some, possibly none, of the items:

```
discount(antfarm, 20).
discount(lips, 40).
discount(rshoes, 10).
```

Finally, state law prohibits same-day purchase of some items. `dontmix/2` specifies prohibitions. Here are some examples:

```
dontmix(scissors,rshoes).
dontmix(hamster,rocket).
dontmix(tiger,catnip).
```

You can only ensure that any prohibited pairings are not included in a single purchase; the well-intentioned prohibitions can be thwarted by making multiple trips to the store!

You are to write a predicate `buy(+Items)` that prints a bill of sale for the specified items. If any mutually prohibited items are in the list, that should be noted and no bill printed.

```
?- buy([hamster,twinkies,hamster,toaster]).
Hamster............................4.00
Twinkies...........................0.75
Hamster............................4.00
Deluxe Toast-a-matic..............14.00
----------------------------------------
Total                            $22.75
true.

?- buy([lips,lips,lips,dip]).
Chicken Lips.......................0.03
Chicken Lips.......................0.03
Chicken Lips.......................0.03
French Onion Dip...................1.29
----------------------------------------
Total                             $1.38
true.

?- buy([scissors,dip,rshoes]).
State law prohibits same-day purchase of "Running Shoes" and
"StaySharp Scissors".
true.
```

You may assume that all items named in a `buy` are valid and that a price exists for every item.

<u>Prohibited items are shown in alphabetical order.</u> If several mutually prohibited items are named in the same `buy` only the first conflict is noted.

Here's the `format/2` specification I use to produce the per-item lines: `'~w~`.t~2f~40|~n'`. The backquote-period sequence causes the enclosing tab to fill with periods.

A set of facts for testing is in `a10/buyfacts.pl` . Include the line

```
:-[a10/buyfacts].
```

in your `buy.pl` to consult the file. For grading I may tests with other sets of facts, too.

Problem 9. (8 points) `mishaps.pl`

The following logic puzzle, "Rural Mishaps", was written by Margaret Shoop. It was published in *The Dell Book of Logic Problems #2*.

> "A butt by the family cow was one of the five different mishaps that befell Farmer Brown, his wife, his daughter, his teenage son, and his farmhand one summer morning. From the rhyme that follows, can you determine the mishap that happened to each of the five, and the order in which

the events occurred?

"The garter snake was surprised in a patch
And bit a grown man's finger.
One person who weeded a flower bed
Received a nasty stinger.
The farmer's mishap happened first;
Son Johnny's happened third.
When Mr. Reston was kicked by the mule,
He said, "My word! My word!"
The sting of the bee was the fourth mishap
To befall our rural cast.
Neither it nor the wasp attacked Mrs. Brown
Whose mishap wasn't the last."

For `mishaps.pl` you are to encode as Prolog goals the pertinent information in the above rhyme and then write a predicate `mishaps/0` that uses those goals to solve the puzzle.

`a10/mishaps.out` shows a run of `mishaps/0` with the exact output you are to produce, but you might enjoy the challenge of solving the puzzle using Prolog **before** looking at the expected output or trying the tester.

The challenge of this problem is of course the encoding of the information as Prolog goals. **Solutions must both produce the correct output and accurately encode the information as stated in the rhyme to earn full credit.** The tester will check for correct output; we'll check manually for accurate encoding.

Implementation notes

Obviously, this problem is in the style of The Zebra Puzzle, which starts on slide 254. The code for the Zebra Puzzle establishes a number of constraints for the list `Houses`. In this problem you'll want to establish constraints for a list of mishaps instead of a list of houses.

You might start like this:

```
mishaps(Mishaps) :- Mishaps = [ ... ].
```

A similar start for the Zebra puzzle would be this:

```
zebra(Houses) :- Houses = [house(norwegian, _, _, _, _), _,
                            house(_, _, _, milk, _), _, _].
```

Querying `zebra(H)` produces only one possible value for `H`, the list of houses:

```
?- zebra(H).
H = [house(norwegian, _G1222, _G1223, _G1224, _G1225), _G1227,
house(_G1233, _G1234, _G1235, milk, _G1237), _G1239, _G1242].
```

Let's add a `member` goal that encodes the statement that the Englishman lives in the red house:

```
zebra(Houses) :- Houses = [house(norwegian, _, _, _, _), _,
                           house(_, _, _, milk, _), _, _],
                 member(house(englishman, _, _, _, red), Houses).
```

Now we get four possible values for H: (blank lines added)

```
?- zebra(H).
H = [house(norwegian, _G2194, _G2195, _G2196, _G2197),
house(englishman, _G2218, _G2219, _G2220, red), house(_G2205,
_G2206, _G2207, milk, _G2209), _G2211, _G2214] ;

H = [house(norwegian, _G2194, _G2195, _G2196, _G2197), _G2199,
house(englishman, _G2206, _G2207, milk, red), _G2211, _G2214] ;

H = [house(norwegian, _G2194, _G2195, _G2196, _G2197), _G2199,
house(_G2205, _G2206, _G2207, milk, _G2209), house(englishman,
_G2218, _G2219, _G2220, red), _G2214] ;

H = [house(norwegian, _G2194, _G2195, _G2196, _G2197), _G2199,
house(_G2205, _G2206, _G2207, milk, _G2209), _G2211,
house(englishman, _G2218, _G2219, _G2220, red)].

?- findall(r, zebra(H), Results), length(Results,N).
Results = [r, r, r, r],
N = 4.
```

If we add a goal that states that the Spaniard owns the dog, we go up to twelve possible values for H:

```
zebra(Houses) :- Houses = [house(norwegian, _, _, _, _), _,
                        house(_, _, _, milk, _), _, _],
                 member(house(englishman, _, _, _, red), Houses),
                 member(house(spaniard, dog, _, _, _), Houses).
```

```
?- findall(r, zebra(H), Results), length(Results,N).
Results = [r, r, r, r, r, r, r, r, r|...],
N = 12.
```

Some goals will make the number of possible values for the list of houses go up, and other goals will make the number of possible values go down. If we properly encode all the information, we'll end up with only one possible value for the list of houses.

Follow a similar process when adding goals to represent information about the mishaps. That is, add goals to mishaps/1 one at a time. Query mishaps(M) after each addition.

If we inadvertently introduce a contradiction, like length(Houses,6), we'd see this:

```
?- zebra(H).
false.
```

If you add a goal and find that mishaps(M) then fails, you'll need to step back and consider why that new goal creates a situation with no possible solutions. You might try leaving the new goal in place and commenting one or more earlier goals to find the conflict.

When you've got mishaps/1 working—producing a single possibility for the list of mishaps—use it to write mishaps/0.

Problem 10. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with "`Hours:`". There must be only one "`Hours:`" line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a10/turnin` to submit your work.

Here's what I see as of press time for the sizes of my solutions, with comments stripped:

```
$ a10/plsize $(grep -v txt a10/delivs)
rotate.pl: 10
ints.pl: 22
outin.pl: 9
btw.pl: 13
fsort.pl: 54
pipes.pl: 262
connect.pl: 100
buy.pl: 65
mishaps.pl: 77
```

Miscellaneous

Aside from `->` and `;`, and any per-problem restrictions notwithstanding, you can use any elements of Prolog that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Prolog slides 1-272.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) In Prolog, `%` is comment to end of line. Comments with `/* ... */`, just like in Java, are supported, too.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 10 to 12 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put twelve hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached twelve hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.

CSC 372, Spring 2016
Assignment "V" (for video!)
Due: Wednesday, May 4 at 23:59:59

In a nutshell

The essence of this assignment is simple:
(1) Find in some programming language an interesting element or aspect that you know little about.
(2) Experiment with that interesting thing and see what you can learn about it.
(3) Make a 7-minute video of yourself talking about that interesting thing.

This assignment is worth 50 points—5% of your final grade.

The final Prolog assignment will also be due on May 4, so budget your time accordingly.

Finding a language

A key requirement is that you pick a topic that you know little about. If you don't know anything about a language then any aspect of that language would qualify, although many might not be very interesting, like straightforward analogs of Java control structures and data types. I wouldn't want you talking about iterators in Ruby, but Ruby threads are certainly fair game. If you feel like taking on a beast, you might try monads in Haskell. If you want to do something in Prolog, ok the topic with me first, to be sure it won't be something we'll be covering.

If you want to work with a language that's new to you, there are thousands to choose from! You'll probably want to choose a language that has an implementation available for your machine or is installed on lectura so that you can experiment with it. A language with a REPL makes experimentation easier, of course.

Slide 24 in the intro set lists a number of prominent langauges. Lots of choices, maybe too many, can be found at http://en.wikipedia.org/wiki/List_of_programming_languages. Some older languages have quite a few features that are fairly different from what we commonly see today and might present some low hanging fruit. COBOL, Forth, MUMPS, and (my favorite old language) SNOBOL4 come to mind.

Finding an interesting thing

In elementary school you might have written a report on "The Civil War". By the time you got to college you probably realized that "The Civil War" was a hopelessly big topic to cover in a single report. But maybe "The Battle of Picacho Peak" would have been about the right size.

Similarly here, **the task is not to cover the full language but just one single aspect or element**. Had we not covered these topics in class, each of the following would make a fine seven-minute topic:

Type inferencing in Haskell
Curried functions and partial evaluation in Haskell
Computations with map, filter, and fold
Ruby strings, arrays, or hashes
Ruby iterators contrasted with conventional iteration

Here are some seven-minute topic ideas:

- Functors or monoids in Haskell
- Ruby's `ObjectSpace`
- Contrast regular expressions in Java vs. Ruby
- Explore some elements of Ruby regular expressions that we didn't talk about
- `method_missing` in Ruby
- List comprehensions in Python
- "Globals" in MUMPS
- `PICTURE` in COBOL
- Lambda expression support in Java 8
- `java.util.stream`
- Prototypes in JavaScript
- ~~Ruby's `Matrix` class~~ (I've seen enough of these to last a lifetime, so this topic is banned!)
- Control structures in `bash`
- Arrays in `bash` (only for those with a strong stomach!)
- Typeless-ness in BCPL
- Pattern matching in SNOBOL4
- "magic methods" in Python
- Channels in Go
- PHP arrays
- String scanning in Icon
- Stack-based programming in PostScript or Forth
- Using C's `sizeof` to explore the language
- Fun with the C preprocessor (beyond what's covered in 352)

One way to look for topic ideas is to browse tables of contents of books on Safari. For high-bandwith browsing, go to the library.

Think of your goal as being technical entertainment. Your challenge is to find some interesting aspect or element on which you think you can create material that will hold somebody's attention for seven minutes.

You can make any assumptions you want about your audience. For example, you might assume that your audience is your 372 classmates, implying that they know Java, Ruby, Haskell, and, by the end, Prolog. If you want to talk about the C preprocessor, you should assume the audience knows some C.

It's fine to use examples you discover on the web and in books; just take a moment to verbally cite the source, like the author's name, the name of the book, or the website. It's fine to watch videos to help find a topic or learn more about your topic.

You don't need to get your topic approved in advance but we'll be happy to offer advice on your ideas, or help you find an idea.

The video

If your language has a REPL, your video might just be a screen recording of you demonstrating a feature and narrating what you're doing.

If you're inclined to write slides with PowerPoint, Keynote, or whatever, and record a presentation, perhaps even projected on a screen in a classroom, that's fine.

You can work with pencil and paper, writing out examples by hand, as one might when working on an Elmo, and shoot video with a cell phone or a camcorder.

Any mix of the above modes or additional modes is fine.

Try a test video well before the deadline. For example, if you're planning to use your cell phone to shoot video, be sure the resolution is adequate to make text legible and that your phone has enough space for whatever length segments you plan to shoot.

Between screen recordings, cell phones, and digital cameras I imagine most everybody will already have the equipment they need but if not, http://www.library.arizona.edu/services/equipment-lending and http://www.uits.arizona.edu/departments/oscr/locations/gtg have loaner cameras and tripods.

I don't expect you to spend time editing video to make it perfect, or anywhere near perfect. You're certainly free to edit if you want, but I'm fine with a single "take".

It's fine to run a little past seven minutes, but keep it under eight minutes. If you're significantly under seven minutes there will be a proportional deduction in your grade. I'll stop watching at eight minutes.

Turning in your work

Park your video somewhere on the web and then submit a plain text file named `video.txt` using `/cs/www/classes/cs372/spring16/av/turnin`. (Just run that script in whatever directory your `video.txt` resides.) The file should be structured as follows:

First line: A title for your video
Second line: The URL for your video
Third line: "`Post: yes`" or "`Post: no`"
Following lines: Any sort of comments or observations you'd like to make, if any.

I'll collect the URLs for the videos that are "`Post: yes`" and post that list on Piazza. I'll use exactly the title you specify. If you want your name or a pseudonym shown, include it in the title, maybe with "... by John Smith" or "The Amazing Haskellon Explores Monads", for example.

Specifying "`Post: yes`" earns you a five-point bonus. You can remain anonymous, except to those who might recognize your voice, for example.

Late submissions accepted

Unlike other assignments, late submissions will be accepted on this assignment, with a penalty of two points per 24 hours or any fraction thereof, with a maximum of five days (5*24 hours) late.

Expectations

I'm picturing that a typical student will spend 6-8 hours on this assignment.

I don't expect you to achieve full understanding of your topic; you just need to know enough to fill seven minutes. It's fine to point out some things that you were unable to figure out.

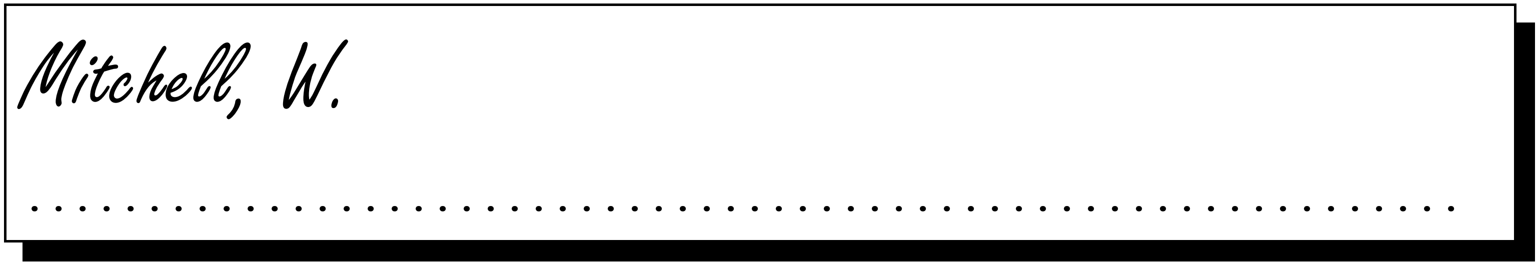
If you have a disability that makes this assignment difficult for you, please let me know; we'll work out an accommodation.

QUIZ!

Quiz Stuff

Use a full sheet of 8½x11" paper. (Half sheet? Half credit!)

Put your last name and first initial in the **far upper left hand corner** of the paper, where a staple would hit it. (It helps when sorting quizzes!)



Mitchell, W.

No need to write out questions.

Numbering responses may help you avoid overlooking a question; it's ok to go ahead and pre-number your sheet.

Two minutes; five questions, plus one extra credit question.

Can everybody see this line?

Quiz 1, January 14, 2016

2 minutes; ½ point/answer; 2½ points total

1. What is the name of any one programming language created before 1975?

2. How many programming languages are there? (pick one: dozens, hundreds, thousands)

3. Who founded the UA CS department and in what year?

4. Name an area of research for which the UA CS department was recognized worldwide in the 1970s and 1980s.

5. Ideally, what percentage of your classmates will get an "A" in this course?

EC ½ point: What's the name of the most recently created language here in UA CS?

Quiz 2, January 21, 2016
3 minutes; 3 points

1. Write a Java expression that has a side effect.

2. Write a Haskell function that computes the area of a rectangle given its width and height. Append **`::Int`** to force it to operate on **`Int`**s.

3. What's the type of the function you wrote in the previous problem?

4. What does REPL stand for? Or, what's the essential functionality provided by a REPL?

5. What's a characteristic of the functional programming paradigm?

6. Imagine that **`:type f`** shows this: **`Foo a => a -> Char`** What does that type mean?

Solutions

1. *Write a Java expression that has a side effect.* **`x++`**

2. *Write a Haskell function that computes the area of a rectangle given its width and height. Append* **`::Int`** *to force it to operate on* **`Int`***s.* **`area w h = w * h :: Int`**

3. *What's the type of the function you wrote in the previous problem?*

   `Int -> Int -> Int`

4. *What does REPL stand for? Or, what's the essential functionality provided by a REPL?* Read-Eval-Print Loop

5. *What's a characteristic of the functional programming paradigm?*

   See slides 24-25. My quick answer: "functions are values"

6. *Imagine that* ***`:type f`*** *shows this:* ***`Foo a => a -> Char`***
   What does that type mean?

   `f` is a function that requires a value whose type is a member of the type class **`Foo`**. **`f`** produces a **`Char`**.

Quiz 3, February 2, 2016
3 minutes; ½ + ½ + 2 points

1. Add parentheses to the following expression to show the order of operations: **`a b + x y z`**

2. The **`length`** function produces the length of a list. What's the type of **`length`**?

3. Write a function **`nzs`** that returns the number of zeroes in a list. (2 points!)

   ```
   > nzs [5,0,0,5]
   2
   ```

EC ½ point:
Write a function **`f`** whose type is inferred to be **`a -> a -> a`**. Be sure that **`a`** doesn't have a class constraint, like **`Eq a`**.

Solutions

1. *Add parentheses to the following expression to show the order of operations:* ***`a b + x y z`***

   ```
   (a b) + ((x y) z)
   ```

2. *The* ***`length`*** *function produces the length of a list. What's the type of* ***`length`****?* **`[a] -> Int`**

3. *Write a function* ***`nzs`*** *that returns the number of zeroes in a list.*

   Two solutions:

   ```
   nzs [] = 0
   nzs (0:t) = 1 + nzs t
   nzs (_:t) = nzs t

   nzs [] = 0
   nzs (h:t)
       | h == 0 = 1 + nzs t
       | otherwise = nzs t
   ```

EC ½ point: Write ***`f`*** *whose type is inferred to be* ***`a -> a -> a`****.*

```
f x y = head [x, y]
```

Quiz 4, February 9, 2016

3 minutes; ½ + ½ + 2 points

1. Give a simple definition for "higher order function".

2. What's the type of **map**? Here's a reminder of how **map** works:

```
> map (add 2) [1..5]
[3,4,5,6,7]
```

3. Write a function **atb f x y** that calls the function **f** with the larger of **x** and **y**. (2 points!)

```
> atb negate 7 2
-7

> atb length "aa" "zzz"
3
```

EC ½ point: In Haskell, what's a "section"? (Ok to just show an example.)

Solutions

1. *Give a simple definition for "higher order function".*
   A function that has one or more arguments that are functions.

2. *What's the type of* **`map`***?*

   ```
   (a -> b) -> [a] -> [b]
   ```

3. *Write a function* **`atb f x y`** *that calls the function* **`f`** *with the larger of* **`x`** *and* **`y`***. (2 points!)*
   Two solutions:

   ```
   atb f x y = f (if x > y then x else y)

   atb f x y
       | x > y = f x
       | otherwise = f y
   ```

EC ½ point: In Haskell, what's a "section"? (Ok to just show an example.)

Short answer: (+3) is a section.

Long answer: A syntactic mechanism that allows creation of a partial application of a binary operator by supplying either operand.

Quiz 5, February 18, 2016
3 minutes; 2 + ½ + ½ points

1. Consider folding a list of strings into the total length of the strings:

   ```
   > foldl f 0 ["just", "a", "test"]
   9

   > foldl f 0 ["abc"]
   3
   ```

   For this problem you are to write a folding function **`f`** that would work as shown with the **`foldl`** calls above

2. What's a difference between **`foldl`** and **`foldr`**?

3. What's a difference between **`foldr`** and **`foldr1`**?

Solutions

1.

```
> let f acm elem = acm + length elem

> foldl f 0 ["just", "a", "test"]
9

> foldl f 0 ["abc"]
3
```

*2. What's a difference between **`foldl`** and **`foldr`**?*

`foldl` folds the list from left to right but **`foldr`** folds from right to left.

*3. What's a difference between **`foldr`** and **`foldr1`**?*

`foldr` needs a "zero" value, for the case of an empty list.

Quiz 6, February 25, 2016
3 minutes; 1 + (½ * 6) points

1. What's a fundamental characteristic of a statically typed language? (one point)

2. Name a language that uses static typing.

3. Name a language that uses dynamic typing.

4. Assuming **`s = "abcdef"`**, what's the value of **`s[1,3]`**?

5. Assuming **`s = "abcd"`**, what's the value of **`s[1..-1]`**?

6. What's a key difference between Ruby arrays and Haskell lists?

7. What program provides a REPL for Ruby?

EC: Who invented Ruby?

Solutions

1. *What's a fundamental characteristic of a statically typed language? (one point)*

   The type of every expression can be determined without running the code.

2. *Name a language that uses static typing.* Java, Haskell, C

3. *Name a language that uses dynamic typing.* Ruby, Python, Icon

4. *Assuming* **`s = "abcdef"`**, *what's the value of* **`s[1,3]`**? **`"bcd"`**

5. *Assuming* **`s = "abcd"`**, *what's the value of* **`s[1..-1]`**? **`"bcd"`**

6. *What's a key difference between Ruby arrays and Haskell lists?*

   Ruby arrays are heterogeneous—they can hold a mix of types.

7. *What program provides a REPL for Ruby?* **`irb`**

EC: Who invented Ruby? Yukihiro Matsumoto ("Matz" ok!)

Quiz 7, March 8, 2016

3 minutes; ½ + ½ + ½ + ½ + 1 points

1. How can we quickly tell whether an identifier is a global variable?

2. In Ruby's world, what is an iterator?

3. What keyword does an iterator use to invoke a block?

4. What element of Haskell is a Ruby block most like?

5. Assuming **`x`** is an array, print the elements in **`x`**, one per line, using the iterator **`each`** with an appropriate block.

E.C. ½ point: Write a trivial iterator.

Solutions

1. *How can we quickly tell whether an identifier is a global variable?* Starts with a dollar sign, like **`$x`**.

2. *In Ruby's world, what is an iterator?*

   An iterator is a method that can invoke a block.

3. *What keyword does an iterator use to invoke a block?* **`yield`**

4. *What element of Haskell is a Ruby block most like?*

   An anonymous function, I'd say.

5. *Assuming **`x`** is an array, print the elements in **`x`**, one per line, using the iterator **`each`** with an appropriate block.*

   ```
   x.each {|e| puts e}
   ```

E.C. ½ point: Write a trivial iterator.

```
def itr
   yield 7
end
```

Quiz 8, March 22, 2016
3 minutes; ½ point each

1. What Ruby operator looks to see if a string contains a match for a regular expression?

2. When that matching operator is successful, what value does it produce?

3. After a successful match, what does the predefined global **`` $` ``** hold? (dollar-backquote)

4. What's the longest string that can be matched by the RE **`/a..z/`**?

5. Languages vary in their level of support for regular expressions. What level of support does Ruby provide—library support or syntactic support?

6. Support your answer for the previous question with a brief explanation.

Solutions

1. *What Ruby operator looks to see if a string contains a match for a regular expression?* **`=~`** is the "match" operator.

2. *When that matching operator is successful, what value does it produce?* The starting position of the first match

3. *After a successful match, what does the predefined global* **`$``** *hold? (dollar-backquote)*

   The portion of the string preceding the match

4. *What's the longest string that can be matched by the RE* ***`/a..z/`****?*

   Four characters

5. *Languages vary in their level of support for regular expressions. What level of support does Ruby provide—library support or syntactic support?* Syntactic support

6. *Support your answer for the previous question with a brief explanation.*

   The expression ***`/text/`*** designates a regular expression.

Quiz 9, March 22, 2016
4 minutes; 2 + 1½ + ½ points

1. In English, describe strings that are matched by this Ruby regular expression: **`/^[xyz]+[abc]?\d{2,3}/`**

2. Write a definition for a Ruby class named **`X`**. Instances of **`X`** are created by specifying a string, like **`X.new("abc")`**. **`X`** has one method, named **`f`**, that returns the length of the string that the instance was created with.

3. The line "**`attr_reader :x`**" in a class definition specifies that there should be a getter for the instance variable **`x`**. What's especially interesting about **`attr_reader`**?

EC ½ point: Briefly, what's the difference between a language being extensible vs. being mutable?

Solutions

1. *Describe strings matched by* ***`/^[xyz]+[abc]?\d{2,3}/`***

   Starts with one or more occurrences of **`x`**, **`y`**, or **`z`**; followed by an optional **`a`**, **`b`**, or **`c`**; followed by two or three digits.

2. *Write a definition for a Ruby class named* ***`X`****.* ...

   ```
   class X
       def initialize s
           @f = s.size
       end
       attr_reader :f
   end
   ```

3. *The line "****`attr_reader :x`****" in a class definition specifies that there should be a getter for the instance variable* ***`x`****. What's especially interesting about* ***`attr_reader`****?*

   `attr_reader` is a method that generates a getter method.

EC ½ point: If a language is mutable, the meanings of operations can be changed, where extensibility only allows for providing meaning for previously undefined operations.

Quiz 10, March 29, 2016
45 seconds; ½ point each

With Prolog in mind...

1. Write an example of an atom.

2. Write an example of a fact.

3. Write an example of a query.

EC: What's the command to run SWI Prolog on lectura?

Solutions

1. *Write an example of an atom.* **`food`**

2. *Write an example of a fact.* **`food(apple).`**

3. *Write an example of a query.* **`food(apple).`**

Shorter:

```
a
a(b).
a(b).
```

EC: What's the command to run SWI Prolog on lectura?

```
swipl
```

Quiz 11, April 12, 2016

4 minutes; ½ + 1 + ½ + 1 + ½ + ½ points

1. Write an example of a structure with two terms.

2. What are two distinct computations that can be done with the predicate **`food/1`** that we've been using?

3. What does the notation **`f/3`** mean?

4. Draw and label the ports of the four-port model.

5. What is the output of the following query?

```
?- A=1, B=2, write(A), A=B, write(B).
```

6. Given these facts, **`a(1). a(1,2). a(2).`**
what is output by the following query?

```
?- a(X), writeln(X), fail.
```

Solutions

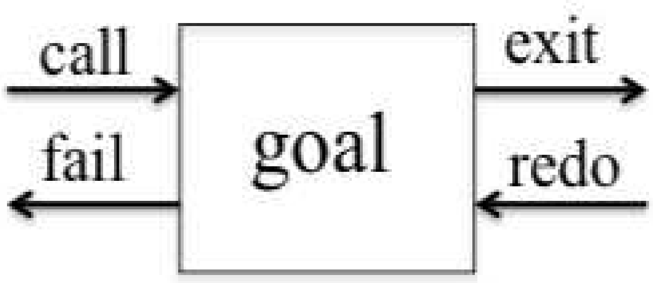
1. *Write an example of a structure with two terms.* `x(1,2)`

2. *What are two distinct computations that can be done with the predicate* **`food/1`** *that we've been using?*

   (1) Ask if something is food. (2) Enumerate all foods.

3. *What does the notation* ***`f/3`*** *mean?* `f` is a three-term predicate.

4. *Draw and label the ports of the four-port model.*



5. *What is the output of the following query?*

```
?- A=1, B=2, write(A), A=B,
   write(B).
1
false.
```

6. *Given these facts,* `a(1). a(1,2). a(2).` *what is output by the following query?*

```
?- a(X), writeln(X), fail.
```

```
1
2
false.
```

Quiz 12, April 19, 2016
3 minutes; 1 + 2 points

1. Write a predicate **`same(?A, ?B, ?C)`** that expresses the relationship that **`A`**, **`B`**, and **`C`** are equal.

```
?- same(1,1,1).
true.

?- same(a,X,a).
X = a.
```

2. Write a predicate **`p(+L)`** that prints the elements of **`L`** that are integers, one per line. (Use **`integer(?X)`** to test.) Be sure it always succeeds!

```
?- p([10,b,c,2,4]).
10
2
4
true.
```

Solutions

1. Write a predicate **`same(?A, ?B, ?C)`** that expresses the relationship that **`A`**, **`B`**, and **`C`** are equal.

```
same(X,X,X).
```

2. Write a predicate **`p(+L)`** that prints the elements of **`L`** that are integers, one per line.

```
p(L) :- member(E,L), integer(E),
        writeln(E), fail.
p(_).
```

Quiz 13, April 26, 2016

4 minutes; 4 points

1. What would Prolog show for **A** and **B** for the folllowing query?

   ```
   ?- [_,A|B] = [1,2,3,4].
   ```

2. Without using **append**, write **head(List, Elem)**.

3. Without using **append**, write **last(List, LastElem)**.

4. What are two predicates that can be used, respectively, to add or remove facts?

EC ½ point: Write **g(L,E)** that generates each element of **L** twice:

```
?- g([a,b],E).
E = a ;
E = a ;
E = b ;
E = b ;
false.
```

Solutions

1. *What would Prolog show for* **A** *and* **B** *for the folllowing query?*

```
?- [_,A|B] = [1,2,3,4].
A = 2, B = [3, 4].
```

2. *Without using* **append***, write* **head(List,Elem)***.*

```
head([H|_],H).
```

3. *Without using* **append***, write* **last(List, LastElem)***.*

```
last([X],X).
last([_|T],X) :- last(T,X).
```

4. *What are two predicates that can be used, respectively, to add or remove facts?* **assert** and **retract**

EC: Write **g(L,E)** *that generates each element of* **L** *twice.*

```
g([H|_],H).
g([H|_],H).
g([_|T],E) :- g(T,E).
```

Quiz 14, April 28, 2016
4 minutes; 4 points

1. Briefly describe the general approach used to solve the pit-crossing puzzle in the slides.

2. Write a predicate **`inc`** that uses **`assert`** and **`retract`** to increment a counter maintained as a **`count/1`** fact. It reports the new value.

```
?- count(N).
N = 0.

?- inc.
Count is 1
true.

?- inc.
Count is 2
true.

?- count(N).
N = 2.
```

Solutions

1. *Briefly describe the general approach used to solve the pit-crossing puzzle in the slides.*

   Pick a plank from the supply. See if it can be placed without ending over a pit. If so, solve it from there using the remaining planks. If not, pick a different plank and try again.

2. *Write a predicate* **`inc`** *that uses* **`assert`** *and* **`retract`** *to increment a counter maintained as a* **`count/1`** *fact. It reports the new value.*

   ```
   inc :-
      count(N0),
      retract(count(_)),
      N is N0+1,
      assert(count(N)),
      format('Count is ~w~n', N).
   ```

Last name

____

CSC 372 Mid-term Exam
Thursday, March 10, 2016

READ THIS FIRST

Read this page now but do not turn this page until you are told to do so. Go ahead and fill in your last name in the box above.

This is a 60-minute exam with a total of 100 points of regular questions and an extra credit section.

The last five minutes of the exam is a "seatbelts required" period, to avoid distractions for those who are still working. If you finish before the "seatbelts required" period starts, you may turn in your exam and leave. If not, you must stay quietly seated—no "packing up"— until time is up for all.

You are allowed no reference materials whatsoever.

If you have a question, raise your hand. I will come to you. DO NOT leave your seat.

If you have a question that can be safely resolved with a minor assumption, like the name of a function or the order of function arguments, state the assumption and proceed.

Feel free to use abbreviations, like "`otw`" for "`otherwise`".

It's fine to use helper functions unless a specific form, such as point-free style, is specifically requested.

Don't make problems hard by assuming that they need to do more than is specifically mentioned in the write-up or that the solution that comes to mind is "too easy."

If you're stuck on a problem, please ask for a hint. Try to avoid leaving a problem completely blank—that's a sure zero.

It is better to put forth a solution that violates stated restrictions than to leave it blank—a solution with violations may still be worth partial credit.

When told to begin, double-check that your name is at the top of this page, and then **put your initials in the lower right hand corner of the top side of each sheet, checking to be sure you have all six sheets.**

BE SURE to enter your last name on the sign-out log when turning in your completed exam.

Problem 1: (8 points)

What is the **type** of each of the following values? If the expression is invalid, briefly state why.

Assume numbers have the type `Int`. Remember that `String` is a type synonym for `[Char]`; those two can be used interchangeably.

Important: Remember that the type of a function includes the type of the arguments as well as the type of the value returned.

```
"abc"
```

```
('4', 52, [5,2])
```

```
last
```

```
[[True]]
```

```
filter even
```

```
map head
```

```
[head, head . tail]
```

```
(++"x")
```

Problem 2: (10 points) (two points each)

This problem is like `warmup.hs` on the assignments—write the following Haskell Prelude functions.

Instances of poor style or needlessly using other Prelude or helper functions will result in deductions.

Remember this order of preference for handling cases: patterns, guards, `if-else`. Be sure to use the wildcard pattern (underscore) when appropriate.

There's no need to specify function types.

`snd` (Returns second element of a two-tuple)

`head` (Ignore empty-list case)

`last` (Ignore empty-list case)

`map`

`zipWith` (Reminder: `zipWith (+) [1,2,3] [10,20,30]` is `[11,22,33]`)

DID YOU REMEMBER TO USE WILDCARDS WHEN APPROPRIATE?

Problem 3: (16 points) (8 points each)

For this problem you are to **write both recursive and non-recursive versions** of a function `numlist`, with type `[[Char]] -> IO ()`, that outputs a numbered list of the strings in a list:

```
> numlist ["just", "testing", "this"]
1. just
2. testing
3. this

> numlist []
[no items]
```

Just as you did with `street` and other problems on the Haskell assignments that produced output, build up a string containing newlines and output it with `putStr` as a final step.

Just like on assignment 3, the recursive version must not use any higher-order functions. Just like on assignment 4, write the non-recursive version imagining that you just don't know to how to write a recursive function.

`unlines` and/or `concat` may be useful:

```
> unlines ["a","b","c"]
"a\nb\nc\n"

> concat ["a","bc","d"]
"abcd"
```

Problem 4: (4 points)

Consider the following interaction with `ghci`:

```
> map count [10,20,30]
[1,2,3]

> map count [7,8,9,10]
[1,2,3,4]
```

For this problem you are to either (1) write a function `count` that behaves as shown above or (2) explain why it is impossible to write such a function.

Problem 5: (8 points)

The following function, `revtups`, takes a list of two-tuples and produces a new list with the tuples' elements swapped **and** the order of the tuples reversed:

```
revtups list = foldl ff [] list
```

Usage:

```
> revtups [(1,'a'),(2,'b'),(3,'c')]
[('c',3),('b',2),('a',1)]

> revtups [("one",[1]), ("two",[2])]
[([2],"two"),([1],"one")]
```

For this problem you are to:

1. State the type of `revtups` (1 point)
2. Write a folding function `ff` that will work with `revtups` as shown above.

Problem 6: (12 points)

Write a Haskell function `co chars string`, with type `[Char] -> [Char] -> Int`, that counts how many times the characters in `chars` occur in `string`.

Usage:

```
> co "aeiou" "just a test"
3

> co ['0'..'9'] "12:15pm"
4

> co "xyz" ""
0

> co "" "12:15pm"
0
```

You may use only Prelude functions on this problem but there are no other restrictions.

Remember that the Prelude has a `sum` function: `sum [3,1,5]` produces `9`.

Assume the characters to count are unique; you won't see something like `co "aaa" ...`.

Problem 7: (22 points)

Write a Ruby program `adjcols.rb` that makes specified adjustments to columns of numeric values in a CSV (comma-separated values) file.

Adjustments are specified as command-line arguments. Here's a sample invocation:

```
ruby adjcols.rb 2:+10 3:=7 5:-1 < x.csv
```

Adjustments have the form *COLUMN-NUMBER:CHANGE*. Examples:

`2:+10` means to add `10` to all values in column `2` (column numbers are 1-based and positive)
`3:=7` means to change all values in column `3` to `7`
`5:-1` means to subtract `1` from all values in column `5`

Here's a CSV file:

```
% cat x.csv
name,q1,q2,q3,q4
whm,2,5,20,5
jmc,7,6,18,10
chris,5,0,10,15
```

`adjcols` reads lines from standard input (use `STDIN.gets` to read lines) and writes the lines to standard output. `adjcols` assumes the first line is a header row and outputs it unchanged.

Execution:

```
% ruby adjcols.rb 2:+10 3:=7 5:-1 < x.csv
name,q1,q2,q3,q4
whm,12,7,20,4
jmc,17,7,18,9
chris,15,7,10,14
```

Columns can be specified in any order. Changes are cumulative. An example of both:

```
% ruby adjcols.rb 2:+100 3:=1 2:+1000 < x.csv
name,q1,q2,q3,q4
whm,1102,1,20,5
jmc,1107,1,18,10
chris,1105,1,10,15
```

The CSV file may have any number of lines, and lines may have any number of columns. Assume that adjustments are well-formed and that the specified column is always present and contains an integer value. Any number of adjustments may be present.

Ruby note: `"+3".to_i` produces `3` and `"-5".to_i` produces `-5`.

THERE'S SPACE FOR YOUR SOLUTION ON THE NEXT PAGE

For reference:

```
% ruby adjcols.rb 2:+10 3:=7 5:-1 < x.csv
name,q1,q2,q3,q4
whm,12,7,20,4
jmc,17,7,18,9
chris,15,7,10,14
```

```
% cat x.csv
name,q1,q2,q3,q4
whm,2,5,20,5
jmc,7,6,18,10
chris,5,0,10,15
```

Problem 8: (5 points) (one point each unless otherwise indicated)

The following questions and problems are related to Haskell.

(1) Add parentheses to the following type to explicitly show the associativity of the -> operator.

```
[Int] -> (Int -> Bool) -> [Bool]
```

(2) Fully parenthesize the following expression to show the order of operations.

```
f 2 g 3 + x f g [ a b c * 1 ]
```

(3) Rewrite the following function binding to use point-free style:

```
f1 x = f (g x)
```

(4) Briefly, how does Haskell's pattern matching contribute to the readability of Haskell code?

(5) Once upon a time I had this line of code in a file:

```
f x y = x*3 + y
```

I wanted to produce an error on that line, so I typed some junk into the line:

```
asdsf sdfs 3 3 3 f x y = x*3 + y
```

But, that goofy code loaded without error! Why? (Hint: Be sure your answer is more than something like "It's still valid code.")

Problem 9: (5 points) (one point each unless otherwise indicated)

The following questions and problems are related to Ruby.

(1) Aside from the fact that strings are mutable in Ruby and are immutable in Java, what's an important difference between the string-handling facilities in Ruby and Java?

(2) Imagine that a Ruby class has methods named `mudge` and `mudge!`. What does that imply?

(3) What's a fundamental semantic (i.e., non-syntactic) difference between `if-else` in Ruby and `if-else` in Java?

(4) Aside from syntax, what's a fundamental difference between `:load x.hs` in `ghci` and `load "x.rb"` in `irb`?

(5) What's the value of each of the two following Ruby expressions?

```
"abc"[5] && false

0 || 1
```

Problem 10: (10 points) (one point each unless otherwise indicated)

Briefly answer the following general questions.

(1) Who founded The University of Arizona's Computer Science department and when?

(2) What are two aspects of a "paradigm", as described in Kuhn's *The Structure of Scientific Revolutions*? Here are two **wrong** answers: "functional" and "object-oriented". (Two points!)

(3) What's a fundamental characteristic of a language that uses static typing?

(4) What's a fundamental characteristic of a language that uses dynamic typing?

(5) Show an example of syntactic sugar in any language you know.

(6) Aside from the intrinsic elements of a language, like its design and performance, what are two factors that can influence the popularity of a language?

(7) whm often says "In Haskell we never change anything; we only make new things." What's an example of that?

(8) When a new graduate research assistant wanted to add a bunch of new features to Icon, what did the project's wise leader say?

(9) What's one way in which having a REPL available makes it easier to learn a programming language?

Extra Credit Section (½ point each unless otherwise noted)

(1) whm believes the abbreviation LHtLaL appears nowhere on the web except in his slides. What does it stand for?

(2) To whom does whm attribute the following quote?
"When you hit a problem you can lean forward and type or sit back and think."

(3) Consider this Haskell function: `add x y = x + y`. What is the exact type of `add`?

(4) Double tricky: What is the exact type of the Haskell list `[foldl, foldr]`?

(5) The Haskell expression `[`*`FROM`*`..`*`TO`*`]` produces an empty list if *FROM* > *TO*. Write a non-recursive function `range from to` that's a little smarter: `range 1 5` returns `[1,2,3,4,5]`, and `range 5 1` returns `[5,4,3,2,1]`.

(6) Suppose the second example for problem 4 (page 5) had been this:

```
> map count [6,7,8,9]
[1,2,3,4]
```

Briefly, how would that have changed your answer for that question?

(7) If you only remember one thing about the Haskell segment of 372, what will it be? (Ok to be funny!)

(8) What is "duck typing"?

(9) With Ruby in mind, define the term "iterator".

(10) Name a language other than Icon that was created at The University of Arizona.

CSC 372 Mid-term Exam
Thursday, March 10, 2016
Solutions

Problem 1: (8 points) (mean = 5.6, median = 5.5)

*What is the **type** of each of the following values? If the expression is invalid, briefly state why.*

```
"abc"
      [Char]

('4', 52, [5,2])
      (Char, Int, [Int])

last
      [a] -> a

[[True]]
      [[Bool]]

filter even
      [Int] -> [Int]

map head
      [[a]] -> [a]

[head, head . tail]
      [[a] -> a]

(++"x")
      [Char] -> [Char]
```

Problem 2: (10 points) (two points each) (mean = 8.3, median = 8.5)

```
snd             (Returns second element of a two-tuple)
      snd (_,x) = x

head            (Ignore empty-list case)
      head (h:_) = h

last            (Ignore empty-list case)
      last [x] = x
      last (_:t) = last t

map
      map _ [] = []
      map f (h:t) = f h : map f t

zipWith         (Reminder: zipWith (+) [1,2,3] [10,20,30] is [11,22,33])
      zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
      zipWith _ _ _ = []
```

Note the ordering of the clauses: The first clause handles the case that both lists have at least one element. If that's not true, we're done, and that's handled by the second clause.

Problem 3: (16 points) (8 points each) (recursive: mean = 6.2, median = 7; non-recursive: mean = 5.0, median = 6)

For this problem you are to ***write both recursive and non-recursive versions*** *of a function* `numlist`, *with type* `[[Char]] -> IO ()`, *that outputs a numbered list of the strings in a list:*

```
> numlist ["just", "testing", "this"]
1. just
2. testing
3. this

> numlist []
[no items]
```

Solutions:

```
numlist [] = putStr "[no items]\n"
numlist items = putStr $ unlines $ helper 1 items
    where
        helper _ [] = []
        helper n (item:items) =
            (show n ++ ". " ++ item) : helper (n+1) items


numlist [] = putStr "[no items]\n"
numlist items = putStr $ unlines $ zipWith f [1..] items
    where
        f num s = show num ++ ". " ++ s
```

Problem 4: (4 points) (mean = 3.3, median = 4)

Consider the following interaction with `ghci`:

```
> map count [10,20,30]
[1,2,3]

> map count [7,8,9,10]
[1,2,3,4]
```

For this problem you are to *either* *(1) write a function* `count` *that behaves as shown above or (2) explain why it is impossible to write such a function.*

It's impossible. Functions always produce the same output for a given input but in the first case, `count 10` produces `1` and in the second case, `count 10` produces `4`.

Problem 5: (8 points) (mean = 5.7, median = 7)

The following function, `revtups`, *takes a list of two-tuples and produces a new list with the tuples' elements swapped* ***and*** *the order of the tuples reversed:*

```
revtups list = foldl ff [] list
```

Usage:

```
> revtups [(1,'a'),(2,'b'),(3,'c')]
[('c',3),('b',2),('a',1)]

> revtups [("one",[1]), ("two",[2])]
[([2],"two"),([1],"one")]
```

For this problem you are to:

1. *State the type of* `revtups` *(1 point)*
2. *Write a folding function* `ff` *that will work with* `revtups` *as shown above.*

```
1.     revtups :: [(a,b)] -> [(b,a)]
2.     ff acm (a,b) = (b,a):acm
```

Problem 6: (12 points) (mean = 8.2, median = 10)

Write a Haskell function `co chars string`*, with type* `[Char] -> [Char] -> Int`*, that counts how many times the characters in* `chars` *occur in* `string`*.*

Usage:

```
> co "aeiou" "just a test"
3

> co ['0'..'9'] "12:15pm"
4
...
```

My first solution was this:

```
co chars str = sum $ map (f str) chars
    where
       f str c = length $ filter (\e -> e == c) str
```

Of the non-recursive solutions I saw, I felt that Ms. McCabe's was the best:

```
co chars string = length $ filter (\x -> x `elem` chars) string
```

Several students wrote a non-recursive solution that was something like this:

```
co chars (c:cs)
    | c `elem` chars = 1 + co chars cs
    | otherwise = co chars cs
co _ _ = 0
```

Problem 7: (22 points) (mean = 14.4, median = 18)

Write a Ruby program `adjcols.rb` *that makes specified adjustments to columns of numeric values in a CSV (comma-separated values) file.*

Adjustments are specified as command-line arguments. Here's a sample invocation:

```
ruby adjcols.rb 2:+10 3:=7 5:-1 < x.csv
```

Adjustments have the form `COLUMN-NUMBER:CHANGE`*. Examples:*

2:+10	*means to add 10 to all values in column 2 (<u>column numbers are 1-based and positive</u>)*
3:=7	*means to change all values in column 3 to 7*
5:-1	*means to subtract 1 from all values in column 5*

This is my solution:

```
puts STDIN.gets

while line = STDIN.gets
```

```
    parts = line.split ","

    for adj in ARGV
        col, delta = adj.split ":"
        col = col.to_i - 1
        if delta[0] == "="
            newval = delta[1..-1].to_i
        else
            newval = parts[col].to_i + delta.to_i
        end
        parts[col] = newval.to_s
    end

    puts parts * ","

end
```

I'd hoped that it would be quickly observed that lines could processed one at a time, but a number of students read all the lines into an array and then iterated over the elements of that array. An attribute that I've found to be common among many long and/or wrong solutions on assignments and exams is this: creation of unnecessary data structures. It takes code to build those unnecessary structures and then code to get data out of those structures. For this problem it also turns a program that could process a file of unlimited size to one that's limited by available memory.

A minor thing that made me groan during grading was the use of a flag to indicate whether the input line at hand was the first line. It seems far simpler to just read and write that first line.

Problem 8: (5 points) (one point each unless otherwise indicated) (mean = 2.9, median = 3)

The following questions and problems are related to Haskell.

(1) *Add parentheses to the following type to explicitly show the associativity of the* `->` *operator.*

```
[Int] -> (Int -> Bool) -> [Bool]

[Int] -> ((Int -> Bool) -> [Bool])
```

(2) *Fully parenthesize the following expression to show the order of operations.*

```
f  2  g  3  +  x  f  g  [ a  b  c  *  1 ]

(((f  2) g) 3)  +  (((x f) g) [(((a  b) c) * 1)])
```

(3) *Rewrite the following function binding to use point-free style:*

```
f1 x = f (g x)

f1 = f . g
```

(4) *Briefly, how does Haskell's pattern matching contribute to the readability of Haskell code?*

Patterns let us bind names to various elements of data structures, so that instead of writing code that digs into data structures, like

```
f x = g (head x) ++ h (tail x)
```

we can write

```
f (h:t) = g h ++ h t
```

(5) *Once upon a time I had this line of code in a file:*

```
f x y = x*3 + y
```

I wanted to produce an error on that line, so I typed some junk into the line:

```
asdsf sdfs 3 3 3 f x y = x*3 + y
```

But, that goofy code loaded without error! Why? (Hint: Be sure your answer is more than something like "It's still valid code.")

The junk I typed turned the two-argument function `f` into a seven-argument function named `asdsf`. Three of the arguments are literal pattern matches for the number 3.

Problem 9: (5 points) (one point each unless otherwise indicated) (mean = 2.3, median = 2)

The following questions and problems are related to Ruby.

(1) *Aside from the fact that strings are mutable in Ruby and are immutable in Java, what's an important difference between the string-handling facilities in Ruby and Java?*

Ruby lets us specify portions of strings with `s[n]`, `s[start,length]`, and `s[Range]`. Additionally, those expressions can be targets of an assignment. Many other answers are valid, too.

(2) *Imagine that a Ruby class has methods named `mudge` and `mudge!`. What does that imply?*

`mudge` has both an applicative and an imperative form. `mudge`, the applicative form returns a result but doesn't modify its receiver. `mudge!`, the imperative form modifies its receiver.

(3) *What's a fundamental semantic (i.e., non-syntactic) difference between `if-else` in Ruby and `if-else` in Java?*

`if-else` is a statement in Java but in Ruby it's an expression. In Ruby I can say something like `val = if a > b then c end`.

(4) *Aside from syntax, what's a fundamental difference between `:load x.hs` in `ghci` and `load "x.rb"` in `irb`?*

`:load` is implemented by the REPL that `ghci` provides; it's not Haskell code. Ruby's `load` is a method; `load "x.rb"` is a Ruby expression that has a side effect of loading the code in `x.rb`.

(5) *What's the value of each of the two following Ruby expressions?*

```
"abc"[5] && false
     nil

0 || 1
     0
```

I was surprised by the number of students that missed this one.

Problem 10: (10 points) (one point each unless otherwise indicated) (mean = 6.1, median = 7)

Briefly answer the following general questions.

(1) *Who founded The University of Arizona's Computer Science department and when?*
Ralph Griswold, in 1971.

(2) *What are two aspects of a "paradigm", as described in Kuhn's The Structure of Scientific Revolutions? Here are two **wrong** answers: "functional" and "object-oriented". (Two points!)*
See Haskell slide 4.

(3) *What's a fundamental characteristic of a language that uses static typing?*

Type errors can be detected without running any code or knowing what specific input values are.

(4) *What's a fundamental characteristic of a language that uses dynamic typing?*

In the general case, type errors can't be detected until code is run.

(5) *Show an example of syntactic sugar in any language you know.*

In Haskell, `"abc"` is syntatic sugar for `['a', 'b', 'c']`.

(6) *Aside from the intrinsic elements of a language, like its design and performance, what are two factors that can influence the popularity of a language?*

See intro slide 34.

(7) *whm often says "In Haskell we never change anything; we only make new things." What's an example of that?*

If the first element of a list of integers was to be changed to zero, we'd do that by making a new list whose head is zero and whose tail is the of the list at hand.

(8) *When a new graduate research assistant wanted to add a bunch of new features to Icon, what did the project's wise leader say?*

Ralph said I could all the features I wanted but for every feature I wanted to add, I first had to find a feature to remove.

(9) *What's one way in which having a REPL available makes it easier to learn a programming language?*

It makes it easy to experiment with things.

Extra Credit Section (½ point each unless otherwise noted) (mean = 1.9, median = 2)

(1) *whm believes the abbreviation LHtLaL appears nowhere on the web except in his slides. What does it stand for?*

Learning How to Learn a Language

(2) *To whom does whm attribute the following quote?*

"When you hit a problem you can lean forward and type or sit back and think."

Dr. Proebsting

(3) *Consider this Haskell function: `add x y = x + y`. What is the exact type of `add`?*

```
Num a => a -> a -> a
```

(4) *Double tricky: What is the exact type of the Haskell list `[foldl, foldr]`?*

My first thought was that it would be a type error because the functions don't have the same type but when I saw that it worked, I realized that the types of the two can be unified:

```
[(b -> b -> b) -> b -> [b] -> b]
```

(5) *The Haskell expression `[FROM..TO]`produces an empty list if `FROM` > `TO`. Write a non-recursive function `range from to`that's a little smarter: `range 1 5` returns `[1,2,3,4,5]`, and `range 5 1` returns `[5,4,3,2,1]`.*

```
range from to
  | from <= to = [from..to]
  | otherwise = reverse [to..from]
```

(6) *Suppose the second example for problem 4 (page 5) had been this:*

```
> map count [6,7,8,9]
[1,2,3,4]
```

Briefly, how would that have changed your answer for that question?

It would now be possible to implement `count`.

Note: This question was ill-conceived—virtually any answer could be argued as being correct but it was only marked as correct if you said that `count` could now be implemented. Consider a half-point of the 8-point adjustment as being compensation for any injustice on this question.

(7) If you only remember one thing about the Haskell segment of 372, what will it be? (Ok to be funny!)
Presented anonymously and in random order, here's what was said:

Haskell has no debugger, and thus sucks.
functional programming can be very useful
That recursion is occasionally ok
maps & folds are useful!
Partial application
The power of cons.
I need to watch my types...
unhelpful error messages :(
It was (fun)ctional!
recursion is easier with patterns
Anonymous functions are beautiful.
The Friday Night Club isn't as fun as it sounds
Don't over think it!
I'm not good at it.
Patterns.
pancakes on the stack
Haskell functions seem to be redundant to death (having to write the function name again for every possibility of input)
I can't eat pancakes any more
putting the fun in FUNctional programming
So much recursion
mapping
You never change anything. Only make new things.
Haskell is short for HaskHell
higher order functions
expected [Char], was instead [[Char]]
A whole lot of time spent to find a solution with very little code
Iteration is for the weak!
Recursion isn't complete hell
Pancake
Crying tears of blood when forgetting to add a base case in recursion problems
Type error everywhere
functions are values
higher order function
Matt Gautreau was right. Don't tell him I said that.
Patterns over guards over if-else.
Pancakes Toyota
There's probably a Prelude function to do something for you if you look hard enough
f this f that f it
variable state, why don't you change!
maps are crazy useful
Recursion is now my best friend haha
Know your recursion!
anonymous functions
recursion!

(8) What is "duck typing"?

A style of programming/mindset for typing where we're only concerned about the operations provided by objects rather than their types.

(9) *With Ruby in mind, define the term "iterator".*

An iterator is a method that can invoke a block.

(10) *Name a language other than Icon that was created at The University of Arizona.*

See intro slide 38.

Statistics

Here are all scores:

```
98.5, 95.5, 94.5, 94, 94, 93, 92.5, 91, 91, 90, 89.5, 89.5, 88, 87.5, 87.5,
84.5, 84, 82.5, 81.5, 81, 81, 79.5, 79.5, 78, 77.5, 76, 75, 74.5, 73.5, 73,
71.5, 71.5, 69.5, 68.5, 68, 67.5, 63, 61.5, 61, 60.5, 60, 59, 58.5, 58.5, 58.5,
53.5, 53.5, 53, 53, 49.5, 48.5, 48, 47.5, 46, 40.5, 37, 34.5, 22.5, 18.5

n = 59
mean = 69.8
median = 73
```

Here's a histogram of all scores. The five-point wide bins cover the interval `[X-2.5,X+2.5)`, where `X` is the label for a bin.

```
+----------------------------------------------------------------------+
|                                                                      |
|                                  *                       *           |
|                                  *               *       *           |
|                                  *       *       *       *   *       |
|                                  *       *   *   *       *   *       |
|                          *   *   *       *   *   *       *   *       |
|                          *   *   *       *   *   *   *   *   *       |
|              *           *   *   *       *   *   *   *   *   *       |
|  *   *       *   *   *   *   *   *   *   *   *   *   *   *   *   *   |
|  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100  |
|                                                                      |
+----------------------------------------------------------------------+
```

The mean and median were lower than I expected, so eight points were added to all scores before loading them on D2L.

Last name

____

CSC 372 Final Exam
Monday, May 9, 2016

READ THIS FIRST

Read this page now but do not turn this page until you are told to do so. Go ahead and fill in your last name in the box above.

This is a 100-minute exam with a total of 100 points of regular questions and an extra credit section.

The last five minutes of the exam is a "seatbelts required" period to avoid distractions for those who are still working. If you finish before the "seatbelts required" period starts, you may turn in your exam and leave. If not, you must stay quietly seated—no "packing up"— until time is up for all.

You are allowed no reference materials whatsoever.

If you have a question, raise your hand. One of us will come to you. DO NOT leave your seat.

If you have a question that can be safely resolved with a minor assumption, like the name of a function or the order of function arguments, state the assumption and proceed.

Don't make problems hard by assuming that they need to do more than is specifically mentioned in the write-up or that the solution that comes to mind is "too easy."

If you're stuck on a problem, please ask for a hint. Try to avoid leaving a problem completely blank—that's a sure zero.

It is better to put forth a solution that violates stated restrictions than to leave it blank—a solution with violations may still be worth partial credit.

When told to begin, double-check that your name is at the top of this page, and then **put your initials in the lower right hand corner of the top side of each sheet, checking to be sure your copy of the exam has all the pages.**

BE SURE to enter your last name on the sign-out log when turning in your completed exam.

I believe that at least one student will be taking a make-up exam, so I won't be posting solutions on Piazza. I'll instead park a copy in this directory: `http://cs.arizona.edu/~whm/more-pancakes-please/`
Write it down! Note the ~ in `~whm`.

Problem 1: (6 points)

Cite three things about programming languages you learned by watching your classmates' video projects. Each of the three should be about a different language and have a bit of depth, as described in the Piazza post that announced this problem.

Problem 2: (1 point)

Write a Haskell function `doflip pancakes flip` that performs an `fsort`-style flip. Examples:

```
> doflip [3,5,4,2] 2
[5,3,4,2]

> doflip it 4
[2,4,3,5]
```

Assume the stack has at least one pancake and that `flip` is in range.

Problem 3: (3 points)

Write a **recursive** Haskell function `totlen list` that returns the total length of the strings in `list`.

```
> totlen ["just", "a", "test"]
9

> totlen []
0
```

Problem 4: (4 points)

Write a Haskell function `xout string` that replaces every letter (a-z) in `string` with an "x" of the same case. Non-letters are unchanged. Example:

```
> xout "Don't get stuck!  Keep moving!"
"Xxx'x xxx xxxxx!  Xxxx xxxxxx!"

> xout ""
""
```

Recall `isUpper` and `isLower` in `Data.Char`:

```
> isUpper 'A'
True

> map isLower "A t!"
[False,False,True,False]
```

You may assume `Data.Char` has been imported. It also has `isLetter`.

Problem 5: (2 points)

Write a folding function `f` such that the `foldr` call below behaves as shown, returning a list of the odd numbers in its last argument.

```
> foldr f [] [5,2,9,4,4,3,1]
[5,9,3,1]
```

Hint: The operation being performed is the same as `filter odd [5,2,9,4,4,3,1]` but you definitely don't want to use `filter` in your folding function!

Problem 6: (5 points)

Write a Ruby **iterator** `sbl(a, max)` ("strings by length") that first yields each one-character string in the array `a`. It then yields each two-character string in `a`. The process continues up through strings of length `max`. Strings are yielded in the order they appear in `a`. `sbl` always returns `nil`.

Assume that the array a contains only strings.

Restriction: You may not use `sort`. (If you did, you'd probably find that the sequence produced wouldn't be fully correct.)

The second example below demonstrates that a `max` of 2 limits results to one- and two-character strings.

```
>> sbl(["ab","b","a","aaa","test","bc","a"],10) { |s| p s }
"b"
"a"
"a"
"ab"
"bc"
"aaa"
"test"
=> nil

>> sbl(["ab","b","a","aaa","test","bc","a"],2) { |s| puts s.size }
1
1
1
2
2
=> nil
```

Remember: Don't use a `sort` method.

Problem 7: (4 points)

Write a Ruby method `pages_re` that returns a regular expression that matches a string iff the string consists of one or more comma-separated page specifications. A page specification is one of these three:

- A number, such as "`12`".
- Two numbers separated by a dash, such as "`1-3`".
- A number followed by a dash, such as "`2500-`".

A "number" is a sequence of one or more digits.

Examples:

```
>> pages_re =~ "1-10,15,20-,75"
=> 0

>> pages_re =~ "1,2,"
=> nil                (Has trailing comma)

>> pages_re =~ "5,ten"
=> nil                ("ten" is invalid)

>> pages_re =~ "Pages: 5,7-9"
=> nil                (Has "Pages: " at the beginning)
```

Problem 8: (11 points)

Write a Ruby program `mostfreq.rb` that reads lines on standard input and writes out each line followed by an annotation that shows which non-space character appears most frequently on the line, and how many times it appears. Input lines are right-padded with spaces to a length of 30. (Assume no line is longer than 30.)

```
% cal | ruby mostfreq.rb
      May 2016                 ('6': 1)
Su Mo Tu We Th Fr Sa           ('S': 2)
 1  2  3  4  5  6  7           ('7': 1)
 8  9 10 11 12 13 14           ('1': 6)
15 16 17 18 19 20 21           ('1': 6)
22 23 24 25 26 27 28           ('2': 8)
29 30 31                       ('3': 2)

%
```

<u>Important:</u> Lines with no non-space characters, such as the last line of `cal` output above, have no annotation but are still padded to a length of 30.

<u>In case of a tie, pick any one of the tying characters.</u> (Note that the first line above has a six-way tie, for example.)

You may imagine that `Hash` has a `sort_by_value` method:

```
>> h = {"x" => 5, "y" => 10, "z" => 3}

>> h.sort_by_value
=> [["z", 3], ["x", 5], ["y", 10]]
```

Recall that `String#ljust` can be used to pad a string with blanks on the right:

```
>> "ab".ljust 5
=> "ab   "
```

Write your solution here, or on the next page.

(space for mostfreq.rb solution)

For reference:

```
% cal | ruby mostfreq.rb
      May 2016                    ('6': 1)
Su Mo Tu We Th Fr Sa              ('S': 2)
 1  2  3  4  5  6  7              ('7': 1)
 8  9 10 11 12 13 14              ('1': 6)
...

%

>> {"x" => 5, "y" => 10, "z" => 3}.sort_by_value
=> [["z", 3], ["x", 5], ["y", 10]]
```

Problem 9: (8 points)

In this problem you are to implement a Ruby class named `Seq` that represents a sequence of values with a maximum length. The maximum length is specified when an instance is created.

Values are added to the end of the sequence using an overloaded << operator. If a value is added to a sequence that is already at maximum length, the first value in the sequence is discarded. The << operation returns `nil`.

`Seq#inspect` returns a string consisting of the values in the sequence separated by dashes. Vertical bars surround the full sequence.

Here's an example of interaction. Remember that `irb` uses `inspect` to display the result of each expression.

```
>> s = Seq.new 3
=> ||

>> s << 10
=> |10|

>> s << 20; s << 30
=> |10-20-30|

>> s << 40
=> |20-30-40|

>> s2 = Seq.new 10
=> ||

>> "this is a test of Seq".each_char {|c| s2 << c }
=> "this is a test of Seq"

>> s2
=> |e-s-t- -o-f- -S-e-q|
```

Implementation note: `Array#shift` discards the first element of an array.

Write your solution here, or on the next page.

(space for `Seq` solution)

```
>> s = Seq.new 3
=> ||

>> s << 10
=> |10|

>> s << 20; s << 30
=> |10-20-30|

>> s << 40
=> |20-30-40|

>> s2 = Seq.new 10
=> ||
```

Problem 10: (6 points)

Write the following simple Prolog predicates. There will be a half-point deduction for each occurrence of a singleton variable or failing to take full advantage of unification.

There are no restrictions.

(a) `fl_same(?L)` expresses the relationship that the first and last elements of `L` are identical. It should able to handle all possible combinations of instantiated and uninstantiated variables. `fl_same` fails if the list is empty. Examples:

```
?- fl_same([a,b,a]).
true.

?- fl_same(L).
L = [_G1191] ;
L = [_G1191, _G1191] ;
L = [_G1191, _G1194, _G1191] ;
L = [_G1191, _G1194, _G1197, _G1191] ;
...

?- fl_same([X,Y,5]).
X = 5.
```

(b) `revgen(+L,-X)` generates the elements of the list `L` in reverse order.

```
?- revgen([a,b,c,d], X).
X = d ;
X = c ;
X = b ;
X = a.
```

(c) Write the library predicate `member/2`. Examples:

```
?- member(X,[1,2]).
X = 1 ;
X = 2.

?- member(3,[1,2]).
false.
```

Problem 11: (6 points)

In this problem you are to write a Prolog predicate that is similar to the earlier Ruby problem "`sbl`" (strings by length).

`abl(+Atoms,+Max,-A)` first instantiates `A` to each one-character atom in `Atoms`, then each two-character atom in `Atoms`, etc. `abl` continues for atoms of lengths up through `Max`, if any are that long.

```
?- abl([abc,b,ab,zzz,a,zzzz,aa,c],10,A).
A = b ;
A = a ;
A = c ;
A = ab ;
A = aa ;
A = abc ;
A = zzz ;
A = zzzz ;
false.

?- abl([abc,b,ab,zzz,a,zzzz,aa,c],1,A).
A = b ;
A = a ;
A = c.
```

Similar to `sbl`'s restriction, you may not use any built-in sorting predicate, like `msort`.

Recall `atom_chars(+Atom, ?List_of_chars)`:

```
?- atom_chars(test,L).
L = [t, e, s, t].
```

Problem 12: (8 points)

Write a Prolog predicate `findrun(+L,+N,+X,-Pos)` that looks for `N`-long runs of `X` in `L`, instantiating `Pos` to the zero-based starting position of each run. Assume `N` is greater than zero.

```
?- findrun([a,b,b,c,d,b,b],2,b,Pos).
Pos = 1 ;
Pos = 5 ;
false.

?- findall(Pos,findrun([a,a,a,a,a,a],3,a,Pos),Positions).
Positions = [0, 1, 2, 3].

?- findall(Pos,findrun([a,b,a,a,b],1,a,Pos),Positions).
Positions = [0, 2, 3].
```

You may assume you have the `repl(+N,+Elem,-List)` predicate you wrote on assignment 9:

```
?- repl(a,5,L).
L = [a, a, a, a, a].
```

Please ask for a hint if you have trouble with this one!

Problem 13: (7 points)

Write a predicate `expand(+List,-Expanded)` that takes a list of (1) atoms and (2) structures of the form *`Atom*N`* and produces an "expanded" list. Assume the `N` terms are `>= 0`.

```
?- expand([a*2, b, test*3, none*0, here], L).
L = [a, a, b, test, test, test, here].

?- expand([],L).
L = [].
```

Problem 14: (11 points)

Write a Prolog predicate `reach(+Destination, +Jumps, +Fences, -Solution)` that finds a sequence of "jumps" on a Cartesian plane from `(0,0)` to the `Destination` (a `pos/2` structure), being sure that no jump lands on a fence.

`Fences` is a list of `fence` structures, like `[fence(x,3),fence(y,1)]`. The first element indicates there is a fence at `x = 3`. The second indicates there is a fence at `y = 1`. There may be any number of fences. Here's a representation of those two fences, and a destination of `pos(5,3)`, marked as `D`.

```
4 |           +
  |           +
3 |           +       D
  |           +
2 |           +
  |           +
1 |+++++++++++++++++++++++
  |           +
  +-----------------------
      1   2   3   4   5
```

`Jumps` is a list of two-term `jump` structures that specify `x` and `y` distance, like `[jump(0,3),jump(1,1),jump(-3,2)]`. There may be any number of jumps.

The final jump must land exactly on the destination. Each jump can be used only once.

Here's a call of `reach` that produces two solutions:

```
?- reach(pos(5,3), [jump(1,1),jump(3,0),jump(1,2),jump(2,2)],
   [fence(x,3),fence(y,1)], Sol).  % Note: query is wrapped around
Sol = [jump(1, 2), jump(1, 1), jump(3, 0)] ;
Sol = [jump(1, 2), jump(3, 0), jump(1, 1)] ;
false.
```

Note that from `pos(0,0)`, doing `jump(1,1)` would land on the fence at `y = 1`.

Similarly, from `pos(0,0)`, doing `jump(2,2)` then `jump(1,1)` would land on the fence at `x = 3`.

Recall how `select/3` was used in the pit crossing example: `select(Plank, Planks, Remaining)`.

Note that the terms in a `jump` structure can be accessed by unifying it with a `jump` structure with uninstantiated variables:

```
?- Jumps = [jump(3,4)], Jumps = [H|_], H = jump(X,Y).
Jumps = [jump(3, 4)],
H = jump(3, 4),
X = 3,
Y = 4.
```

`reach` produces all possible solutions in turn. (Just don't use any cuts and things should be fine!)

There's space for your solution on the next page.

(space for solution for `reach`)

For reference:

```
4 |           +
  |           +
3 |           +       D
  |           +
2 |           +
  |           +
1 |++++++++++++++++++++++++
  |           +
  +------------------------
      1   2   3   4   5
```

<u>The final jump must land exactly on the destination.</u> <u>Each jump can be used only once.</u>

```
?- reach(pos(5,3), [jump(1,1),jump(3,0),jump(1,2),jump(2,2)],
   [fence(x,3),fence(y,1)], Sol).  % Note: query is wrapped around
Sol = [jump(1, 2), jump(1, 1), jump(3, 0)] ;
Sol = [jump(1, 2), jump(3, 0), jump(1, 1)] ;
false.
```

Problem 15: (4 points) (one point each)

The following questions and problems are related to Prolog.

(1) When writing documentation for Prolog predicates, arguments are often prefixed with the symbols `+`, `-`, and `?`, such as `p(+X,?Y,-Z).` What does each of those three symbols mean?

(2) Write a sentence that expresses the relationship between the terms "fact", "clause", and "rule".

(3) Consider the following goal written by a novice Prolog programmer:

```
X is X + length(List)
```

What are **two** misunderstandings evidenced by that goal?

(4) What's the most important fundamental difference between a Prolog predicate like `append/3` and a function/method of the same name in Haskell, Ruby, or Java?

Problem 16: (7 points) (one point each unless otherwise indicated)

Below is a transcript of interaction with Standard ML of New Jersey, a functional programming language. Annotate the transcript with seven significant observations about Standard ML. Excellent or additional observations may earn up to a total of three points of extra credit. Here's an example of an insignificant observation: "There's a `map` function."

```
$ sml
Standard ML of New Jersey v110.69 [built: Mon Jun  8 23:24:21 2009]

- 5 + ~7;
val it = ~2 : int


- (1, 2.0, "three");
val it = (1,2.0,"three") : int * real * string


- explode "test";
val it = [#"t",#"e",#"s",#"t"] : char list


- implode it;
val it = "test" : string


- map (fn(n) => n * 3) [3, 1, 5, 9];  {- See above re map -}
val it = [9,3,15,27] : int list



- it::[];
val it = [(1,2.0,"three")] : (int * real * string) list



- fun f1 a b = [a,b];
val f1 = fn : 'a -> 'a -> 'a list



- fun f2 a b = [a = b];
val f2 = fn : ''a -> ''a -> bool list



- 3 + 4.5;
stdIn:1.1-1.6 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real

- (hd [1,2,3], tl [1,2,3]);
val it = (1,[2,3]) : int * int list
```

Problem 17: (7 points) (one point each unless otherwise indicated)

Answer the following general questions.

(1) What's something significant related to programming languages that you remember from the JarWars video we viewed and discussed during the second-to-last class?

(2) Ralph Griswold said, "If you're going to invent a language, be sure to invent a language that you ________________________________."

(3) What is the fundamental characteristic of a dynamically typed language?

(4) How does Icon avoid the "to versus through" problem that plagues string indexing in many languages and libraries?

(5) What's something significant about Icon you recall from the Icon by Observation exercise during the last class? (Hint: Don't cite your answer for the previous question!)

(6) With programming languages in general, what's the fundamental difference between a statement and an expression?

(7) Which of the three languages we covered this semester are you most glad we covered, and why?

Extra Credit Section (½ point each unless otherwise noted)

(1) In as few words as possible, what is "The Cathedral and the Bazaar"? (Mentioned in `a7` solutions.)

(2) In what month of the year 1891 were classes first held at The University of Arizona?

(3) With Prolog in mind, why is "camelcase variable" an oxymoron?

(4) The technology known as Leda was mentioned on Piazza. What is Leda?

(5) Draw an appropriate avatar for any one of the three languages we covered.

(6) Name a language that was once studied in depth in 372 but isn't any more.

(7) Who do you think whm will vote for in November's presidential election?

(8) Why did whm's solution for `pipes.pl` have the following line? `do(p) :- do(pipes).`

(9) (1 point) `mostfreq.rb` says to assume that `Hash` has a `sort_by_value` method. Write Ruby code that makes that be true.

(10) What did Knuth say about premature optimization?

(11) (1 point) Using only `append` and `length`, and no recursion, write a Prolog predicate `longer(A,B)`, which is true iff list `A` is longer than list `B`.

(12) (1 point) In SNOBOL4, how do you indicate where control should go if a statement fails?

(13) whm hates writing up Piazza posts with suggested readings! They usually don't align well with his slides and he wonders if anybody actually does any of the readings. Over the course of the full semester, how many hours do you estimate you spent doing the suggested readings? (This is just data collection, no right/wrong.)

(14) Write a joke about programming languages.

(15) Finish this sentence: "If I only remember one thing about 372 it will be ...".

CSC 372 Final Exam
Monday, May 9, 2016
Solutions

Problem 1: (6 points) (mean: 5.6, median: 6)

Cite three things about programming languages you learned by watching your classmates' video projects. Each of the three should be about a different language and have a bit of depth, as described in the Piazza post that announced this problem.

When grading I tallied the video topics that were cited and came up with the counts below, with a cut-off of five. In some cases I lumped together all topics for a language into a single count.

Magic Methods in Python	20
Ruby's method_missing	17
PHP	17
JavaScript	10
Character sets in Icon	9
R	9
Swift	9
Lambdas in Java 8	8
Threads in Ruby	8
Functors in Haskell	7
Arrays in Go	6
Randomness in Haskell	6
List comprehensions in Python	6
Hotswapping in Java	5

Videos excelled in different dimensions but if I had to pick an overall winner, it would be K. Taylor's *APL - The Implications of Unicode*.

Here are some others that I felt were particularly outstanding for one reason or another. Authors are not cited because I promised the potential of anonymity.

Closures in JavaScript

Introduction to Common Lisp: S-Expressions and `list/quote/eval`

Haskell Monads in 8 Minutes

Control Structures in bash

Magic Methods in Python

Introduction to Monoids

An Exploration of Channels in Go

A Little on Functors

"Globals" in MUMPS

Python Asterisk(*) usage

Why Nobody Uses Go Arrays

Problem 2: (1 point) (mean: 0.3, median: 0)

Write a Haskell function `doflip pancakes flip` that performs an `fsort`-style flip.

```
doflip pancakes n = reverse (take n pancakes) ++ (drop n pancakes)
```

I intended this one to be a dead-simple list-manipulation problem that leveraged a concept from `fsort` but only about a third of you attempted it. The rest perhaps instinctively recoiled as soon as they saw "pancakes".

Problem 3: (3 points) (mean: 2.3, median: 3)

*Write a **recursive** Haskell function `totlen list` that returns the total length of the strings in `list`.*

```
totlen [] = 0
totlen (h:t) = length h + totlen t
```

Problem 4: (4 points) (mean: 2.7 median: 4)

Write a Haskell function `xout string` that replaces every letter (a-z) in `string` with an "x" of the same case. Non-letters are unchanged.

```
xout = map f
    where
        f c
            | isLower c = 'x'
            | isUpper c = 'X'
            | otherwise = c
```

Problem 5: (2 points) (mean: 1, median: 0.75)

Write a folding function `f` such that the `foldr` call below behaves as shown, returning a list of the odd numbers in its last argument.

```
> foldr f [] [5,2,9,4,4,3,1]
[5,9,3,1]

f elem acm
  | odd elem = elem:acm
  | otherwise = acm
```

Problem 6: (5 points) (mean: 4.6, median: 5)

Write a Ruby **iterator** `sbl(a, max)` ("strings by length") that first yields each one-character string in the array `a`. It then yields each two-character string in `a`. The process continues up through strings of length `max`. Strings are yielded in the order they appear in `a`. `sbl` always returns `nil`.

```
def sbl(a,max)
    for i in 1..max
        for s in a
            if s.size == i
                yield s
            end
        end
    end
    nil
end
```

Problem 7: (4 points) (mean: 2.9, median: 3)

Write a Ruby method `pages_re` that returns a regular expression that matches a string iff the string consists of one or more comma-separated page specifications. A page specification is one of these three:

- *A number, such as `"12"`.*
- *Two numbers separated by a dash, such as `"1-3"`.*
- *A number followed by a dash, such as `"2500-"`.*

```
def pages_re
    num = /\d+|\d+-\d*/
    /^#{num}(,#{num})*$/
end
```

Problem 8: (11 points) (mean: 8.8, median: 10)

Write a Ruby program `mostfreq.rb` that reads lines on standard input and writes out each line followed by an annotation that shows which non-space character appears most frequently on the line, and how many times it appears.

```
def main
    while line = gets
        line.chomp!
        puts "#{line.ljust(30)}#{count line}"
    end
end

def count s
    counts = Hash.new(0)

    s.each_char {|c| counts[c] += 1 if c != " "}
    first = counts.sort_by_value[-1]

    if first
        " ('#{first[0]}': #{first[1]})"
    else
        ""
    end
end

main
```

Problem 9: ***(8 points)*** (mean: 5.6, median: 6.5)

In this problem you are to implement a Ruby class named `Seq` *that represents a sequence of values with a maximum length.*

```
    class Seq
        def initialize n
            @n = n
            @vals = []
        end

        def << rhs
            @vals << rhs
            if @vals.size > @n
                @vals.shift
            end
            self
        end

        def inspect
            "|" + @vals * "-" + "|"
        end
    end
```

Problem 10: (6 points) (mean: 4.1, median: 5)

Write the following simple Prolog predicates. There will be a half-point deduction for each occurrence of a singleton variable or failing to take full advantage of unification.

(a) `fl_same(?L)` *expresses the relationship that the first and last elements of* `L` *are identical. It should able to handle all possible combinations of instantiated and uninstantiated variables.* `fl_same` *fails if the list is empty. Examples:*

```
fl_same(L) :- L = [H|_], last(L,H).
```

(b) `revgen(+L,-X)` *generates the elements of the list* `L` *in reverse order.*

```
revgen(L,X) :- reverse(L,RL), member(X,RL).
```

(c) Write the library predicate `member/2`. Examples:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Problem 11: ***(6 points)*** (mean: 3.2, median: 3)

`abl(+Atoms,+Max,-A)` *first instantiates* `A` *to each one-character atom in* `Atoms`*, then each two-character atom in* `Atoms`*, etc.* `abl` *continues for atoms of lengths up through* `Max`*, if any are that long.*

```
    abl(Atoms,Max,A) :- between(1,Max,N), member(A,Atoms),
        atom_chars(A,Chars), length(Chars,N).
```

Problem 12: (8 points) (mean: 4.1, median: 5)

Write a Prolog predicate `findrun(+L,+N,+X,-Pos)` that looks for `N`-long runs of `X` in `L`, instantiating `Pos` to the zero-based starting position of each run. Assume `N` is greater than zero.

```
findrun(L,N,X,Pos) :-
  repl(X,N,Mid), append(Pre,Rest,L), append(Mid,_,Rest),
length(Pre,Pos).
```

Problem 13: (7 points) (mean: 5.1, median: 5.5)

Write a predicate `expand(+List,-Expanded)` that takes a list of (1) atoms and (2) structures of the form `Atom*N` and produces an "expanded" list. Assume the `N` terms are >= `0`.

```
expand([],[]).
expand([Atom*N|T],R)
     :- repl(Atom,N,Rep), expand(T,More), append(Rep,More,R), !.
expand([H|T],[H|R]) :- expand(T,R).
```

Problem 14: (11 points) (mean: 6.6, median: 8)

Write a Prolog predicate `reach(+Destination, +Jumps, +Fences, -Solution)` that finds a sequence of "jumps" on a Cartesian plane from `(0,0)` to the `Destination` (a pos/2 structure), being sure that no jump lands on a fence.

```
reach(Dest, Jumps, Fences, Solution) :-
     reach0(Dest, Jumps, Fences, pos(0,0), Solution).

reach0(pos(X,Y),_,_,pos(X,Y),[]).

reach0(Dest, Jumps, Fences, pos(CX,CY), [Jump|MoreJumps]) :-
    select(Jump,Jumps,Remaining),
     Jump = jump(X,Y),
     NX is CX+X, NY is CY+Y,
     \+on_fence(pos(NX, NY), Fences),
     reach0(Dest, Remaining, Fences, pos(NX,NY), MoreJumps).

on_fence(pos(X,Y),Fences) :-
   (member(fence(x,X),Fences);member(fence(y,Y),Fences)).
```

Problem 15: (4 points) *(one point each)* (mean: 2.5, median: 2.5)

<u>The following questions and problems are related to Prolog.</u>

(1) *When writing documentation for Prolog predicates, arguments are often prefixed with the symbols +, −, and ?, such as `p(+X,?Y,-Z)`. What does each of those three symbols mean?*

plus: should be instantiated
minus: will be instantiated
question mark: either instantiated or uninstantiated

(2) *Write a sentence that expresses the relationship between the terms "fact", "clause", and "rule".*

Facts and rules are clauses.

(3) *Consider the following goal written by a novice Prolog programmer:*

```
X is X + length(List)
```

What are ***two*** *misunderstandings evidenced by that goal?*

(1) Seems to be treating is as assignment, not unification. `X is X + anything` won't succeed unless anything is zero.

(2) Seems to be thinking of `length` not as a predicate but an arithmetic function that `is/2` knows about.

(4) *What's the most important fundamental difference between a Prolog predicate like `append/3` and a function/method of the same name in Haskell, Ruby, or Java?*

Prolog's `append/3` can be used to perform many operations, like taking a list apart.

Problem 16: (7 points) (mean: 6.2, median: 7)

Below is a transcript of interaction with Standard ML of New Jersey, a functional programming language. Annotate the transcript with seven significant observations about Standard ML. Excellent or additional observations may earn up to a total of three points of extra credit.

```
$ sml
Standard ML of New Jersey v110.69 [built: Mon Jun  8 23:24:21 2009]
```

A few students said, "`sml` starts a REPL". That was a little thin, but good enough!

```
- 5 + ~7;
val it = ~2 : int
```

Tilde is the unary negation operator.

Some said that tilde indicates an approximate value, and although 5 plus around 7 wouldn't be around 2, we took it, on grounds of creativity.

`int` is SML's name for an integer type.

The name `it` is bound to the last value produced.

Semicolons are required after an expression.

```
- (1, 2.0, "three");
val it = (1,2.0,"three") : int * real * string
```

There are tuples.

Tuples can be heterogenous.

`real` and `string` are names for types.

The type of tuples is shown as *`TYPE1 * TYPE2 * ... * TYPEN`*.

An ML programmer would read that type as "int cross real cross string", by the way.

```
- explode "test";
val it = [#"t",#"e",#"s",#"t"] : char list
```

`string` and `char` are distinct types. (Worth two points.)

`char` literals are `#"c"`.

List types are shown with the suffix `list`, in contrast to Haskell's `[TYPE]` notation.

Juxtaposition is function call.

```
- implode it;
val it = "test" : string
```

`implode` and `explode` are inverses.

```
- map (fn(n) => n * 3) [3, 1, 5, 9];  {- See above re map -}
val it = [9,3,15,27] : int list
```

Anonymous functions are supported.

Higher-order functions are supported.

`{- ... -}` is a comment

`[...]` is a list literal

```
- it::[];
val it = [(1,2.0,"three")] : (int * real * string) list
```

`::` is a "cons" operation.

```
- fun f1 a b = [a,b];
val f1 = fn : 'a -> 'a -> 'a list
```

`fun` is used to declare a function.

An equals sign separates the "parameters" (which could be patterns, just like Haskell) from the expression that specifies the value.

Functions are curried.

Type variables are denoted with an apostrophe. (An ML programmer would read `'a` as "alpha" and `'b` as "beta". You might have occasionally heard me inadvertently use "alpha" and "beta" when reading a type variable in Haskell.)

```
- fun f2 a b = [a = b];
val f2 = fn : ''a -> ''a -> bool list
```

The boolean type is named `bool`.

= is an equality operator.

If memory serves, only 2-3 students observed that the type variables for `f2` are preceded with two apostrophes instead of only one. I believe only student went further and correctly guessed that the double apostrophes rose from the comparison. That observation was surely worth two points.

Background: Standard ML doesn't have the notion of type classes like `Eq`, but it does distinguish types whose values can be compared for equality, and `''a` would be recogized as an *equality type*.

```
- 3 + 4.5;
stdIn:1.1-1.6 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
```

"Mixed-mode" arithmetic is not allowed.

```
- (hd [1,2,3], tl [1,2,3]);
val it = (1,[2,3]) : int * int list
```

`hd` and `tl` are "head" and "tail"

I last taught Standard ML in 372 in Fall 2006. My slides are here: http://cs.arizona.edu/classes/cs372/fall06/sml.sli.pdf You'll see that a lot of ML examples translate well into Haskell examples...

Problem 17: ***(7 points)*** *(one point each unless otherwise indicated)* (mean: 3.1, median: 3.5)

Answer the following general questions.

(1) *What's something significant related to programming languages that you remember from the JarWars video we viewed and discussed during the second-to-last class?*

Tiger was the code name for Java 5, which introduced generics.

(2) *Ralph Griswold said, "If you're going to invent a language, be sure to invent a language that* you want to use*."*

(3) *What is the fundamental characteristic of a dynamically typed language?*

In general, the type of an expression can't be known without executing the code.

(4) *How does Icon avoid the "to versus through" problem that plagues string indexing in many languages and libraries?*

String positions are considered to be between characters.

(5) *What's something significant about Icon you recall from the Icon by Observation exercise during the last class? (Hint: Don't cite your answer for the previous question!)*

`*x` produces the number of elements in `x`.

(6) *With programming languages in general, what's the fundamental difference between a statement and an expression?*

Expressions produce a value; statements don't. Typically the only reason to execute a statement is to produce a side-effect.

(7) *Which of the three languages we covered this semester are you most glad we covered, and why?*

Haskell	22
Prolog	20
Ruby	9

Extra Credit Section (½ point each unless otherwise noted) (mean: x, median: y)

(1) *In as few words as possible, what is "The Cathedral and the Bazaar"? (Mentioned in* `a7` *solutions.)*
Book

(2) *In what month of the year 1891 were classes first held at The University of Arizona?*
October. See also
http://www.arizona.edu/topics/about-university/about-university-arizona/ua-history-and-traditions

(3) *With Prolog in mind, why is "camelcase variable" an oxymoron?*
Something like `maxValue` would be an atom!

(4) *The technology known as Leda was mentioned on Piazza. What is Leda?*
A multi-paradigm programming language created by Tim Budd.

(5) Draw an appropriate avatar for any one of the three languages we covered.
My favorite was Mr. Ferra's "Haskell-Guy", but I'm inclined to call him Lambda Man.

(6) Name a language that was once studied in depth in 372 but isn't any more.
C++, Icon, Standard ML, Lisp, Emacs Lisp are five I know of.

(7) *Who do you think whm will vote for in November's presidential election?*
There was a lot of variety here, including "nobody", "not Trump", Ralph Griswold, Mickey Mouse (an always popular write-in, election after election), and "self".

Popular guesses were these:

Bernie	9
Hillary	12
Trump	6

Early on I liked Fiorina, Cruz, Rubio, and Trump; I later came to like Kasich. But Trump's getting my vote in November!

(8) *Why did whm's solution for* `pipes.pl` *have the following line?* `do(p) :- do(pipes).`
So the pipes could be shown with just `p.`.

(9) *(1 point)* `mostfreq.rb` *says to assume that* `Hash` *has a* `sort_by_value` *method. Write Ruby code that makes that be true.*

```
class Hash
    def sort_by_value
        self.sort {|a,b| a[1] <=> b[1]}
        end
    end
```

(10) What did Knuth say about premature optimization?

It's the root of all evil.

(11) (1 point) Using only `append` *and* `length`*, and no recursion, write a Prolog predicate* `longer(A,B)`*, which is true iff list* `A` *is longer than list* `B`*.*

```
    longer(A,B) :- length(A,N), length(X,N), append(B,[_|_],X).
        % by Patrick
```

(12) (1 point) In SNOBOL4, how do you indicate where control should go if a statement fails?

```
    ... :F(LABEL)
```

(13) whm hates writing up Piazza posts with suggested readings! They usually don't align well with his slides and he wonders if anybody actually does any of the readings. Over the course of the full semester, how many hours do you estimate you spent doing the suggested readings? (This is just data collection, no right/wrong.)

```
    N = 43
    mean = 3.670
    median = 1.000
```

(14) Write a joke about programming languages.

A number of students expressed their true feelings about various languages with these one-word offerings:

C, Java, JavaScript, Perl, and Prolog.

I believe these are original and worth repeating:

"Prolog was invented by aliens, it's more logical than life."

"I just made a new programming language called FUN (Fantastically Underachieving Newbie!)"

"It won't right on the first run, or the second, or the third, but it might eventually."

"Prolog should be called Prolonger than Haskell."

"What did Ruby say to her boyfriend Prolog on Valentine's Day? You better HAS something good for me or I'll KELL you!"

"Why are Java programmers wealthy? Inheritance. [crickets]"

"Knock, knock.
    Who's there?
Haskell.
    Go away."

"My love for Prolog is TRUE."

(15) Finish this sentence: "If I only remember one thing about 372 it will be ...".

Here are a few:

"To start early."

"the FRIDAY NIGHT CLUB"

"That logic programming is virtually impossible."

"Pancakes" [several of these]

"The day you almost lost your voice during the first week of lecture." [almost?]

"Ruby is a cool language and to stay away from Prolog."

"If you are looking for a job, everyone should know [you are looking for a job]."

"append in Prolog is amazing"

"Programming problems are better when they have pancakes/food in them."

"pancakes"

"The horror of pancakes."

"Prolog programmers think Haskell is a cute toy."

"How much I love quizzes."

"1000+ slides"

"Haskell sucks"

"Ralph Griswold founded the CS department in 1971."

Statistics

All scores, in order:

```
    104.00,  95.50,  95.00,  92.00,  92.00,  92.00,  91.50,  90.50,  88.50,  88.50,
    87.50, 87.50, 87.50, 86.50, 86.00, 86.00, 85.00, 83.00, 81.50, 81.00, 80.50,
    80.50, 79.50, 79.00, 79.00, 77.50, 77.00, 76.50, 75.00, 74.50, 74.00, 72.00,
    71.50, 71.00, 70.50, 70.00, 69.50, 69.00, 68.50, 68.50, 67.50, 62.50, 62.50,
    62.50, 61.00, 58.00, 58.00, 58.00, 57.50, 57.25, 57.00, 56.50, 55.50, 54.00,
    50.50, 39.00, 39.00, 31.00
```

```
N = 58
mean = 73.306
median = 74.75
```

Using the Tester
CSC 372
Last updated: January 29, 2016 17:00

The syllabus says,

> *For programming problems great emphasis will be placed on the ability to deliver code whose output exactly matches the specification. Failing to achieve that will typically result in large point deductions, sometimes the full value of the problem.*

Whenever possible I'll use an automated testing tool, "the Tester", to test your solutions for programming problems. For each assignment I'll make available a "student set" of tests that you can run yourself using the Tester. The set of tests used when grading (the "grading set") will often have additional tests. Unless otherwise specified for a problem or an entire assignment, passing all the tests in the student set will guarantee at least 75% of the points for a given problem. It some cases it will be higher, or even 100%, with the student set used as the grading set. I'll call that 75% minimum the "student set guarantee".

Test cases in the grading set are weighted. Those weights are not supplied in the student set but a rule of thumb is that cases for basic functionality are weighted more than edge cases.

Sometimes I'll give you a break for bonehead mistakes but there's no excuse for not using the Tester. If a student says, "I spent many hours on this and it works great, but it failed every test because I had an extra space at the of a line.", I'll ask, "Why didn't you use the Tester?"

The Tester is on lectura

The Tester is a collection of bash scripts, Ruby programs, an Icon program, and assorted files that are all on lectura. The instructions in this document assume that you're working on lectura.

You'll use the Tester by running `aN/tester`, where `N` is the assignment number. For assignment 3 you'll be using `a3/tester`. Note that the path, `a3/tester`, assumes that you've made an `a3` symlink, as described in the `a3` write-up.

When to use the Tester

If you're doing TDD with an xUnit tool like JUnit, it's appropriate to run tests frequently, typically after almost every change, but that's not the intention with the Tester. I recommend that you first manually test your code with examples from the write-up and other cases that come to mind. When things are look right to the naked eye, then that's the time to run the Tester.

There is an HUnit for Haskell, as well as Hspec, QuickCheck and more, but I haven't seen anything that looks likely to provide more benefit than trouble for our modest needs in 372.

Quick summary of usage, for the TL;DR crowd

With the `a3` symlink in place and your solution for `warmup.hs` in the current directory, you can test it like this:

```
% a3/tester warmup.hs
```

To stop it early, use `^C` (control+C). It might take more than one `^C`, too.

You can also name multiple problems to be tested:

```
% a3/tester join cpfx warmup
```

The above also shows that the `.hs` suffix is not required.

It's a tedious time-killer to scroll back looking for the start of the Tester's output. One approach is to pipe into `less`:

```
% a3/tester warmup | less
```

An alternative on OS X with Terminal and iTerm is to do `cmd-K` before running the Tester—that clears the scrollback, so when you scroll back, you'll be at the start of the Tester output. I don't know of a keyboard shortcut to clear the scrollback with PuTTY but `spring16/bin/clr` is a two-line script that clears PuTTY's scrollback. (Let me know if you know a simpler way to do that.)

You'll probably want to test problems one at a time, as you develop them, but to test all problems you can run the tester with no arguments. We can combine that with `grep` to simply look for failures:

```
% a3/tester | grep FAIL
```

Be sure to capitalize `FAIL`, or use `grep -i fail`, to ignore case.

GREAT IDEA: Do `a3/tester | grep FAIL` as a final double-check before running `a3/turnin.`

More detail on running the tester

For the purpose of this example we'll imagine that there are two more problems on `a3`: `hello.hs` and `letters.hs`.

Let's work with a simple "hello" function, whose code is correct and is in the file `hello.hs`:

```
% cat hello.hs
hello s = "Hello, " ++ s ++ "!"

% ghci hello.hs
...
> :type hello
hello :: [Char] -> [Char]

> hello "whm"
"Hello, whm!"
```

Here's a test run with no failures:

```
% a3/tester hello

-------------------------------------------------------------
|                                                           |
|                          hello                            |
|                                                           |
-------------------------------------------------------------
```

```
------------------------------------------------------------
|                                                          |
|                      Test Execution                      |
|                                                          |
------------------------------------------------------------


Test: 'ulimit -t 2; a3/tesths hello.hs ':type hello'': PASSED

Test: 'ulimit -t 2; a3/tesths hello.hs 'hello "world"'': PASSED
```

Those two lines starting with "`Test:` " indicate that two tests were run. Both passed. I've underlined and bolded the text that shows what's actually being tested. The first test, `:type hello`, checks the type of the function `hello`. The second test runs the function, with `hello "world"`.

The `Test:` lines start with `ulimit -t 2;`, which limits the CPU time for the test to two seconds. The text that follows that semicolon, up to the final apostrophe, is the exact command that's run for that test. You can do a copy/paste to run it yourself. Let's try both of them:

```
% a3/tesths hello.hs ':type hello'
*Main> > > "TESTING START"
> hello :: [Char] -> [Char]
> Leaving GHCi.

% a3/tesths hello.hs 'hello "world"'
*Main> > > "TESTING START"
> "Hello, world!\n"
> Leaving GHCi.
```

`a3/tesths` is a bash script that loads the named file with `ghci` and then feeds the third argument, such as "`:type hello`", into `ghci`.

Note: If you include the `ulimit -t 2;` when trying a test, like this,

```
% ulimit -t 2; a3/tesths hello.hs 'hello "world"'
```

you'll set the CPU time limit to two seconds for all future commands in that instance of bash. If you inadvertently do that, you'll need to log out and log in again to clear it. (An ordinary user can decrease their CPU time limit, but cannot raise it.)

Understanding differences reported by the Tester

If a test fails, the `diff` command is to used to show the differences between the expected output and the actual output. "diffs" can sometimes be hard to understand. Googling for "understanding diffs" or "deciphering diffs" turns up a lot of stuff, but here are a couple of Tester-specific examples.

Let's intentionally break `hello` by removing the comma in "`Hello, `". Here's what the Tester produces, with line numbers added for reference. Line 5 is long and is shown wrapped around.

```
% a3/tester hello.hs
[...header lines not shown...]

1.  Test: 'ulimit -t 2; a3/tesths hello.hs ':type hello'': PASSED
2.
```

```
3.  Test: 'ulimit -t 2; a3/tesths hello.hs 'hello "world"'': FAILED
4.  Differences (expected/actual):
5.  *** a3/master/tester.out/hello.out.02   2016-01-28
    12:52:52.292586244 -0700
6.  --- tester.out/hello.out.02   2016-01-28 23:22:41.190616251 -0700
7.  ***************
8.  *** 1,3 ****
9.    *Main> > > "TESTING START"
10. ! > "Hello, world!"
11.   > Leaving GHCi.
12. --- 1,3 ----
13.   *Main> > > "TESTING START"
14. ! > "Hello world!"
15.   > Leaving GHCi.
```

The type of `hello` is unaffected by removing that comma but the output differs, so the first test still passes but the second test now fails.

Lines 5 and 6 name the two files that are being "diffed" (compared). I've underlined and bolded the file names. The first is the file that contains the expected output, `a3/master/tester.out/hello.out.02`. The second, `tester.out/hello.out.02`, contains the output produced by running `a3/tesths hello.hs 'hello "world"'` in the current directory. That directory, `tester.out`, was created in the current directory by the Tester, to hold various files created by the testing process. Needless to say, you can look at both files with `cat`, `less`, editors, or any other tool.

The names of the files being compared are preceded by `***` and `---` , which are used later, on lines 8 and 12, to identify the files those blocks of text come from. Line 8's "`*** 1,3 ****`" means that what follows are lines 1-3 from the expected output. Line 12's "`--- 1,3 ----`" means that what follows are lines 1-3 from the actual output. (Diffs in tester output always follow the convention of showing the expected output first and the actual output second.)

The exclamation marks on lines 10 and 14 indicate that those lines differ between the expected and actual output. If we didn't already know what we did to break it, we might need to look close to see that the lines differ by only a single comma.

For a more interesting "diff", let's work with a function named `letters` that prints the first N lower-case letters, one per line. Like the examples on slides 136-148, and the `street` problem on `a3`, this function directly produces printed output using `putStr` rather than producing a value that is in turn displayed by `ghci`. Here's an example of expected behavior:

```
> letters 4
a
b
c
d
```

Here's what the Tester shows with our buggy version, with line numbers added to aid explanation:

```
% a3/tester letters.hs
[...header lines not shown...]

1.  Test: 'ulimit -t 2; a3/tesths letters.hs 'letters 4'': FAILED
```

```
2.  Differences (expected/actual):
3.  *** a3/master/tester.out/letters.out.01 2016-01-28
    12:58:44.406350815 -0700
4.  --- tester.out/letters.out.01 2016-01-28 23:25:53.772537760 -0700
5.  ***************
6.  *** 1,6 ****
7.    *Main> > > "TESTING START"
8.  ! > a
9.    b
10. - c
11.   d
12.   > Leaving GHCi.
13. --- 1,8 ----
14.   *Main> > > "TESTING START"
15. ! >
16. ! a
17. ! x
18.   b
19.   d
20. + Done!
21.   > Leaving GHCi.
```

We see in line 1 that `letters 4` is the Haskell expression that's being tested.

Lines 3 and 4 identify the two files being diffed. Note that line 3 wraps around. Blocks from the expected output will be identified with `***`; blocks from the actual output will be identified with `---`.

The exclamation marks on line 8 and lines 15-17 show that those sections apparently correspond to each other but their content differs.

The minus sign on line 10 shows that there's a line in the expected output, with just a "`c`", that doesn't appear in the actual output.

The plus sign on line 20 shows that there's a line in the actual output, "`Done!`", that doesn't appear in the expected output.

If we have trouble understanding a diff, it often helps to directly examine the files being diffed. Here's the file with the expected output:

```
% cat a3/master/tester.out/letters.out.01
*Main> > > "TESTING START"
> a
b
c
d
> Leaving GHCi.
```

Here's what was actually output when tested:

```
% cat tester.out/letters.out.01
*Main> > > "TESTING START"
>
a
```

```
x
b
d
Done!
> Leaving GHCi.
```

Of course, instead of looking at the file with the actual output, we could try manually running the exact command the tester ran:

```
% a3/tesths letters.hs 'letters 4'
*Main> > > "TESTING START"
>
a
x
b
d
Done!
> Leaving GHCi.
```

For complex differences you might open the expected and actual files in side-by-side windows in an editor. A simple form of that is provided by `vimdiff`: (type `:q<ENTER>` TWICE to get out!)

```
% vimdiff a3/master/tester.out/letters.out.01 tester.out/letters.out.01
```

It's not shown in the examples above but following the `diff` output is a line showing the names of the files that were diffed:

```
Test: 'ulimit -t 2; a3/tesths letters.hs 'letters 4'': FAILED
Differences (expected/actual):
...
+ Done!
  > Leaving GHCi.

Files diffed:
a3/master/tester.out/letters.out.01 tester.out/letters.out.01
```

That's provided so you can select the whole line with multiple clicks, type `vimdiff` or some other command and then paste both file names onto that line.

`pr -mT` provides a simple side-by-side display of two files:

```
% pr -mT a3/master/tester.out/letters.out.01 tester.out/letters.out.01
*Main> > > "TESTING START"          *Main> > > "TESTING START"
> a                                 >
b                                   a
c                                   x
d                                   b
> Leaving GHCi.                     d
                                    Done!
                                    > Leaving GHCi.
```

`diff -y`, which produces a side-by-side diff, is sometimes useful.

If `diff` is claiming a difference but the text looks identical, the problem might be trailing whitespace or embedded non-printable characters, like NULs (ASCII code 0). Problems like that can be turned up by piping into `cat -A`:

```
% a3/tester hello | cat -A
...
```

Exceeding the time limit—handled poorly...

A bug in a recursive function can produce infinite recursion. Infinite recursion will cause the test's time limit to be exceeded and the test will be killed. Sadly, the Tester doesn't provide any clear evidence of the time limit being exceeded. Here's what we see for a diff with a version of `hello` that infinitely recurses:

```
Test: 'ulimit -t 2; a3/tesths hello.hs 'hello "world"'': FAILED
Differences (expected/actual):
*** a3/master/tester.out/hello.out.02   2016-01-28
12:52:52.292586244 -0700
--- tester.out/hello.out.02     2016-01-28 23:36:23.020876709 -0700
***************
*** 1,3 ****
  *Main> > > "TESTING START"
! > "Hello, world!\n"
! > Leaving GHCi.
--- 1,2 ----
  *Main> > > "TESTING START"
! >
\ No newline at end of file
```

Note two things: (1) The actual output doesn't end with "`Leaving GHCi.`" (2) `diff` says there's no newline at the end of the file. The combination of those two things typically indicates the time limit was exceeded.

A good next step is to try the command yourself, without a time limit. Let's try it outside the Tester:

```
 % a3/tesths hello.hs 'hello "world"'
...wait a while...give up...hit ^C
```

The time limits set for tests are usually far more than what's needed but in rare cases you may find that your solution is simply slow, and that it does complete when run outside the Tester. If so, let us know. If it's not outrageously slow, we might just bump up the time limit on the test.

Icon

CSC 372, Spring 2015
The University of Arizona
William H. Mitchell
whm@cs

A little history

Icon is a descendent of SNOBOL4 and SL5.

Icon was designed at the University of Arizona in the late 1970s by a team lead by Ralph Griswold. The first implementation was in Ratfor (rational FORTRAN), to facilitate porting Icon to a variety of machines. It was later reimplemented in C.

The last major upheaval in the language itself was in 1982, but a variety of minor elements have been added in the years since.

Idol, an object-oriented derivative was developed in 1988 by Clint Jeffery.

Graphics extensions evolved from 1990 through 1994.

Unicon (Unified Extended Icon) evolved from 1997 through 1999 and incremental change continues. Unicon has support for object-oriented programming, systems programming, and programming-in-the-large.

The development of Icon was supported by about a decade of funding by the National Science Foundation.

Efficiency by virtue of limited resources

Compared to today, computing resources were very limited when Icon was developed.

The Ratfor implementation of Icon was developed on PDP-10 mainframe with perhaps 1.5 MIPS and maybe a megabyte or two of virtual address space. However, that was a timesharing system that supported users campus-wide and was quite slow at times.

The UNIX implementation of Icon was developed on a PDP-11/70 owned by the CS department. It limited programs to 64k bytes of program code and 64k bytes of data. Its speed was perhaps 1 MIP.

Due to these limits Icon's implementation was required to be small and efficient.

A little Icon by observation

```
% /cs/www/classes/cs372/spring15/bin/ie -nn
Icon Evaluator, Version 1.1, ? for help
][ 3+4
  r := 7  (integer)

][ "abc" || (3 + 4.5)
  r := "abc7.5"  (string)

][ type(r)
  r := "string"  (string)

][ type(type)
  r := "procedure"  (string)

][ *r
  r := 9  (integer)
```

Icon by observation, continued

```
][ s := "testing"
  r := "testing"  (string)

][ s[1]
  r := "t"  (string)

][ s[-1]
  r := "g"  (string)

][ s * 3
Run-time error 102, numeric expected
offending value: "testing"
{"testing" * 3} from line 40 in ._ie_tmp.icn

][ repl(s,3)
  r := "testingtestingtesting"  (string)
```

Icon by observation, continued

```
][ 'testing this'
  r := ' eghinst'  (cset)

][ &digits
  r := &digits  (cset)

][ split("Thursday, 4/29/2015", &digits)
  r := L1:["Thursday, ","/","/"]  (list)

][ split("Thursday, 4/29/2015", ~&digits)
  r := L1:["4","29","2015"]  (list)

][ *(&letters ++ &digits)
  r := 62  (integer)
```

Icon by observation, continued

```
][ line := read()
here's some input!
  r := "here's some input!"  (string)

][ write("just",2,"test")
just2test
  r := "test"  (string)

][ x := [1, [2], "three"]
  r := L1:[1,L2:[2],"three"]  (list)

][ x[1]
  r := 1  (integer)

][ *(x ||| x)
  r := 6  (integer)
```

Icon by observation, continued

```
][ t := table("Go fish!")
  r := T1:[]  (table)

][ t["one"] := 1
  r := 1  (integer)

][ t['two'] := 2
  r := 2  (integer)

][ t
  r := T1:["one"->1,'otw'->2]  (table)

][ t["three"]
  r := "Go fish!"  (string)

][ table()[1]
  r := &null  (null)
```

String indexing

In Icon, positions in a string are <u>between</u> characters and run in both directions.

```
  1   2   3   4   5   6   7   8
  |   |   |   |   |   |   |   |
    t   o   o   l   k   i   t
  |   |   |   |   |   |   |   |
 -7  -6  -5  -4  -3  -2  -1   0
```

Several forms of subscripting are provided.

```
    ][ s[3:-1]
      r := "olki"  (string)

    ][ s[1+:4]
      r := "tool"  (string)
```

s[i] is a shorthand for **s[i:i+1]**

```
    ][ s[5]
      r := "k"  (string)
```

What problem does between-based positioning avoid?

It avoids the "to" vs. "through" problem.

Strings use "value semantics"

Assignment of string values does not cause sharing of data:

```
][ s1 := "Knuckles"
  r := "Knuckles"  (string)

][ s2 := s1
  r := "Knuckles"  (string)

][ s1[1:1] := "Fish "
  r := "Fish "  (string)

][ s1
  r := "Fish Knuckles"  (string)

][ s2
  r := "Knuckles"  (string)
```

Any substring can be the target of an assignment.

Failure

A key design feature of Icon is that an expression can fail to produce a result. A simple example of an expression that fails is an out of bounds string subscript:

```
][ s := "testing"
  r := "testing"  (string)

][ s[5]
  r := "i"  (string)

][ s[50]
Failure
```

We say, "**s[50]** fails"—it produces no value.

If an expression produces a value it is said to have *succeeded*.

When an expression is evaluated it either succeeds or fails.

Failure, continued

An important rule:

An operation is performed only if a value is present for all operands. If due to failure a value is not present for all operands, the operation fails.

Another way to say it:

If evaluation of an operand fails, the operation fails. And, failure propagates.

```
][ s := "testing"
  r := "testing"  (string)

][ "x" || s[50]
Failure

][ reverse("x" || s[50])
Failure

][ s := reverse("x" || s[50])       # s is unchanged
Failure
```

When working in Icon, unexpected failure is the root of madness.

Failure, continued

Another example of an expression that fails is a comparison whose condition does not hold:

```
][ 3 = 0
Failure

][ 4 < 3
Failure
```

A comparison that succeeds produces the value of the right hand operand as the result of the comparison:

```
][ 1 < 2
  r := 2  (integer)

][ 10 ~= 20
  r := 20  (integer)
```

Failure, continued

What do these expressions do?

```
write(a < b)

f(a < b, x = y, 0 ~= *s)

max := max < n

max <:= 30
```

How do Java exceptions compare to Icon's failure mechanism?

Failure, continued

Here's a string that represents a hierarchical data structure:

```
/a:b/apple:orange/10:2:4/xyz/
```

Major elements are delimited by slashes; minor elements are delimited by colons.

Imagine an Icon procedure to access an element given a major and minor:

```
][ extract("/a:b/apple:orange/10:2:4/xyz/", 2, 1)
  r := "apple"  (string)

][ extract("/a:b/apple:orange/10:2:4/xyz/", 3, 4)
Failure
```

Implementation:

```
procedure extract(s,m,n)
   return split(split(s, '/')[m], ':')[n]
end
```

How does **extract** make use of failure?

The **while** expression

Icon has several traditionally-named control structures, but they are driven by success and failure.

Here's the general form of the while expression:

```
while expr1 do
    expr2
```

If *expr1* succeeds, *expr2* is evaluated. This continues until *expr1* fails.

Here is a loop that reads lines and prints them:

```
while line := read() do
   write(line)
```

The **while** expression

At hand:

```
while line := read() do
   write(line)
```

If no body is needed, the **do** clause can be omitted.

Here's a more concise way to write the loop above.

```
while write(read())
```

What causes termination of this more compact version?

read() fails at end of file.

That failure propagates outward, causing the **write()** to fail.

The **while** terminates because its control expression, **write(...)**, failed.

The **&** operator

The general form of the **&** operator:

```
expr1 & expr2
```

expr1 is evaluated first. If *expr1* succeeds, *expr2* is evaluated. If *expr2* succeeds, the entire expression succeeds and produces the result of *expr2*. If either *expr1* or *expr2* fails, the entire expression fails.

Example:

```
while line := read() & line[1] ~== "." do
    write(line)
```

Here is pseudo-code for the implementation of **&**:

```
Value andOp(Value expr1, Value expr2) { return expr2 }
```

How does it work?

andOp only gets called if evaluation of both operands succeeded, so all it needs to do is to return the value of the right-hand operand!

Procedures

All executable code in an Icon program is contained in *procedures*. A procedure may take arguments. It may return a value of interest.

Execution of an Icon program begins by calling the procedure **main**.

A simple program with two procedures:

```
procedure main()
   while n := read() do
      write(n, " doubled is ", double(n))
end

procedure double(n)
   return 2 * n
end
```

Use **icont** to compile and run. **-s** suppresses some chatty stuff. **-x** says to execute; without it, **icont** would only produce the executable **double**.

```
% icont –s double.icn –x
```

Procedures, continued

A procedure may produce a result or it may fail. Here's a more flexible version of double:

```
procedure double(x)
   if type(x) == "string" then
      return x || x
   else if numeric(x) then
      return x + x
   else
      fail
end
```

```
][ double(5)
  r := 10  (integer)

][ double("xyz")
  r := "xyzxyz"  (string)

][ double([1,2])
Failure
```

Procedures, continued

Does **double** exemplify duck typing?

```
procedure double(x)
   if type(x) == "string" then
      return x || x
   else if numeric(x) then
      return x + x
   else
      fail
end
```

Here is the Ruby counterpart:

```
def double x
   x * 2
end
```

Is Icon duck-challenged? If so, why?

What are tradeoffs in having different operators for addition and concatenation?

```
][ s := "abc"; n := 123
][ s || n
   r := "abc123"  (string)
```

```
>> s = "abc"; n = 123
>> s + n
TypeError: ...
>> s + n.to_s
=> "abc123"
```

Call tracing in procedures

One of Icon's debugging facilities is call tracing. Tracing is activated by setting the keyword **&trace** or the **TRACE** environment variable.

```
% TRACE=-1 icont –s sum.icn -x
       :   main()
sum.icn: 2 | sum(3)
sum.icn: 7 | | sum(2)
sum.icn: 7 | | | sum(1)
sum.icn: 7 | | | | sum(0)
sum.icn: 6 | | | | sum returned 0
sum.icn: 6 | | | sum returned 1
sum.icn: 6 | | sum returned 3
sum.icn: 6 | sum returned 6
6
sum.icn: 3 main failed
%
```

```
% cat -n sum.icn
1   procedure main()
2      write(sum(3))
3   end
4
5   procedure sum(n)
6      return if n = 0 then 0
7               else n + sum(n-1)
8   end
```

Generator basics

In most languages, evaluation of an expression produces either a result or an exception.

We've seen that Icon expressions can fail, producing no result.

Some expressions in Icon are *generators*, and can produce many results.

Here's a generator:

```
    1 to 3
```

`1 to 3` has the *result sequence* `{1, 2, 3}`.

The `.every` *directive* of `ie` can be used to show the result sequence of a generator:

```
    ][ .every 1 to 3
      1  (integer)
      2  (integer)
      3  (integer)
```

Generator basics, continued

Some languages allow generative constructs in particular contexts, like a "for" control structure but an Icon generator can appear at any place in any expression.

```
][ .every repl("*", 1 to 3)
  "*"  (string)
  "**"  (string)
  "***"  (string)

][ s := "abcd"
][ .every write(reverse(s[1:2 to *s]))
a
  "a"  (string)
ba
  "ba"  (string)
cba
  "cba"  (string)
```

Generator basics, continued

If an expression fails to produce a result, Icon resumes the last generator to produce a result.

```
][ i := 1 to 10 & i % 2 = 0 & write(i) & 1 = 2
2
4
6
8
10
Failure
```

Icon backtracks through previous expressions to find an active generator. If one is found, it starts evaluating the following expressions again.

What does this back and forth movement remind you of?

The above is an example of *goal-directed evaluation*.

Generator basics, continued

The **every** control structure drives a generator to failure, making it produce all its results. Example:

```
every i := 1 to 5 do
    write(repl("*", i))
```

Output:

```
*
**
***
****
*****
```

Here's a more concise version:

```
every write(repl("*", 1 to 5))
```

The generator "bang" (!)

Another built-in generator is the unary exclamation mark, called "bang".

It is polymorphic, as is the size operator (*). For character strings it generates the characters in the string one at a time.

```
][ every write(!"abc")      Note: using every control structure
a
b
c
Failure
```

The result sequence of `!"abc"` is `{"a", "b", "c"}`.

For lists, ! generates the elements:

```
][ every write(![&lcase,&ucase,&digits])
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
Failure
```

"bang", continued

A program to count vowels appearing on standard input:

```
procedure main()
   vowels := 0
   while line := read() do
      every c := !line do
         if c == !"aeiouAEIOU" then
            vowels +:= 1
   write(vowels, " vowels")
end
```

Execution:

```
% echo "testing" | icont -s vowels.icn -x
2 vowels
```

"bang", continued

Speculate: What does the following program do?

```
procedure main()
    lines := []
    every push(lines, !&input)
    every write(!lines)
end
```

Execution:

```
% seq 3 | icont -s tac.icn -x
3
2
1
```

Alternation

The alternation control structure looks like an operator:

```
    expr1 | expr2
```

This creates a generator whose *result sequence* is the result sequence of ***expr1*** followed by the result sequence of ***expr2***.

For example, the expression

```
    3 | 7
```

has the result sequence {3, 7}.

The expression

```
    (1 to 5) | (5 to 1 by -1)
```

has the result sequence {1, 2, 3, 4, 5, 5, 4, 3, 2, 1}.

Alternation, continued

Alternation used in goal-directed evaluation:

```
procedure main()
 while time := (writes("Time? ") & read()) do {
   if time = (10 | 2 | 4) then
      write("It's Dr. Pepper time!")
   }
end
```

A program to read lines from standard input and write out the first twenty characters of each line:

```
procedure main()
  while line := read() do
     write(line[1:(21|0)])
end
```

Would it work with `line[1:21]` instead?

Multiple generators

An expression may contain any number of generators:

```
][ every write(!"ab", !"+-", !"cd")
a+c
a+d
a-c
a-d
b+c
b+d
b-c
b-d
Failure
```

Generators are resumed in a LIFO manner: the generator that most recently produced a result is the first one resumed.

What does **every write(!x == !y)** do?

Multiple generators, continued

Recall this vowel counter:

```
procedure main()
   vowels := 0
   while line := read() do
      every c := !line do
         if c == !"aeiouAEIOU" then
            vowels +:= 1
   write(vowels, " vowels")
end
```

Here is a more concise version, using multiple generators:

```
procedure main()
   vowels := 0
   every !!&input == !"aeiouAEIOU" do
       vowels +:= 1
   write(vowels, " vowels")
end
```

Multiple generators, continued

A program to show the distribution of the sum of three dice:

```
procedure main()
  every N := 1 to 18 do {
     writes(right(N,2), " ")
     every (1 to 6) + (1 to 6) + (1 to 6) = N do
        writes("*")
     write()
   }
end
```

```
 1
 2
 3 *
 4 ***
 5 ******
 6 **********
 7 ***************
 8 *********************
 9 *************************
10 ***************************
11 ***************************
12 *************************
13 *********************
14 ***************
15 **********
16 ******
17 ***
18 *
```

String scanning

The SNOBOL4 programming language has a very powerful string pattern matching facility but it shares a problem with regular expressions in Ruby: you're either doing regular computation or you're matching a pattern—the operations can't be interleaved smoothly, like they can be in Prolog.

A design goal for Icon was to integrate string pattern matching with regular computation—match a little, compute a little, match a little, compute a little, etc.

The end result was a handful of *string scanning* functions that can be used in conjunction with Icon's other facilities to achieve the desired full integration of string pattern matching with regular computation.

In the end, Icon's string scanning facility turned to be a disappointment. It is small and powerful but the techniques involved are non-trivial. Too often, the first version of code using string scanning is not correct. Ditto for the second version.

The following slides provide a very brief look at Icon's string scanning facility. (About 50-60 slides are required for an in-depth study of the facility.)

The scanning operator

String scanning is initiated with `?`, the scanning operator:

```
expr1 ? expr2
```

The value of `expr1` is established as the *subject* of the scan (&subject). The scanning position in the subject (`&pos`) is set to 1. `expr2` is then evaluated.

A trivial example:

```
][ "testing" ? { write(&subject); write(&pos) }
testing
1
  r := 1  (integer)
```

The result of the scanning expression is the result of `expr2`.

String scanning functions

There are two string scanning functions that change **&pos**—the current position in **&subjec**t:

move(n) Move forwards or backwards by **n** characters.
(**&pos +:= n**)

tab(n) Move to position **n**. (**&pos := n**)

Both **move** and **tab** return the string between the old and new values of **&pos**.

String scanning functions, continued

There is a group of functions that produce positions to be used in conjunction with **tab**:

many(cs) produces position after run of characters in **cs**
upto(cs) generates positions of characters in **cs**
find(s) generates positions of **s**
match(s) produces position after **s**, if **s** is next
any(cs) produces position after a character in **cs**
bal(s, cs1, cs2, cs3)
similar to **upto(cs)**, but used with "balanced" strings.

There is one other string scanning function:
pos(n) tests if **&pos** is equivalent to **n**

The string scanning facility consists of only the above functions (including **move** and **tab**), the **?** operator, and the **&pos** and **&subject** keywords. Nothing more.

upto, **many**, and **tab**

Here's a procedure that sums the integers it finds in a string:

```
procedure sumnums(s)
   sum := 0
   s ? while tab(upto(&digits)) do
        sum +:= integer(tab(many(&digits)))
     return sum
end
```

upto(&digits) produces the position of the next digit after **&pos**, the current position. The wrapping **tab(...)** advances **&pos** to that position.

tab(many(&digits)) advances over the digits and returns them as a string.

```
][ sumnums("values: 10, 20 and 30")
  r := 60  (integer)
```

A goal of string scanning was to be able to interleave scanning operations with ordinary computation. Does **sumnums** exemplify that?

Here's a procedure that generates matches for strings of the form $\mathbf{a}^N\mathbf{b}^N\mathbf{c}^N$:

```
procedure aNbNcN()
   tab(upto('a')) &
   start := &pos &
   as := tab(many('a')) &
   bs := tab(many('b')) &
   cs := tab(many('c')) &
   *as = *bs = *cs &
   suspend [start, as || bs || cs]
end
```

The **&**s are needed to produce procedure-wide backtracking.

A **main** to test with:

```
procedure main()
   while writes("Line? ") & line := read() do {
      line ? every m := aNbNcN() do
         printf("At %d: '%s'\n", m[1], m[2])
      }
end
```

```
Line? aabbcc abbc aaabbbccc ab abc
At 1: 'aabbcc'
At 13: 'aaabbbccc'
At 26: 'abc'
```

Graphics in Icon

Facilities for graphical programming in Icon evolved in the period 1990-1994.

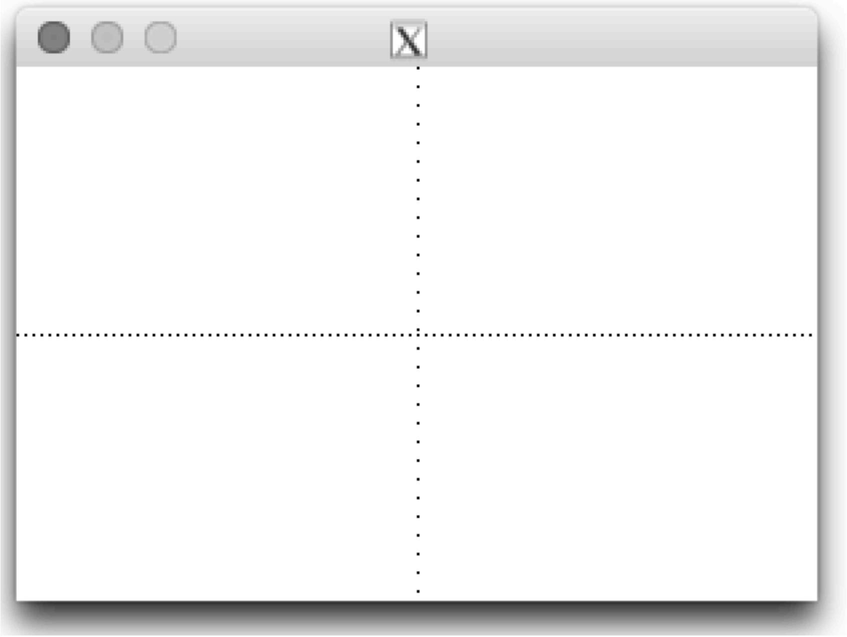
A philosophy of Icon is to insulate the programmer from details and place the burden on the language implementation. The graphics facilities were designed with same philosophy.

Icon's graphical facilities are built on the X Window System on UNIX machines. On Microsoft Windows platforms the facilities are built on the Windows API.

Graphics, continued

Here is a program that draws a "crosshair" of dots in a window:

```
link graphics
procedure main() # g1.icn
   WOpen("size=300,200")

   every x := 0 to 300 by 3 do
      DrawPoint(x, 100)   # horizontal

   every y := 0 to 200 by 7 do
      DrawPoint(150, y)  # vertical

   WDone()  # wait for a "q" to be typed
end
```



Graphics, continued

Here is a program that randomly draws points.

```
link graphics

$define Height 700    # symbolic constants
$define Width 900     #  via preprocessor

procedure main() # g2.icn
   WOpen("size=" || Width ||","||Height)

   repeat {
      DrawPoint(?Width-1, ?Height-1)
      }
end
```

Speculate: How long will it take it to black out every single point?

Simple game

```
$define Width 500
$define Height 500
procedure main() # g3.icn
   WOpen("size="||Width||","||Height, "drawop=reverse")

   x := ?Width; y := ?Height; r := 50
   repeat {
      DrawCircle(x, y, r)
      hit := &null
      every 1 to 80 do {
         WDelay(10)
         while *Pending() > 0 do {
            if Event()=== &lpress then {
               if sqrt((x-&x)^2+(y-&y)^2) < r then {
                  FillCircle(x,y, r)
                  WDelay(500)
                  FillCircle(x,y,r)
                  hit := 1
                  break break
                  }}}}
      DrawCircle(x,y,r)
      if \hit then r *:= .9 else r *:= 1.10
      x := ?Width; y := ?Height
      }
end
```

This program draws a circular target at random location If the player clicks inside the target within 800ms, the radius shrinks by 10%. If not, the radius grows by 10%.

Kobes' Curve Editor

Steve Kobes wrote this very elegant curve editor in 2003:

```
procedure main()
  WOpen("height=500", "width=700", "label=Curve Editor")
  pts := []
  repeat case Event() of {
      &lpress: if not(i := nearpt(&x, &y, pts)) then
                  { pts |||:= [&x, &y]; draw(pts)}
      &ldrag: if \i then { pts[i] := &x; pts[i + 1] := &y; draw(pts) }
      !"Qq": break
  }
end

procedure draw(pts)
  EraseArea()
  DrawCurve!(pts ||| [pts[1], pts[2]])
  every i := 1 to *pts by 2 do
      FillCircle(pts[i], pts[i + 1], 3)
end

procedure nearpt(x, y, pts)
  every i := 1 to *pts by 2 do
      if abs(x - pts[i]) < 4 & abs(y - pts[i + 1]) < 4 then return i
end
```

Icon resources

http://www.cs.arizona.edu/icon is the Icon home page.

http://www.cs.arizona.edu/~whm/451 has the materials from a full-semester course I taught on Icon in 2003.

On the home page, under "Books About Icon", I recommend three:

The Icon Programming Language, 3rd edition

A comprehensive treatment of the language, with numerous examples of non-numerical applications.

The Implementation of the Icon Programming Language

For a time, Ralph taught a course that covered the implementation of Icon's run-time system. This book rose out of that course. If you're interested in how dynamic languages are implemented, this book is definitely worth a look.

Graphics Programming in Icon

Some parts are dated but lots of interesting stuff, like Lindenmayer systems and a caricature algorithm.

unicon.org is the home page for Unicon, a derivative of Icon that supports object-oriented programming, systems programming, and programming-in-the-large.

UNIX Stuff for 372
William H. Mitchell
Last Revised: February 7, 2015 at 3:45pm

I've talked to a few students that are having trouble dealing with some of the mechanics of UNIX. Here are a few things that may help.

My old 352 slides

A set of my 352 UNIX slides is in http://cs.arizona.edu/~whm/352/unix.sli.pdf. Those slides represent a basic set of things that I think are important/useful/handy to know when working with UNIX. Whether you've had 352 or not, I bet you'll learn at least a few things if you just flip through those slides. Additional 352 stuff is in that same directory.

Dead simple, low-tech backup

Here's a one-line, low-tech backup that you can run on lectura, in your a2 directory:

```
$ pr *.hs | mail -s 372 your-netid@email.arizona.edu
```

The `pr` command writes the content of each of your `.hs` files in turn to standard output, and that output is piped into `mail`, a command line mailer. (Try `man mail`.) You'll get a message with the subject "372". For a more descriptive subject you could use `-s "372 a2 backup"` instead, but note that quotes are used in that case to get the spaces embedded in the string. `pr` generates some page headers that you'll have to hack out if you need to recover files, but your source code will all be there.

Like any backup procedure, try some recoveries for practice to possibly discover omissions or flaws in the procedure.

Package that low-tech backup in a script!

The easier it is to do a backup the more likely you are to do it.

Put that "pipeline" with `pr` above into a file. I like short names for commands I'm going to run frequently, so I'll suggest the name `bk`. Here's what that file should look like, as shown by `cat`:

```
$ cat bk
pr *.hs | mail -s "372 a2 backup" whm@email.arizona.edu
```

We could run it right now with `bash bk` but let's make it executable, to save a little typing:

```
$ chmod +x bk
```

We should now be able to run `bk` by simply typing `./bk`. Depending on how your path is configured you might be able to type just `bk`. (For more details on `bk` vs. `./bk`, search for `PATH` in my 352 slides and also see http://cs.arizona.edu/~whm/352/Spring05/dotornot.pdf.)

You're now able to ship off a copy of your `.hs` files with 3-5 keystrokes. If you want to send your backups to two locations, maybe a Yahoo mail account, too, just duplicate that line and change the email address, or see slide 88 in my UNIX set to learn about the `for` statement in bash.

Things like Dropbox and Google Drive give you instaneous backup and that's great but I really hate to pay for the same real estate twice, so I supplement those with some low-tech backups. If Dropbox were to have a catastrophic failure, I'm sure they'd put out a well-written note expressing their regret and a rededication to ensuring data integrity but I doubt they'd be mailing any checks to help say how sorry they are that all your files are gone.

`~/.snapshot` has periodic snapshots of all your files on lectura

Let's look at my `~/.snapshot` directory on lectura. We'll use `-1` (dash-one) to force single-column output:

```
$ ls -1 ~/.snapshot
hourly.0
hourly.1
hourly.2
...
nightly.0
nightly.1
...
weekly.0
weekly.1
...
```

Each of those is a directory that's a "copy" of my directory tree on lectura at those particular times. You can see that the frequency of retention decreases for older copies but the latest snapshots are four hours apart. If you want to see and/or recover a file from some point in the past, just `cd` into the appropriate snapshot directory and copy the file out. Example:

```
$ cd ~/.snapshot/hourly.2/372/a2
$ ls -l
[...lots...]
$ cat join.hs
--this version compiles!
x = 1
$ cp join.hs ~/372/a2/join-compiles.hs
```

See http://faq.cs.arizona.edu/index.php?action=artikel&cat=5&id=58 and http://www.cs.arizona.edu/computing/accounts/snapshots.html for more details on `~/.snapshot`. See also slide 55 in my UNIX set.

What does tilde (~) mean in a path?

`bash`, like many shells, interprets a path like `~/xyz` to mean "`xyz` in my home directory".

The `echo` command can be used to see what a command line looks like after the shell does its various expansions and substitutions on a command line. Here

```
$ echo ~/.ghci
/home/whm/.ghci
$ echo ~/372/a2/cpfx.hs
/home/whm/372/a2/cpfx.hs
```

Tilde substitution done only if the tilde is at the start of a line:

```
$ echo ~/x~x/y~
/home/whm/x~x/y~
```

Try `echo ~postgres/x`.

See slide 62+ in my UNIX set for more on tilde expansion.

Although unrelated to ~, here are some examples of exploring with `echo`:

```
$ echo $HOME
/home/whm
```

```
$ echo $PATH
/home/whm/sbin:/home/whm/3bin:/home/whm/bin:/usr/local/bin:...lots
more...
$ echo $RANDOM $RANDOM
2344 16797
$ echo *.hs
join-compiles.hs join.hs x.hs
```

See slide 82+ in my UNIX set for more on variables (`$...`); 64+ covers wildcards.

A little more with symlinks

The `a2` writeup recommends creating this symlink in your assignment 2 directory,

```
$ ln -s /cs/www/classes/cs372/spring15/a2 .
```

so you can then run the tester with `a2/tester`. Here's a symlink that lets you run the tester with `./t` (or maybe just `t`—see dotornot.pdf above.)

```
 $ ln -s a2/tester t
```

That creates a symlink named `t` that uses the `a2` symlink.

For future assignments you might do something more general, like this for `a3`:

```
$ cd ~/372
$ ln -s /cs/www/classes/cs372/spring15 www
$ cd a3
$ ln -s ../www/a3 a3
$ ln -s a3/tester t
```

That creates a symlink `www` in `~/372` that references the root directory for the 372 materials parked on the web. Then, in `a3`, you make a symlink named `a3` that uses the `~/372/www` symlink, saving the trouble of having to type out `/cs/www/....` In turn, the symlink `t` uses the `a3` symlink—triple indirection!

See slide 58+ in my UNIX set for more on symlinks.

In WinSCP use Commands > Keep Remote Directory up to Date... (!)

If I had a dollar for every time I've seen a student drag a file between WinSCP windows to copy the latest from their machine to lectura, I might be able to pay for a class trip to Hawaii for Spring Break.

Instead of that per-save dragging, just get the source and target directories open in WinSCP and do Commands > Keep Remote Directory up to Date... It'll ask if you want to perform full synchronization of the remote directory first. You do, unless you've been editing some files directly on lectura. (If that's the case, just get the latest, greatest versions back to your Windows machine before you activate this automatic synchronization.)

WinSCP has good help about this facility.

Cyberduck is a WinSCP equivalent for the Mac that I've heard good things about but have never experimented with.

Remote editing

I typically use Emacs' remote editing facilities for editing files on lectura from my Mac.

Students last Spring seemed to have good luck using http://wbond.net/sublime_packages/sftp for remote editing with Sublime. I've got a dim memory of some remote editing package for Sublime having a race condition that would occasionally result in another user getting your code when they tried to open their file but I can't find any trace of that of that incident now. If you should see anything remotely like that (no pun intended!), let me know ASAP.

If you'd rather type just `lec` instead of `lectura.cs.arizona.edu`...

On OS X, Linux and Windows, host names are looked up in `/etc/hosts` before consulting DNS. This lets you add a short name for a host, like `lec` for `lectura.cs.arizona.edu`. Here's the line for lectura from my `/etc/hosts`:

```
192.12.69.186  lec
```

It'd be an unusual situation for the IP address for lectura to be changed but if it ever should, you'd need to update that `/etc/hosts` entry.

I use `sudo vi /etc/hosts` to edit that file on my Mac but you can use any editor.

Windows has a corresponding file but it's a longer story. Here's a link for what looks to be a pretty good how-to: http://www.rackspace.com/knowledge_center/article/how-do-i-modify-my-hosts-file

Once you've done this, just about everywhere you used to type `lectura.cs.arizona.edu` you can type just `lec` instead. That works in PuTTY, WinSCP, Emacs, on the command line, and lots more.

If you're doing web development, a handy addition to the 127.0.0.1 entry is just an "l" (L).

```
127.0.0.1       localhost l
```

With that you can hit `l://...` instead of `localhost://...`