

The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts
Spring 2002

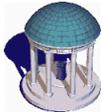
Lecture 19: Functions, Types and Data Structures in Haskell

Felix Hernandez-Campos

Feb 25

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

1



Functions

- Functions are the most important kind of value in functional programming
 - Functions are values!
- Mathematically, a function f associates an element of a set X to a unique element of second set Y
 - We write $f :: X \rightarrow Y$

```
three    :: Integer -> Integer
infinity :: Integer
square   :: Integer -> Integer
smaller  :: (Integer, Integer) -> Integer
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

2



Currying Functions

- Functional can be applied to a partially resulting in new functions with fewer arguments

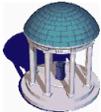
```
smaller      :: (Integer, Integer) -> Integer
smaller (x,y) = if x <= y then x else y

smaller2     :: Integer -> Integer -> Integer
smaller2 x y = if x <= y then x else y
```

- The value of the application `smaller2 x` is a function with type `Integer -> Integer`
 - This is known as *currying* a function

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

3



Curried Functions

- Curried functions help reduce the number of parentheses
 - Parentheses make the syntax of some functional languages (e.g. Lisp, Scheme) ugly
- Curried functions are useful **Function as argument**

```
twice      :: (Integer -> Integer) -> (Integer -> Integer)
twice f x = f (f x)

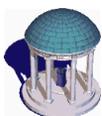
square     :: Integer -> Integer
square x = x * x

quad       :: Integer -> Integer
quad      = twice square
```

Function as result

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

4

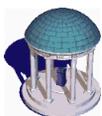


Operators

- *Operators* are functions in infix notation rather than prefix notation
 - E.g. `3 + 4` rather than `plus 3 4`
- Functions can be used in infix notation using the ‘*f*’ notation
 - E.g. `3 `plus` 4`
- Operators can be used in prefix notation using the (*op*) notation
 - E.g. `(+) 3 4`
- As any other function, operators can be applied partially using the *sections* notation
 - E.g. The type of `(+3)` is `Integer -> Integer`

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

5



Operator Associativity

- Operators associate from left to right (*left-associative*) or from right to left (*right-associative*)
 - E.g. `-` is left-associative `3 - 4 - 5` means `(3 - 4) - 5`
- Operator `->` **is** right-associative
 - `x -> y -> z` means `x -> (y -> z)`
- Exercise: deduce the type of `h` in

```
h x y = f (g x y)
```

```
f :: Integer -> Integer
```

```
g :: Integer -> Integer -> Integer
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

6



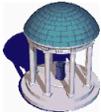
Function Definition

- Functions with no parameters are constants
 - E.g. `pi = 3.14` has type `pi :: Float`
- The definition of a function can depend on the value of the parameters
 - Definitions with *conditional expressions*
 - Definitions with *guarded equation*

```
smaller      :: Integer -> Integer -> Integer
smaller x y = if x <= y then x else y
smaller2 x y
  | x <= y = x
  | y > x  = y
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

7



Recursive definitions

- Functional languages make extensive use of recursion

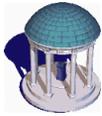
```
fact      :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n - 1)
```

- What is the result of `fact -1`?
- The following definition is more a

```
fact n
  | n < 0   = error "negative argument for fact"
  | n == 0  = 1
  | otherwise = n * fact (n-1)
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

8



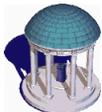
Local Definitions

- Another useful notation in function definition is a *local definition*
- Local declarations are introduced using the keyword `where`
- For instance,

```
f      :: Integer -> Integer -> Integer
f x y
  | x <= 10 = x + a
  | x > 10  = x - a
  where a = square b
        b = y + 1
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

9



Types

- The following primitive types are available in Haskell
 - `Bool`
 - `Integer`
 - `Float`
 - `Double`
 - `Char`
- Any expression in Haskell has a type
 - Primitive types
 - Derived types
 - Polymorphic types

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

10



Polymorphic Types

- Some functions and operations work with many types
- Polymorphic types are specified using *type variables*
- For instance, the curry function has a polymorphic type

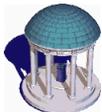
```
curry      :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
```

- Type variables can be qualified using *type classes*

```
(*)       :: Num a => a -> a -> a
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

11



Lists

- Lists are the workhorse of functional programming
- Lists are denoted as sequences of elements separated by commas and enclosed within square brackets
 - E.g. [1, 2, 3]
- The previous notation is a shorthand for the `List` data type
 - E.g. 1:2:3:[]

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

12



Lists Functions

- Functions that operate on lists often have polymorphic types

Polymorphic List

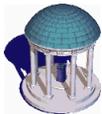
```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
```

Pattern Matching

- In the previous example, the appropriate definition of the function for the specific arguments was chosen using *pattern matching*

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

13



Lists Example Derivation

```
length :: [a] -> Integer
length [] = 0 (1)
length (x:xs) = 1 + length xs (2)
```

```
length [1,2,3]
= { definition (2) }
  1 + length [2,3]
= { definition (2) }
  1 + 1 + length [3]
= { definition of + }
  2 + length [3]
= { definition (2) }
  2 + 1 + length []
```

```
= { definition of + }
  3 + length []
= { definition (1) }
  3 + 0
= { definition of + }
  3
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

14



Lists

Other Functions

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

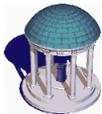
```
head (x:xs) = x
```

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

```
tail (x:xs) = xs
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

15



Data Types

- New data types can be created using the keyword **data**
- Examples

```
data Bool = False | True
```

Type Constructor

```
data Color = Red | Green | Blue | Violet
```

```
data Point a = Pt a a
```

Polymorphic Type

Type Constructor

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

16



Polymorphic Types

- Type constructors have types

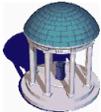
```
Pt :: a -> a -> Point  
data Point a = Pt a a
```

- Examples

```
Pt 2.0 3.0 :: Point Float  
Pt 'a' 'b' :: Point Char  
Pt True False :: Point Bool
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

17



Recursive Types Binary Trees

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

```
Branch :: Tree a -> Tree a -> Tree a
```

```
Leaf :: a -> Tree a
```

- Example function that operates on trees

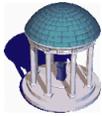
```
fringe :: Tree a -> [a]
```

```
fringe (Leaf x) = [x]
```

```
fringe (Branch left right) = fringe left  
                             ++ fringe right
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

18



List Comprehensions

- Lists can be defined by enumeration using *list comprehensions*

– Syntax:

```
[ f x | x <- xs ]
```

```
[ (x,y) | x <- xs, y <- ys ]
```

Generator

- Example

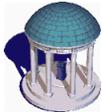
```
quicksort [] = []
```

```
quicksort (x:xs) = quicksort [y | y <- xs, y < x] ++ [x]
```

```
++ quicksort [y | y <- xs, y >= x]
```

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

19



Reading Assignment

- *A Gentle Introduction to Haskell* by Paul Hudak, John Peterson, and Joseph H. Fasel.

– <http://www.haskell.org/tutorial/>

– Read sections 1 and 2

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

20