University of Arizona, Department of Computer Science

CSc 372 — Assignment 5 — Due noon, Fri Nov 4 — 7%

Christian Collberg
October 24, 2005

1 Introduction

The purpose of this assignment is for you to get familiar with list manipulation in Prolog, as well as learning some common Prolog programming patterns, such as *generate-and-test*. We will also use Prolog's ability to define operators for the first time.

Every predicate should be commented. At the very least, the comments should state what the predicate does, which arguments it takes, and what result it produces.

You should make your predicates as simple and elegant as possible.

You will be graded primarily on **correctness**, **elegance**, **clarity**, and **documentation**.

This assignment is graded out of 100. It is worth 7% of your final grade.

Input files to the programs in this assignment can be found here: `http://www.cs.arizona.edu/~collberg/Teaching/372/2005/Assignments/index.html`

2 Useful Predicates

You may find the following built-in Prolog predicates useful.

The predicate `name(Atom,List)` splits up an atom into a string (list of ASCII characters), and converts a string back into an atom:

```
| ?- name(hello,L).
L = [104,101,108,108,111]
| ?- name(L, [104,101,108,108,111]).
L = hello
```

Prolog's built-in operator `\+` ("not") succeeds when its argument fails:

```
| ?- \+ member(a,[]).
yes
| ?- \+ member(a,[a]).
no
```

The *cut* operator `!` acts like a *barrier*: once execution has passed the `!` operator, it will never backtrack past it again.

```
fruit(apple).
fruit(orange).
fruit(pear).

| ?- fruit(L).
L = apple ? ;
L = orange ? ;
L = pear
yes

| ?- fruit(L),!.
L = apple
yes
```

The `fail` predicate will always fail. It is often used to generate all possible solutions to a query:

```
tastiness(apple,2).
tastiness(pear, 6).
tastiness(orange,10).

yummy :-
   fruit(L),
   tastiness(L, T),
   T > 5,
   write(L), nl,
   fail.
yummy.

| ?- yummy.
orange
pear
```

The `yummy` predicate will "iterate" (using backtracking) back and forth between `fruit(L)` and `fail` until no more tasty fruits can be found. The second `yummy` predicate makes sure that `yummy` will always succeed (return `yes`).

3 l33t Translator [25 points]

```
pr010g \/\/45 ph1r57 |>351g/\/3|> 70 |>0 <0/\/\pu74710/\/41 11/\/gu1571<5,
ph0r 3x4/\/\p13 70 7r4/\/51473 b37\/\/33/\/ 3/\/g115h 4/\/|> phr3/\/<h. 7h15
\/\/0u1|> 54v3 b07h <u17ur35 phr0/\/\ 4<7u411y h4v1/\/g 70 134r/\/ ph0r31g/\/
14/\/gu4g35. 45 4 <0/\/53qu3/\/<3, pr010g h45 /\/\4/\/y pr0p3r7135 7h47 /\/\4|35
17 u53phu1 70 v4r10u5 7r4/\/514710/\/ 745|5.
```

Or, in plain English...

Prolog was first designed to do computational linguistics, for example to translate between English and French. This would save both cultures from actually having to learn foreign languages. As a consequence, Prolog has many properties that makes it useful for various translation tasks.

`l33t` (pronounced *elite*) is a "language" used by `k00l d00d5` (cool dudes) in online conversations, such as IM, chat rooms, EQ, etc. There are many dialects, depending on how much of a `ph@ l33+ h4x0r` (fat elite hacker) you are.

You can read more here: `http://www.planetquake.com/turkey/l33t_a.htm` and many other places on the web. Google `l33t` for more info.

Here's a translation table (stored in the file `r00l5.pl`) from English letters to `l33t`:

```
translate("a", "4").
translate("b", "b").
translate("c", "<").
translate("c", "k").
translate("d", "|>").
translate("e", "3").
translate("f", "ph").
translate("g", "g").
translate("g", "9").
translate("h", "h").
translate("i", "1").
translate("j", "j").
translate("k", "|").
translate("l", "1").
translate("m", "/\\/\\").
translate("n", "/\\/").
translate("o", "0").
translate("p", "p").
translate("q", "q").
translate("r", "r").
translate("s", "5").
translate("t", "7").
translate("t", "+").
translate("u", "u").
translate("v", "v").
translate("w", "\\/\\/").
translate("x", "x").
translate("y", "y").
translate("z", "z").
translate(" ", " ").
translate(".", ".").
translate("!", "!").
translate("?", "?").
translate("-", "-").
translate(";", ";").
translate(":", ":").
```

```
translate(",", ",").
```

To simplify the translation we only translate one character at a time, ignoring such transformations as `at`↦`@`.

Write a translator that takes an English string as input and produces one or more `l33t` translations:

```
| ?- [r00l5,l33t]. % load the translation table and your program
| ?- english2l33t("cat",E).
E = [60,52,55] ? ;
E = [60,52,43] ? ;
E = [107,52,55] ? ;
E = [107,52,43] ? ;
```

Since Prolog represents strings as lists of ASCII character numbers the output gets a bit hard to read. The following predicate cleans up the output by converting the string to an atom:

```
l33t(E,M) :-
   english2l33t(E,N),
   name(M,N).

| ?- l33t("cat",E).
E = '<47' ? ;
E = '<4+' ? ;
E = k47 ? ;
E = 'k4+' ? ;
```

`l33t` only deals with lower-case letters since h4x0r5 are way too busy to use the shift key. Therefore write a predicate `tolower` that converts all the upper-case letters in a string to lower case, and leaves the other characters untouched:

```
| ?- tolower("HELLO",L), name(N,L).
L = [104,101,108,108,111]
N = hello ?

| ?- tolower("HeLLo D00d",L), name(N,L).
L = [104,101,108,108,111,32,100,48,48,100]
N = 'hello d00d' ?
```

Our `l33t` translator now becomes

```
english2l33t(L,E) :- ...
tolower(L,E) :- ...

l33t(E,M) :-
   tolower(E,L),
   english2l33t(L,N),
   name(M,N).
```

```
| ?- l33t("CaT",E).
E = '<47' ? ;
E = '<4+' ? ;
E = k47 ? ;
E = 'k4+' ? ;
```

My solution is 25 lines long.

4 Rational Arithmetic [25 points]

Write a package for doing rational arithmetic in Prolog. A rational number (such as $\frac{5}{6}$) is represented by the Prolog term `5\6`. The following operations should be made available (`A`, `B` and `E` are rational expressions):

A isr E	—	Evaluate `E` and instantiate `A` to the result.
A + B	—	`A` plus `B`.
A - B	—	`A` minus `B`.
A * B	—	`A` times `B`.
A / B	—	`A` divided by `B`.
~A	—	The inverse of `A`.

The result of an evaluation should always be given in the simplest form. Here are some examples:

```
?- [rat].
?- A isr 20\14
   A = 10\7

?- A isr 20\14 + 4\7
   A = 2

?- A isr ~2
   A = 1\2

?- A isr ~2\3 + 5
   A = 13\2

?- A isr 2\5/10\3
   A = 3\25
```

Your program should look like this:

```
:- op(900, xfx, isr).
:- op(100, fy, ~).
:- op(10, xfx, \).

% Auxiliary predicates here...
```

```
% Greatest common divisor...
gcd(X, Y, D) :- ...

% Simplify the rational number A\B into C\D
% by dividing A and B by their largest common factor
simp(A\B, C\D) :- ...

Z isr X \ Y :- ...
Z isr X * Y :- ...
Z isr X + Y :- ...
Z isr X - Y :- ...
Z isr X / Y :- ...
Z isr ~X :- ...
```

The `op` *directives* create new operators. `Z isr X \ Y :- ...` is equivalent to `isr(Z, X \ Y) :- ...`, but using an operator notation for `isr`. `gcd(X,Y,D)` computes the *greatest common divisor* of two integers `X` and `D`. This function is used to simplify a rational number.

My solution is 55 lines long.

5 A Friendster Database [25 points]

Assume that we have a database of who is friends with whom, such as might be used by `friendster.com` or `orkut.com`:

```
friend(christian,margaret).
friend(christian,jas).
friend(christian,todd).
friend(christian,ji).
friend(christian,geener).

friend(todd,christian).
friend(todd,susan).

friend(susan,todd).

friend(jas,christian).
friend(jas,geener).
friend(jas,clark).

friend(geener,christian).
friend(geener,jas).
friend(geener,ji).

friend(clark,pat).
```

```
friend(pat,mike).
friend(pat,clark).

friend(margaret,christian).

friend(ji,christian).
friend(ji,geener).
```

This database is stored in the file `friends.pl`.

We would like to be able to answer the question: "who is X friends with, through at most three people, and what is the friendship path?" For example:

```
| ?- [friendster,friends].

| ?- friendster(christian,clark,L).
L = [christian,jas,clark] ?

| ?- friendster(christian,pat,L).
L = [christian,jas,clark,pat] ?

| ?- friendster(christian,mike,L).
no

| ?-  friendster(christian,mark,L).
no

| ?- friendster(christian,christian,L).
no

| ?- friendster(christian,geener,L).
L = [christian,geener] ? ;
L = [christian,jas,geener] ? ;
L = [christian,ji,geener] ? ;
no
```

Note that the path from `christian` to `mike` is too long for them to be friends, and that `christian` is friends with `geener` through three different paths.

My solution is 12 lines long.

6 Generating Vanity Plates [25 points]

Write a Prolog program to generate *vanity plates*, car license plates with cute messages. The idea is to try to shorten long words taken from a standard dictionary by replacing sequences of letters with shorter character sequences. The resulting license plate must consist of at most six letters and digits. Here are some examples:

Original Word	License Plate	Rules Used
fortunate	4TUN8	for → 4, ate → 8
stench	S10CH	ten → 10
ozone	OZ1	one → 1
mezzanine	MEZZA9	nine → 9
foursquare	4SQARE	four → 4, qu → q
forklift	4KLIFT	for → 4
european	EUROPN	pea → p
detour	DE2UR	to → 2

The interface to your program is simply the predicate `vanity` (with no arguments), which generates *all possible variations* of vanity plates, given a set of rules and a dictionary of words:

```
| ?- [short,rules,vanity].
| ?- vanity.
[a,n,t,e,a,t,e,r] --> [a,n,t,e,8,r]
[b,e,f,o,r,e] --> [b,e,4,e]
[c,l,i,f,f,o,r,d] --> [c,l,i,f,4,d]
[f,o,r,t,u,n,a,t,e] --> [4,t,u,n,8]
[e,u,r,o,p,e,a,n] --> [e,u,r,o,p,n]

yes
```

My rule database `rules.pl` looks like this:

```
rule([a,t,e],[8]).
rule([e,i,g,h,t],[8]).
rule([t,e,n],[1,0]).
rule([f,o,r],[4]).
rule([f,o,u,r],[4]).
rule([s,i,x],[6]).
rule([o,n,e],[1]).
rule([t,o,o],[2]).
rule([t,w,o],[2]).
rule([t,o],[2]).
rule([n,i,n,e],[9]).
rule([b,e,e],[b]).
rule([s,e,e],[c]).
rule([s,e,a],[c]).
rule([d,e,e],[d]).
rule([d,e,a],[d]).
rule([g,e,e],[g]).
rule([k,a,y],[k]).
rule([p,e,e],[p]).
```

```
rule([p,e,a],[p]).
rule([q,u,e],[q]).
rule([a,r,e],[r]).
rule([e,s,s],[s]).
rule([t,e,a],[t]).
rule([t,e,e],[t]).
rule([y,o,u],[u]).
rule([y,e,w],[u]).
rule([j,a,y],[j]).
rule([e,g,s],[x]).
```

My dictionary database `short.pl` looks like this:

```
word([a,n,t,e,a,t,e,r]).
word([b,e,a,t,s]).
word([b,e,f,o,r,e]).
word([c,l,i,f,f,o,r,d]).
word([f,o,r,t,u,n,a,t,e]).
word([e,u,r,o,p,e,a,n]).
```

If you are feeling adventurous you could also try a bigger database of words, `words.pl`, but then you have to increase `gprolog`'s internal table sizes. This is how you do it under unix:

```
> setenv GLOBALSZ 100000
> setenv TRAILSZ 10000
> gprolog
| ?- [words,rules,vanity].
| ?- vanity.
 ...
```

The assignment can be solved easily (albeit inefficiently) by judicious use of the `append` predicate and Prolog's *generate-and-test* pattern. My solution is twenty lines long. The predicate that does the actual work is six lines long.

There are many collections of vanity plates on the internet that can give you ideas for constructing your own rule base. See, for example, `http://www-chaos.umd.edu/misc`.

7 Submission and Assessment

The deadline for this assignment is noon, Fri Nov 4. It is worth 7% of your final grade.

You should submit the assignment electronically using the `Unix` command

`turnin cs372.5 l33t.pl rat.pl friendster.pl vanity.pl`.

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.