EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

November 10, 2015

Naming Requirements

- Function and argument names must begin with a lower-case letter. For example:

`myFun` `fun1` `arg_2` `x'`

- By convention, list arguments usually have an s suffix on their name. For example:

`xs` `ns` `nss`

The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

```
a = 10

b = 20

c = 30
```

✓

```
a = 10

 b = 20

c = 30
```

✗

```
 a = 10

b = 20

 c = 30
```

✗

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c
    where
       b = 1
       c = 2
d = a * 2
```

means

```
a = b + c
     where
        {b = 1;
         c = 2}
d = a * 2
```

implicit grouping

explicit grouping

Exercises

(1) Try out slides 2-8 and 14-17 using Hugs.

(2) Fix the syntax errors in the program below, and test your solution using Hugs.

```
N = a 'div' length xs
    where
       a = 10
      xs = [1,2,3,4,5]
```

(3) Show how the library function last that selects the last element of a list can be defined using the functions introduced in this lecture.

(4) Can you think of another possible definition?

(5) Similarly, show how the library function init that removes the last element from a list can be defined in two different ways.

PROGRAMMING IN HASKELL

Chapter 3 - Types and Classes

What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

$\boxed{\texttt{Bool}}$

contains the two logical values:

$\boxed{\texttt{False}}$ $\boxed{\texttt{True}}$

Type Errors

Applying a function to one or more arguments of the wrong type is called a <u>type error</u>.

```
> 1 + False
Error
```

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

- In Hugs, the :type command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```

Basic Types

Haskell has a number of basic types, including:

`Bool`	- logical values
`Char`	- single characters
`String`	- strings of characters
`Int`	- fixed-precision integers
`Integer`	- arbitrary-precision integers
`Float`	- floating-point numbers

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

In general:

[t] is the type of lists with elements of type t.

Note:

- The type of a list says nothing about its length:

```
[False,True]       :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Tuple Types

A tuple is a sequence of values of different types:

```
(False,True)      :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

Note:

- The type of a tuple encodes its size:

```
(False,True)       :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b'])  :: (Bool,[Char])
```

Function Types

A <u>function</u> is a mapping from values of one type to values of another type:

```
not     :: Bool → Bool

isDigit :: Char → Bool
```

In general:

> t1 → t2 is the type of functions that map values of type t1 to values to type t2.

Note:

- The arrow → is typed at the keyboard as ->.

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add        :: (Int,Int) → Int
add (x,y)   = x+y

zeroto     :: Int → [Int]
zeroto n    = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add'     :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer x and returns a function add' x. In turn, this function takes an integer y and returns the result x+y.

Note:

- add and add’ produce the same final result, but add takes its two arguments at the same time, whereas add’ takes them one at a time:

```
add  :: (Int,Int) → Int

add’ :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called curried functions, celebrating the work of Haskell Curry on such functions.

- Functions with more than two arguments can be curried by returning nested functions:

```
mult       :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by <u>partially applying</u> a curried function.

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow → associates to the <u>right</u>.

```
Int → Int → Int → Int
```

Means Int → (Int → (Int → Int)).

- As a consequence, it is then natural for function application to associate to the left.

```
mult x y z
```

Means ((mult x) y) z.

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

for any type a, length takes a list of values of type a and returns an integer.

Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2

> length [1,2,3,4]
4
```

a = Bool

a = Int

- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]

id   :: a → a
```

Overloaded Functions

A polymorphic function is called overloaded if its type contains one or more class constraints.

```
sum :: Num a ⇒ [a] → a
```

for any numeric type a, sum takes a list of values of type a and returns a value of type a.

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> sum [1,2,3]
6

> sum [1.1,2.2,3.3]
6.6

> sum ['a','b','c']
ERROR
```

a = Int

a = Float

Char is not a numeric type

- Haskell has a number of type classes, including:

  `Num` - Numeric types

  `Eq` - Equality types

  `Ord` - Ordered types

- For example:

```
(+)  :: Num a ⇒ a → a → a

(==) :: Eq a  ⇒ a → a → Bool

(<)  :: Ord a ⇒ a → a → Bool
```

Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type;

- Within a script, it is good practice to state the type of every new function defined;

- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.

Exercises

(1) What are the types of the following values?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail,init,reverse]
```

(2) What are the types of the following functions?

```
second xs     = head (tail xs)

swap (x,y)    = (y,x)

pair x y      = (x,y)

double x      = x*2

palindrome xs = reverse xs == xs

twice f x     = f (f x)
```

(3) Check your answers using Hugs.