# CSc 372

# Comparative Programming Languages

## *4 : Haskell — Basics*

Christian Collberg

collberg+372@gmail.com

Department of Computer Science

University of Arizona





# The Hugs Interpreter

- The Haskell implementation we will be using is called Hugs.
- You interact with Hugs by typing commands to the interpreter, much like you would to a powerful calculator:

```
$ hugs
> 6 * 7
42
> 126 ‘div‘ 3
4
```



# The Hugs Interpreter...

- Haskell programs (known as scripts) are just text files with function definitions that can be loaded into the interpreter using the `:load script` command:

```
$ hugs
> :load file.hs
```

- Haskell scripts take the file extension `.hs`.



# Haskell Types

# Expressions

- When we “run” a Haskell program, we actually evaluate an evaluate expression, and the result of the program is the value of that expression.
- Unlike Java programs. Haskell programs have no statements — there is no way to assign a new value to a variable for example.



# Haskell Types

- Haskell is strongly typed. This means that every expression has exactly one type.
- Haskell is statically typed. This means that the type of an expression can be figured out before we run the program.
- The basic types in Haskell include
  1. Int (word-sized integers)
  2. Integer (arbitrary precision integers)
  3. Float (Floating point numbers)
  4. Tuples and Lists
  5. Strings (really just lists)
  6. Function types



# Type inference

- In Java and most other languages the programmer has to declare what type variables, functions, etc have.
- We can do this too, in Haskell:

```
> 6*7 :: Int
42
```

:: Int asserts that the expression `6*7` has the type `Int`.

- Haskell will check for us that we get our types right:

```
> 6*7 :: Bool
ERROR
```



# Type inference. . .

- We can let the Haskell interpreter infer the type of expressions, called type inference.
- The command `:type expression` asks Haskell to print the type of an expression:

```
> :type "hello"
"hello" :: String

> :type True && False
True && False :: Bool

> :type True && False :: Bool
True && False :: Bool
```

# Simple Types



# Int

- The `Int` type is a 32-bit signed integer, similar to Java's `int` type:

```
Prelude> (3333333 :: Int) * (444444444444444 :: Int)
Program error: arithmetic overflow
```

Some Haskell versions may instead overflow the integer (yielding a negative number).



# Int — Operators

- The normal set of arithmetic operators are available:

| Op | Precedence | Associativity | Description |
|---|---|---|---|
| `^` | 8 | right | Exponentiation |
| `*, /` | 7 | left | Mul, Div |
| `` `div` `` | 7 | free | Division |
| `` `rem` `` | 7 | free | Remainder |
| `` `mod` `` | 7 | free | Modulus |
| `+, -` | 6 | left | Add, Subtract |
| `==,/=` | 4 | free | (In-) Equality |
| `<,<=,>,>=` | 4 | free | Relational Comparison |



# Int. . .

- Note that the `div` operator has to be in backquotes when used as an infix operator:

```
> 4*12-6
42
> 126 `div` 3
42
> div 126 3
42
```

# Int...

- The standard precedence and associativity rules apply:

```
1+2-3       ⇒ (1+2)-3
1+2*3       ⇒ 1+(2*3)
2^3^4       ⇒ 2^(3^4)
4==5==6     ⇒ 0.666666666666667
12/6/3      ⇒ ERROR
12/(6/3)    ⇒ 6
```



# Integer

- Haskell also has an infinte precision integer type, similar to Java's `java.math.BigInteger` class:

```
> (3333333 :: Integer) * (444444444444444 :: Integer)
1481481333333331851852
```

- `Integers` are the default integer type:

```
> 2^64
18446744073709551616
```



# Integer...

- `Ints` and `Integers` aren't compatible:

```
> (3333333 :: Integer) * (44 :: Int)
ERROR - Type error in application
```

- but we can convert from an `Int` to an `Integer`:

```
> (toInteger (55 :: Int)) * (66 :: Integer)
3630
```



# Float and Double

- Haskell also has built-in floating point numbers `Float` and `Double`:

```
> sqrt 2 :: Float
1.414214
> sqrt 2 :: Double
1.4142135623731
```

- `sqrt` is a built-in library function.
- Double is the default:

```
> sqrt 2
1.4142135623731
```

# Char

- Literals: `'a'`, `'b'`. Special characters: `'\n'` (newline).
- ASCII: `'\65'` (decimal), `'\x41'` (hex).
- There are standard functions on characters (`toUpper`, `isAlpha`, etc) defined in the a separate module Char:

```
> :load Char
> toUpper 'A'
'A'
> toUpper 'a'
'A'
> ord 'a'
97
```



# Char — Built-in Functions

```
ord :: Char -> Int
char :: Int -> Char
toUpper, toLower :: Char -> Char
isAscii,isDigit,··· :: Char -> Bool
isUpper,isLower,··· :: Char -> Bool
ord 'a' ⇒ 97    toUpper 'a' ⇒ 'A'
chr 65 ⇒ 'A'    isDigit 'a' ⇒ False
```



# String

- Strings are really lists of characters.

```
> "hello"
"hello"
> :type "hello"
"hello" :: String
> "hello" :: String
"hello"
> length "hello"
5
> "hello" ++ " world!"
"hello world!"
```

- `++` does string/list concatenation.



# Bool

- There are two boolean literals, `True` and `False`

| Op | Precedence | Associativity | Description |
|---|---|---|---|
| `&&` | 3 | right | logical and |
| `\|\|` | 2 | right | logical or |
| `not` | 9 | – | logical not |

```
3 < 5 && 4 > 2           ⇔ (3 < 5) && (4 > 2)
True || False && True    ⇔ True || (False && True)
```

# Haskell Functions



# Functions

- Here's the ubiquitous factorial function:

```
fact ::  Int -> Int
fact n =    if n == 0 then
               1
            else
               n * fact (n-1)
```

- The first part of a function definition is the type signature, which gives the domain and range of the function:

```
fact ::  Int -> Int
```

- The second part of the definition is the function declaration, the implementation of the function:

```
fact n = if n == 0 then ···
```



# Functions...

- The syntax of a type signature is

```
fun_name ::  arg_types
```

  `fact` takes one integer input argument and returns one integer result.
- The syntax of function declarations:

```
fun_name param_names = fun_body
```

- `fact` is defined recursively, i.e. the function body contains an application of the function itself.
- Function application examples:

```
fact 1      ⇒  1
fact 5      ⇒  120
fact (3+2)  ⇒  120
```



# List and Tuple Types

# Lists

- A list in Haskell consists of a sequence of elements, all of the same type:

```
> [1,2,3]
[1,2,3]
> [True,False] ::  [Bool]
[True,False]
> :type [True,False]
[True,False] ::  [Bool]
> :type [['A','B'],['C','D'],[]]
[['A','B'],['C','D'],[]] ::  [[Char]]
> [1,True]
ERROR
> length [1,2,3]
3
```



# Tuples

- A Haskell tuple is similar to a record/struct in C – it is a collection of objects of (a limited number of) objects, possibly of different types. Each C struct elements has a unique name, whereas in Haskell you distinguish between elements by their position in the tuple.
- Syntax: $(t_1, t_2, \cdots, t_n)$.

Examples:

```
type Complex = (Float,Float)
mkComplex ::  Float -> Float -> Complex
mkComplex re im = (re, im)
```



# Tuples. . .

```
e Complex = (Float,Float)
omplex ::  Float -> Float -> Complex
omplex re im = (re im)

omplex 5 3 ⇒ (5, 3)

Complex ::  Complex -> Complex -> Complex
Complex (a,b) (c,d) = (a+c,b+d)

Complex (mkComplex 5 3) (mkComplex 4 2) ⇒ (9,5)
```



# Haskell Scripts

# Editing and Loading Scripts

- `:load name` (or `:l name`) loads a new Haskell program.
- `:reload` (or `:r`) reloads the current script.
- `:edit name` (or `:e name`) edits a script. On Unix you can set the `EDITOR` environment variable to control which editor to use:

```
setenv EDITOR emacs
```

- `:?` shows all available commands.
- `:quit` quits Hugs.



# The Offside Rule

- When does one function definition end and the next one begin?

```
square x =  x * x
            +2
cube x = ···
```

- Textual layout determines when definitions begin and end.



# The Offside Rule...

- The first character after the "=" opens up a box which holds the right hand side of the equation:

```
square x = x * x
           +2
```

- Any character to the left of the line closes the box and starts a new definition:

```
square x = x * x
           +2
cube x = ...
```



# Comments

- Line comments start with `--` and go to the end of the line:

```
-- This is a comment.
```

- Nested comments start with `{-` and end with `-}`:

```
{-
   This is a comment.
   {-
      And here's another one....
   -}
-}
```

# Editing Scripts



# Emacs

- On Unix, `emacs` is the editor of choice.
- Depending on your system, it may be called `emacs` or `xemacs`.
- For a list of common commands, see the links below.



# Readings and References

- Chapters 1-3 of *Programming in Haskell*, bu Graham Hutton, is a good introduction to Haskell:
  http://www.cs.nott.ac.uk/~gmh/book.html
- Emacs Guide:
  http://www.cs.arizona.edu/classes/cs372/fall03/04.html
- Emacs Reference Card:
  http://www.cs.arizona.edu/classes/cs372/fall03/emacs.html



# Summary

- Haskell has all the basic types one might expect: `Ints`, `Chars`, `Floats`, and `Bools`.
- Haskell functions come in two parts, the signature and the declaration:

```
fun_name ::  argument_types
fun_name param_names = fun_body
```

- Many Haskell functions will use recursion.
- Haskell doesn't have assignment statements, loop statements, or procedures.
- Haskell tuples are similar to records in other languages.

# Homework

1. Start Hugs.
2. Enter the `commaint` function and try it out.
3. Enter the `addComplex` and `mkComplex` functions and try them out.
4. Try the standard functions `fst x` and `snd x` on complex values. What do `fst` and `snd` do?
5. Try out the Eliza application in `/usr/local/hugs98/lib/hugs/demos/Eliza.hs` on lectura.



# Homework. . .

- Write a Haskell function to check if a character is alphanumeric, i.e. a lower case letter, upper case letter, or digit.

```
? isAlphaNum 'a'
  True
? isAlphaNum '1'
  True
? isAlphaNum 'A'
  True
? isAlphaNum ';'
  False
? isAlphaNum '@'
  False
```



# Homework. . .

- Define a Haskell exclusive-or function.

```
eOr :: Bool -> Bool -> Bool
eOr x y = ···

? eOr True True
  False
? eOr True False
  True
? eOr False True
  True
? eOr False False
  False
```



# Homework. . .

- Define a Haskell function `charToInt` which converts a digit like `'8'` to its integer value 8. The value of non-digits should be taken to be 0.

```
charToInt :: Char -> Int
charToInt c = ···

? charToInt '8'
  8
? charToInt '0'
  0
? charToInt 'y'
  0
```