

Save-the-word Haskell

Chris Lin

July 23, 2016

1. The easiest way: Haskell platform

(a) GHC: the most widely used Haskell compiler.

How to use:

- Start (use them both, one by one):
`ghci`
`:set prompt "ghci> "`
- Load up a file (provided **myfunctions.hs**):
`:l myfunctions`
- Reload:
`:r`

2. Basic knowledge

(a) Always surround a negative number with parentheses.

(b) inequality symbol:

`/=`

(watch out for the difference between `4` and `"4"`)

(c) function

- Functions are called by writing the function name, a space and then the parameters, separated by spaces. For examples,
`min 9 10`
So,
`bar (bar 3)` means `bar(bar(3))` in C.
And there is no `bar(bar 3)` in Haskell.
- Function application has the highest precedence, which means these two statements are equivalent:
`succ 9 + max 5 4 + 1`
`(succ 9) + (max 5 4) + 1`
- If a function takes two parameters, we can also call it as an infix function by surrounding it with backticks.
`div 92 10`
`92 `div` 10`
- Write your own functions:
 - how to make functions (contents in **myfilename.hs**):
`doubleMe x = x + x`

- how to make use of it:

```
:l myfilename
doubleMe 9
```

- some examples:

```
doubleMe x = x + x
doubleUs x y = x*2 + y*2
```

(by having these, we can also run:

```
doubleUs 28 88 + doubleMe 123)
```

(we can also redefine the function `doubleUs` as:

```
doubleUs x y = doubleMe x + doubleMe y)
```

- Functions in Haskell don't have to be in any particular order, so it doesn't matter if you define `doubleMe` first and then `doubleUs` or if you do it the other way around.

- if statement:

```
doubleSmallNumber x = if x > 100
.....then x
.....else x*2
```

(Each '.' indicates a space. Because I fail to create spaces :P)

* the `else` part is mandatory in Haskell.

* if statement in Haskell is an expression:

```
doubleSmallNumber' x = (if x > 100 then x else x*2)
+ 1
```

notes: That apostrophe (') doesn't have any special meaning. It's ok in a function name. We usually use ' to either denote a strict version of a function (one that isn't lazy) or a slightly modified version of a function or a variable.

- what is more:

* Functions can't begin with uppercase letters.

(d) lists

- list basic

- elements need to be of the same type

- make a list:

```
let lostNumbers = [4,8,15,16,23,42]
```

(Doing `let a = 1` inside GHCi is the equivalent of writing `a = 1` in a script and then loading it.)

- strings are lists of characters

- putting two lists together:

* [1,2,3,4] ++ [9,10,11,12]
(take a while if the left one is too big)

* 'A':" SMALL CAT"
(instantaneous)

* ':' takes a number and a list of numbers or a character
and a list of characters, whereas '++' takes two lists.

(So if you're adding an element to the end of a list with
'++', you have to surround it with square brackets so it
becomes a list.)

([1,2,3] means 1:2:3:[] or 1:2:[3])

- [], [[]] and [[] , [] , []] are different:
an empty list;
a list that contains one empty list;
a list that contains three empty lists.
- access an element by index (start from 0):
"Steve Buscemi" !! 6
[9.4,33.2,96.2,11.2,23.25] !! 1
- The lists within a list can be of different lengths but they
can't be of different types.
- Lists can be compared if the stuff they contain can be com-
pared, and they are compared in lexicographical order from
left to right.
[3,4,2] > [3,4]
- Basic functions that operate on lists:

head, last, tail, init
Be careful not to use them on empty lists.

length, null, reverse, take, drop, maximum, minimum, sum,
product, elem

- Ranges

- examples:
[1..20]
['K'..'Z']
[3,6..20] (end up with 18)
[20,19..1]
- **Don't** use floating point numbers in ranges! Because they
are not completely precise.
- Can also use ranges to make infinite lists by just not speci-
fying an upper limit:
take 24 [13,26..
take 10 (repeat 5)

```
take 10 (cycle [1,2,3])
```

(Because Haskell is lazy, it won't try to evaluate the infinite list immediately because it would never finish. It'll wait to see what you want to get out of that infinite lists.)

(If you want exact number of the same element in a list, you will need `replicate` instead of `repeat`, like: `replicate 3 10`.)

- list comprehension:

```
- [x*2 | x <- [1..10]]
- [x*2 | x <- [1..10], x*2 >= 12]
- [ x | x <- [50..100], x `mod` 7 == 3]
- boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!"
  | x <- xs, odd x]
  boomBangs [7..13]
- [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
- let nouns = ["hobo","frog","pope"]
  let adjectives = ["lazy","grouchy","scheming"]
  [adjective ++ " " ++ noun | adjective <- adjectives,
  noun <- nouns]
(notice the order of the result)
- length' xs = sum [1 | _ <- xs]
(_ means that we don't care what we'll draw from the list
anyway)
- removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
- Nested list comprehensions:
  let xxs = [[1,3,5,2,3,1,2,4,5], [1,2,3,4,5,6,7,8,9],
  [1,2,4,2,1,6,3,1,3,2,3,6]]
  [ [ x | x <- xs, even x ] | xs <- xxs]
(You can write list comprehensions across several lines. )
```

- Tuples

- diff between lists and tuples:
 - * Lists:
 - of type of its components
 - num of components can be infinite.
 - you can compare two lists of different sizes.
 - * Tuples:
 - type depends on the num and types of the components.
 - cannot be infinite.
 - components can be of diff types.
 - tuples can be compared with each other if their components can be compared. But you can't compare two tuples of different sizes
 - * examples:

```

· ok: [[1,2], [8,11], [4,5] ]
· ok: [[1,2], [8,11,5], [4,5]]
· ok: [(1,2), (8,11), (4,5)]
· wrong: [(1,2), (8,11,5), (4,5)]
· wrong: [(1,2),("One",2)]
· ok: ("Christopher", "Walken", 55)
    (so tuples can also contain lists)
- zip
  examples:
  * zip [1,2,3,4,5] [5,5,5,5,5]
  * zip [1 .. 5] ["one", "two", "three", "four", "five"]
  * zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
  * zip [1..] ["apple", "orange", "cherry", "mango"]
  * let rightTriangles' = [ (a,b,c) | c <- [1..10] ,
    b <- [1..c] , a <- [1..b] , a^2 + b^2 == c^2,
    a+b+c == 24]

```

3. Syntax in Functions

- Pattern matching
 - used in inputs
 - kinda like “if-else”
 - can be used on tuples
 - example:


```
length' [] = 0
length' (_,xs) = 1 + length' xs
```
 - binding a pattern to a name:


```
capital all@(x:xs) = "The first letter of " ++ all ++ "
is " ++ [x]
```
 - can't use ++ in pattern matches
- GUARDS:


```
bmiTell weight height
....| weight / height ^ 2 <= 18.5 = "You're underweight, you
emo, you!"
....| weight / height ^ 2 <= 25.0 = "You're supposedly normal.
Pffft, I bet you're ugly!"
....| weight / height ^ 2 <= 30.0 = "You're fat! Lose some
weight, fatty!"
....| otherwise = "You're a whale, congratulations!"
```

 (again, each '.' indicates a space)
- GUARDS and WHERE BINDING:


```
bmiTell weight height
....| bmi <= skinny = "You're underweight, you emo, you!"
....| bmi <= normal = "You're supposedly normal. Pffft, I
bet you're ugly!"
....| bmi <= fat = "You're fat! Lose some weight, fatty!"
....| otherwise = "You're a whale, congratulations!"
```

- ```
....where bmi = weight / height ^ 2
.....skinny = 18.5
.....normal = 25.0
.....fat = 30.0
```
- define function in WHERE block

```
calcBmis xs = [bmi w h | (w, h) <- xs]
....where bmi weight height = weight / height ^ 2
```
  - LET BINDING
    - Where bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards.
    - Let bindings let you bind to variables anywhere and are expressions themselves, but are very local, so they don't span across guards.
    - example 1:

```
cylinder r h =
....let sideArea = 2 * pi * r * h
.....topArea = pi * r^2
....in sideArea + 2 * topArea
```
    - example 2:

```
4 * (let a = 9 in a + 1) + 2
```
    - example 3:

```
[let square x = x * x in (square 5, square 3, square 2)]
```
    - example 4:

```
(let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey ";
bar = "there!" in foo ++ bar)
```
    - example 5:

```
(let (a,b,c) = (1,2,3) in a+b+c) * 100
```
    - example 6:

```
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2,
bmi >= 25.0]
(We can't use the bmi name in the (w,h)<-xs part because it's
defined prior to the LET BINDING. The names defined in a
LET inside a list comprehension are visible to the output func-
tion(the part before the '|') and all predicates and sections that
come after of the binding.)
```
  - CASE EXPRESSION
    - pattern matching in function definitions is syntactic sugar for case expressions:

```
head' xs = case xs of [] -> error "No head for empty lists!"
.....(x:_) -> x
```
    - Whereas pattern matching on function parameters can be used in defining functions, case expressions can be used anywhere:

```
describeList xs = "The list is " ++ case xs of [] -> "empty."
.....[x] -> "a
singleton list."
.....xs -> "a
longer list."
```