CS 314 Lecture 11

Functional programming and Haskell

(adapted from Brent Yorgey's CIS 194)

February 28, 2019

Typing

	Static	Dynamic
Imperative	C, Java	Python
Functional	Haskell, ML	Scheme

Haskell resources

- Learn You a Haskell for Great Good
- Programming in Haskell (Hutton, 2nd ed.)
- Haskell homepage
- Hoogle – API search
- Hackage – packages

Haskell

What is Haskell?

- Functional
- Pure
- Lazy
- Statically typed

Functional

- Functions are values
- Focus on evaluating expressions rather than executing instructions

Pure

Expressions are *referentially transparent*:

- No mutation
- No side effects
- Same function + same arguments = same value

Pure

This allows for:

- Equational reasoning – replacing equals by equals
- Parallelism – expressions don't affect each other
- Easier debugging?

Laziness

Expressions aren't evaluated until their results are needed

- Easy to define new "syntax"
- Infinite data structures
- Easy to compose functions together

But it complicates understand the time/space usage of your code.

Statically typed

Every expression has a type, checked at compile-time.

- Type inference
- Helps with design
- Helps with debugging
- Makes code easier to read and understand

Don't repeat yourself

Haskell is very good at abstraction.

- Algebraic data types
- Polymorphism
- Type classes
- Monoids, functors, monads, ...

"Wholemeal programming"

```
int acc = 0;
for (int i = 0; i < lst.length; i++) {
    acc = acc + 3 * lst[i];
}
```

- Lots of low-level details
- Combines two operations (times 3 and sum)

“Wholemeal programming”

In Scheme:

```
(reduce + (map (lambda (x) (* 3 x)) lst) 0)
```

“Wholemeal programming”

Same idea in Haskell:

```
sum (map (3*) lst)
```

Running haskell

Haskell can be either interpreted or compiled:

- `ghci`
  - `:l foo` – load foo.hs
  - `main` – evaluate main
  - `:r` – reload current file
  - `:q` – quit
- `ghc`
  - `ghc foo.hs` – compile foo.hs
  - `./foo` – run foo
- `runhaskell foo.hs`
- Jupyter

Hello world

In helloworld.hs:

```haskell
main = putStrLn "Hello world!"
```

Variables

```haskell
-- this is a comment

{- this is also
   a comment -}

x :: Int -- x has type Int
x = 3
```

Variables

Variables are immutable. This is illegal:

```
x = 3
x = 4
```

= is like mathematical equality, not assignment!

Variables

What does this do?

```
y :: Int
y = y + 1
```

Types

- Int (42)
- Integer (12345678909872184652998347212983498723 4)
- Float (3.14)
- Double (3.14)
- Bool (True, False)
- Char ('a', 'b') – Unicode
- String – a list of Chars

Arithmetic

```haskell
ex01 = 3 + 2
ex02 = 19 - 27
ex03 = 2.35 * 8.6
ex04 = 8.7 / 3.1
ex05 = mod 19 3
ex06 = 19 `mod` 3 -- backticks make mod infix
ex07 = 7 ^ 222
ex08 = (-3) * (-7) -- negative numbers should be
    written with parentheses
```

Arithmetic

Haskell doesn't do implicit type conversion. This doesn't work:

```
x :: Int
x = 3

y :: Integer
y = 4

z = x + y
```

Arithmetic

Use `fromIntegral` to convert from Int or Integer to another numeric type:

```
x :: Int
x = 3

y :: Integer
y = 4

z = (fromIntegral x) + y
```

Arithmetic

To convert floating-point to an integer type:

- round
- floor
- ceiling

Arithmetic

Division does floating-point division and the operands must be floating-point values.

For integer division, use `div`:

```
ex09 = i1 `div` i2
ex10 = 12 `div` 5
```

Boolean logic

```haskell
ex11 = True && False
ex12 = not (False || True)
```

Equality

Compare for equality with == and /=, or the usual ordering relations <, <= >, >=.

```
ex13 = ('a' == 'a')
ex14 = (16 /= 3)
ex15 = (5 > 3) && ('p' <= 'q')
ex16 = "Haskell" > "C++"
```

If expressions

```
if 1 < 2
then "yes"
else "no"
```

The else part must be present!

Functions

```haskell
abs :: Int -> Int
abs n = if n >= 0 then n else (-n)
```

Functions

We can write functions by cases:

```haskell
-- Compute the sum of the integers from 1 to n.
sumtorial :: Integer -> Integer
sumtorial 0 = 0
sumtorial n = n + sumtorial (n-1)
```

Functions

Choices can also be made using Boolean expressions ("guards"):

```
collatz :: Integer -> Integer
collatz n
  | n `mod` 2 == 0 = n `div` 2
  | otherwise      = 3*n + 1
```

Functions

We can abstract out the evenness check:

```
isEven :: Integer -> Bool
isEven n
  | n `mod` 2 == 0 = True
  | otherwise      = False
```

Functions

A better version:

```
isEven :: Integer -> Bool
isEven n = n `mod` 2 == 0
```

Functions

Then we can use this function in collatz:

```
collatz :: Integer -> Integer
collatz n
  | isEven n  = n `div` 2
  | otherwise = 3*n + 1
```

Pairs

We can pair things together like so:

```
p :: (Int, Char)
p = (3, 'x')
```

Pairs

The elements of a pair can be extracted again with pattern matching:

```
sumPair :: (Int,Int) -> Int
sumPair (x,y) = x + y
```

Functions

To apply a function to some arguments, just list the arguments after the function, separated by spaces, like this:

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
ex17 = f 3 17 8
```

Functions

Function application has higher precedence than any infix operators.

This is probably incorrect:

```
f 3 n+1 7
```

It parses as:

```
(f 3 n) + (1 7)
```

Instead, you have to use parentheses:

```
f 3 (n+1) 7
```

Lists

```haskell
nums, range, range2 :: [Integer]
nums   = [1,2,3,19]
range  = [1..100]
range2 = [2,4..100]
```

Lists

Haskell also has list comprehensions:

```haskell
xs = [ x^2 | x <- [1..10] ]
```

In Python:

```python
[ x**2 for x in range(1, 11) ]
```

Strings

Strings are just lists of characters

```haskell
hello1 :: [Char]
hello1 = ['h', 'e', 'l', 'l', 'o']

hello2 :: String
hello2 = "hello"

helloSame = hello1 == hello2
```

Constructing lists

```
emptyList = []
```

We can build larger lists using ":" (cons):

```
ex18 = 1 : []
ex19 = 3 : (1 : [])
ex20 = 2 : 3 : 4 : []

ex21 = [2,3,4] == 2 : 3 : 4 : []
```

Constructing lists

```haskell
-- Generate the sequence of collatz iterations from
    a starting number.
collatzSeq :: Integer -> [Integer]
collatzSeq 1 = [1]
collatzSeq n = n : collatzSeq (collatz n)
```

Functions on lists

```
-- Compute the length of a list of Integers.
intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (x:xs) = 1 + intListLength xs
```

Note that we never use the x on the last line.

Functions on lists

```haskell
-- Compute the length of a list of Integers.
intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (_:xs) = 1 + intListLength xs
```

We can replace unused variables with a placeholder "_".

Patterns

We can also use nested patterns:

```haskell
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo []         = []
sumEveryTwo (x:[])     = [x]
sumEveryTwo (x:(y:zs)) = (x + y) : sumEveryTwo zs
```

Combining functions

```
-- The number of collatz steps needed to reach 1
-- from some number
collatzLen :: Integer -> Integer
collatzLen n = intListLength (collatzSeq n) - 1
```

Error messages

Haskell error messages can be a bit scary at first, but they're actually quite useful.

```
> 'x' ++ "foo"

<interactive>:1:1:
    Couldn't match expected type ‘[Char]’ with
                           actual type ‘Char’
    In the first argument of ‘(++)’, namely 'x'
    In the expression: 'x' ++ "foo"
    In an equation for ‘it’: it = 'x' ++ "foo"
```