CSc 520

Principles of Programming Languages

11: Haskell — Basics

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

Basic Types – Int

- As we've seen, Haskell supports the integer (`Int`) type.

Integer Operators:

Op	Precedence	Associativity	Description
`^`	8	right	Exponentiation
`*, /`	7	left	Mul, Div
`div`	7	free	Division
`rem`	7	free	Remainder

Basic Types – Int...

Op	Precedence	Associativity	Description
`mod`	7	free	Modulus
`+, -`	6	left	Add, Subtract
`==,/=`	4	free	(In-) Equality
`<,<=,>,>=`	4	free	Relational Comparison

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


Basic Types – 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

- 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 ···
```


Haskell 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
```


Basic Types – Char

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

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
```


Basic Types – Tuples

- A Haskell tuple is similar to a Pascal record – it is a collection of objects of (a limited number of) objects, possibly of different types. Each Pascal record 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)
```

Basic Types – 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)
```


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 = ...
```


Readings and References

- A free implementation
  `ftp://ftp.dcs.gla.ac.uk/pub/haskell/gofer`.
- Compiler and interpreter available on linux:
  `/home/cs520/2003/bin/linux/gofer`.
- You can also use the Haskell compiler on lectura:
  `/home/cs520/2003/ghc-5.04.1/bin/sparc-sun-solaris2/ghci`.
- `http://dmoz.org/Computers/Programming/Languages/Haskell`.

Summary

- Haskell has all the basic types one might expect: `Int`s, `Char`s, `Float`s, and `Bool`s.
- 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 (Mac || Unix) Haskell.
2. Enter the `commaint` function and try it out.
3. Enter the `addComplex` and `mkComplex` functions and try them out.
4. Turn on tracing (Options:Trace Reductions in MacHaskell) and try the functions again.
5. Try the standard functions `fst x` and `snd x` on complex values. What do `fst` and `snd` do?
6. Try out the Eliza application in `Demos:Eliza`.

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
```