Table of Contents

Abstract

Do you want a more expressive and concise programming language? A language that catches more of your bugs at compile time, before they cause problems? A language with a fresh perspective that will change the way you think about programming? Meet Haskell! Haskell is an efficient and mature functional programming language that has, for the past quarter century, been pushing the boundaries of what a programming language can do. In this course you will learn the basics of Haskell, as well as some of the features that set Haskell apart from the multitude of other programming languages.

Table of Contents

Functions

Benson Joeris
bjoeris@gmail.com
bjoeris.wordpress.com

pluralsight
hardcore developer training

Overview

- Calling and Defining Functions
- Pure Functions
- Recursion
- Lists/Tuples
- Pattern Matching
- Let/Where
- Laziness

Table of Contents

Calling a Function

```
GHCi> sqrt 3
Result: 1.7320508075688772
```

- No parentheses

Two Arguments

```
GHCi> max 5 7
Result: 7
```

- No commas

When to Use Parentheses

```
GHCi> max (5 + 2) (sqrt 17)
Result: 7
```

- Parentheses for grouping

```
GHCi> (5 + 2) * (3 - 4)
Result: -7
```

Defining Functions

```
square x = x * x
```

- No parentheses around parameters
- No ``return'' keyword

```
GHCi> let square x = x * x
```

Defining Functions with Multiple Parameters

```
-- multiplies the largest of a and b by x
multMax a b x = (max a b) * x
```

Simple Conditionals

```
posOrNeg x =
  if x >= 0
  then "Positive"
  else "Negative"
```

- No parentheses around condition
- No return statements .

Table of Contents

Pure Functions

- *All* Haskell functions are *pure*.
  - Cannot modify state
  - Cannot depend on state
  - Given the same arguments, always returns the same output

Pure Function Examples

- Print a string to the console
  - Not pure -- modifies external state
- Read a file
  - Not pure -- depends on external state at different times
- Compute the length of a string
  - Pure -- no state
- Get the current time
  - Not pure -- returns different values when called at different times
- Get a random number
  - Not pure -- returns different values each time it is called

Table of Contents

Recursion

```
-- pow2 n = 2 to the power n
pow2 n =
  if n == 0
  then 1
  else 2 * (pow2 (n-1))
```

```
int pow2(int n) {
  int x = 1;
  for(int i = 0; i<n; ++i)
    x *= 2;
  return x;
}
```

More Recursion

```
repeatString str n =
  if n == 0
  then ""
  else str ++ (repeatString str (n-1))
```

```
int repeatString(String str, int n) {
  String result = "";
  for(int i = 0; i < n; ++i)
    result += str;
  return result;
}
```

Recursion Replaces Loops

```
int pow2(int n) {
  int x = 1;
  for(int i = 0; i<n; ++i)
    x *= 2;
  return x;
}
```

```
pow2 n = pow2loop n 1 0
pow2loop n x i =
  if i<n
  then pow2loop n (x*2) (i+1)
  else x
```

Table of Contents

Lists

```
x = [1,2,3]
```

```
empty = []
```

```
y = 0 : x -- [0,1,2,3]
```

```
x' = 1 : (2 : (3 : []))
```

```
x'' = 1 : 2 : 3 : []
```

Strings

```
str = "abcde"
```

```
str' = 'a' : 'b' : 'c' : 'd' : 'e' : []
```

Concatenating Lists

```
GHCi> [1,2,3] ++ [4,5]
Result: [1,2,3,4,5]
```

```
GHCi> "hello " ++ "world"
Result: "hello world"
```

Homogeneous Lists

```
error = [1,"hello",2]
```

Table of Contents

Accessing Lists

GHCi> `head [1,2,3]`

Result: `1`

GHCi> `tail [1,2,3]`

Result: `[2,3]`

GHCi> `head (tail [1,2,3])`

Result: `2`

Testing for empty list

GHCi> `null []`

Result: `True`

GHCi> `null [1,2]`

Result: `False`

List Functions

```
double nums =
  if null nums
  then []
  else (2 * (head nums)) : (double (tail nums))
```

List Functions

```
removeOdd nums =
  if null nums
  then []
  else
    if (mod (head nums) 2) == 0 -- even?
    then (head nums) : (removeOdd (tail nums))
    else removeOdd (tail nums)
```

Table of Contents

Tuples

```
x = (1, "hello")
```

```
y = ("pi", 3.14159, [1,2,3], "four")
```

Tuples vs Lists

Tuples	Lists
(…)	[…]
different types	same type
fixed length	unbounded length

Returning Tuples

```
headAndLength list = (head list, length list)
```

Accessing Tuple Elements

GHCi> `fst (1, "hello")`

Result: `1`

GHCi> `snd (1, "hello")`

Result: `"hello"`

Tuple Warning

- Big tuples
- Tuples spanning different parts of an application

Table of Contents

Pattern Matching

```
fst' (a,b) = a
```

```
snd' (a,b) = b
```

Pattern Matching Lists

```
null' [] = True
null' (x : xs) = False
```

Pattern Matching Lists

```
head' (x : xs) = x
head' []       = ?
```

-

Pattern Matching Lists

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

Pattern Matching Lists

```
head' (x : xs) = x
head' [] = error "head of empty list"
```

Using Pattern Matching

```
double nums =
  if null nums
  then []
  else (2 * (head nums)) : (double (tail nums))
```

```
double [] = []
double (x : xs) = (2 * x) : (double xs)
```

Table of Contents

Guards

```
pow2 n
  | n == 0     = 1
  | otherwise = 2 * (pow2 (n-1))
```

- No ``=" before guards -
- ``|" before each guard -

Guards

```
removeOdd nums =
  if null nums
  then []
  else
    if (mod (head nums) 2) == 0 -- even?
    then (head nums) : (removeOdd (tail nums))
    else removeOdd (tail nums)
```

```
removeOdd [] = []
removeOdd (x : xs)
  | mod x 2 == 0 = x : (removeOdd xs)
  | otherwise    = removeOdd xs
```

Table of Contents

Case Expressions

```
double nums = case nums of
  []       -> []
  (x : xs) -> (2 * x) : (double xs)
```

Case Expressions

```
anyEven nums = case (removeOdd nums) of
  []       -> False
  (x : xs) -> True
```

Case Expressions

- No guards in case expressions

Table of Contents

Let Binding

```
fancySeven =
  let a = 3
  in 2 * a + 1
```

Let Binding

```
fancyNine =
  let x = 4
      y = 5
  in x + y
```

Let Example

```
numEven nums =
  let evenNums = removeOdd nums
  in length evenNums
```

Where Binding

```
fancySeven = 2 * a + 1
  where a = 3
```

Where Binding

```
fancyNine = x + y
  where x = 4
        y = 5
```

Where vs Let Binding

- ``Where'' goes with a function definition

```
fancyTen = 2 * (a + 1 where a = 4)
```

```
fancyTen = 2 * (let a = 4 in a + 1)
```

Where vs Let Binding

- Where - top down
- Let - bottom up

Table of Contents

Whitespace

- Do not use tabs. *Ever*.

Whitespace

- Indent further when breaking expression onto another line

```
pairMax p = max (fst p)
                (snd p)
```

```
pairMax p = max (fst p)
             (snd p)
```

```
pairMax p = max (fst p)
  (snd p)
```

Whitespace

- Line up variable bindings

```
fancyNine =
  let x = 4
      y = 5
  in x + y
```

```
fancyNine =
  let x = 4
    y = 5
  in x + y
```

Table of Contents

Lazy Function Evaluation

```
foo(alpha(1),beta(2));
```

```
foo (alpha 1) (beta 2)
```

Lazy Infinite Lists

```
intsFrom n = n : (intsFrom (n+1))
ints = intsFrom 1
```

Table of Contents

Summary

- Calling and defining functions
- Pure Functions
- Recursion
- Lists
- Tuples
- Pattern Matching
- Local Bindings
- Whitespace
- Lazy Function Evaluation