Homework 0: Install the "Haskell Platform"

1. Google `"haskell platform"`
2. Download the installer
3. Run the installer

Haskell for Grownups

Bill Harrison

September 10, 2018

Table of Contents

Haskell Basics

- Modern (pure) lazy functional language
- Statically typed, supports type inference
- Compilers and interpreters:
  - http://www.haskell.org/implementations.html
  - GHC Compiler
  - GHCi interpreter
- A peculiar language feature: indentation matters
- Also: capitalization matters

Some Reference Texts

- *Programming in Haskell* by Graham Hutton.
  This is an excellent, step-by-step introduction to Haskell. Graham also has a lot of online resources (slides, videos, etc.) to go along with the book.
- *A Gentle Introduction to Haskell* by Hudak, Peterson, and Fasal.
  Available at `http://www.haskell.org/tutorial/`.
- *Learn You a Haskell for Good* by Miran Lipovaca.
  Highly amusing and informative; available online.
- *Real World Haskell* by Bryan O'Sullivan.
  Also available online (I believe). "Haskell for Working Programmers".
- *Course notes and Slides* by me.
- Google.

Question: What does this program do?

```
n = i;
a = 1;
while (n > 0) {
    a = a * n;
    n = n - 1;
}
```

Functions in Mathematics

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

Functions in Mathematics

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

What does this have to do with that?

```
n = i;
a = 1;
while (n > 0) {
    a = a * n;
    n = n - 1;
}
```

First Haskell Function

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

First Haskell Function

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n * (n-1)! & \text{if} \quad n > 0 \end{cases}$$

It's relationship to this Haskell function is apparent:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Hello World in C

```c
#include <stdio.h>
int main() {
  printf("hello␣world\n");
}
```

Hello World in Haskell

```
module HelloWorld where
helloworld :: IO ()
helloworld = print "Hello World"
```

Factorial Revisited

```c
#include <stdio.h>
int fac(int n) {
  if (n==0)
    { return 1; }
  else
    { return (n * fac (n-1)); }
}

int main() {
  printf("Factorial 5 = %d\n",fac(5));
  return 0;
}
```

Hello Factorial

```
#include <stdio.h>
int fac(int n) {
  printf("hello␣world");      // new
  if (n==0)
    { return 1; }
  else
    { return (n * fac (n-1)); }
}
    ...
```

Hello Factorial

```
#include <stdio.h>
int fac(int n) {
  printf("hello␣world");       // new
  if (n==0)
    { return 1; }
  else
    { return (n * fac (n-1)); }
}
    ...
```

(N.b., the type is the same)

int fac(int n) {...}

Hello Factorial in Haskell

```
fac :: Int -> IO Int -- the type changed
fac 0 = do print "hello␣world"
           return 1
fac n = do print "hello␣world"
           i <- fac (n-1)
           return (n * i)
```

Data Types + Functions = Haskell Programs

Haskell programming is both data type and functional programming!

- Arithmetic interpreter
  - data type:

    ```
    data Exp = Const Int | Neg Exp | Add Exp Exp
    ```

  - function:

    ```
    interp :: Exp -> Int
    interp (Const i)    = i
    interp (Neg e)      = - (interp e)
    interp (Add e1 e2) = interp e1 + interp e2
    ```

Data Types + Functions = Haskell Programs

Haskell programming is both data type and functional programming!

- Arithmetic interpreter
  - data type:

    ```
    data Exp = Const Int | Neg Exp | Add Exp Exp
    ```
  - function:

    ```
    interp :: Exp -> Int
    interp (Const i)    = i
    interp (Neg e)      = - (interp e)
    interp (Add e1 e2) = interp e1 + interp e2
    ```
- How do Haskell programs use data?
  - Patterns break data apart to access:
    "`interp (Neg e) =...`"
  - Functions recombine into new data:
    "`interp e1 + interp e2`"

Type Synonym

Type synonym: new name for an existing type; e.g.,

```
type String = [Char]
```

`String` is a synonym for the type `[Char]`.

Type synonyms can be used to make other types easier to read; e.g., given:

```
type Pos = (Int,Int)

origin     :: Pos
origin      = (0,0)
left       :: Pos -> Pos
left (x,y) = (x-1,y)
```

Parametric Polymorphism

Type synonyms can also have parameters

```
type Pair a = (a,a)

mult       :: Pair Int -> Int
mult (m,n) = m*n

copy       :: a -> Pair a
copy x      = (x,x)
```

Nesting Type Synonyms

Type declarations can be nested

```
type Pos   = (Int,Int)      -- GOOD

type Trans = Pos -> Pos     -- GOOD
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])    -- BAD
```

Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

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

Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

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

- `Bool` is a new type.
- `False` and `True` are called **constructors** for `Bool`.
- Type and constructor names begin with upper-case letters.
- Data declarations are similar to context free grammars.

New types can be used in the same way as built-in types

For example, given

```
data Answer = Yes | No | Unknown
```

New types can be used in the same way as built-in types

For example, given

```
data Answer = Yes | No | Unknown
```

We can define:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer -> Answer
flip Yes      = No
flip No       = Yes
flip Unknown = Unknown
```

Constructors with Parameters

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square          :: Float -> Shape
square n         = Rect n n
area            :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.
- Circle and Rect can be viewed as functions that construct values of type Shape:

```
-- Not a definition
Circle :: Float -> Shape
Rect   :: Float -> Float -> Shape
```

Not surprisingly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
safediv    :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Table of Contents

General form of a Haskell module

```
module ModuleName where
import L₁        -- imports
    ⋮
import Lₖ

data D₁ = ···   -- type decls
    ⋮
data Dₙ = ···

f₁ = ···        -- fun decls
    ⋮
fₘ = ···
```

- Order does not matter
- Modules, Types & constructors are always capitalized
- Module *ModuleName* stored in file, *ModuleName*.hs

Starting GHCi

```
bash-3.2> ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
*Prelude>
```

Loading a File into GHCi

```
*Prelude> :l ExtractList
[1 of 1] Compiling ExtractList (ExtractList.hs, interpreted)
Ok, modules loaded: ExtractList.
*ExtractList>
```

- `:l` is short for `:load`.
- Could type `ghci ExtractList` to load it at start up.

Checking Types in GHCi

```
*ExtractList> :t head
head :: [a] -> a
*ExtractList> :t tail
tail :: [a] -> [a]
*ExtractList> :t (:)
(:) :: a -> [a] -> [a]
```

- `:t` is short for `:type`.
- Can check the type of any function definition
  - The above are list functions defined in Prelude

Reloading and Quitting GHCi

```
*ExtractList> :r
Ok, modules loaded: ExtractList.
*ExtractList> :q
Leaving GHCi.
bash-3.2>
```

- `:r` is short for `:reload`.
- `:q` is short for `:quit`.
- Reload only recompiles the current module. Use it when you have only made changes to the current module—it's faster.
- Emacs Users: can use C-p, C-n, C-e, C-a, C-f at the GHCi prompt to cycle through and edit previous commands.

Table of Contents

Type-Driven Programming in Haskell

Types first, then programs

- Writing a function with type $A \to B$, then you have a lot of information to use for fleshing out the function.
- Why? Because the input type A — *whatever it happens to be* — has a particular form that determines a large part of the function itself.
- This is, in fact, the way that you should develop Haskell programs.

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

`Nat` is a new type, with constructors

```
Zero :: Nat
Succ :: Nat -> Nat
```

Note:

- A value of type `Nat` is either `Zero`, or of the form `Succ n` where n :: Nat. That is, `Nat` contains the following infinite sequence of values:

```
Zero
Succ Zero
Succ (Succ Zero)
      ⋮
```

Note:

- We can think of values of type Nat as natural numbers, where `Zero` represents `0`, and `Succ` represents the successor function `1+`.
- For example, the value

```
Succ (Succ (Succ Zero))
```

  represents the natural number

```
1 + (1 + (1 + 0))
```

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int          :: Nat -> Int
nat2int Zero      = 0
nat2int (Succ n) = 1 + nat2int n

int2nat          :: Int -> Nat
int2nat 0         = Zero
int2nat n         = Succ (int2nat (n - 1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add     :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

The recursive definition for add corresponds to the laws

$$0 + n = n$$

and

$$(1 + m) + n = 1 + (m + n)$$

The edit-compile-test-until-done paradigm

I'm guessing that this is familar to you

When I was a student—the process of writing a C program tended to follow these steps:

1. Create/edit a version of the <u>whole</u> program using a text editor.
2. Compile. If there were compilation errors, develop a hypothesis about what the causes were and start again at 1.
3. Run the program on some tests. Do I get what I expect? If so, then declare victory and stop; otherwise, develop a hypothesis about what the causes were and start again at 1.

An Exercise

- Write a function that
  1. takes a list of items,
  2. takes a function that returns either `True` or `False` on those items,
  3. and returns a list of all the items on which the function is true.
- This is called *filter*, and it's a built-in function in Haskell, but let me show you how I'd write it from scratch.
  - I call the function I'm writing "`myfilter`" to avoid the name clash with the built-in version.

Step 1. Figure out the type of the thing you're writing

- Think about the type of `filter` and write it down as a type specification in a Haskell module (called `Sandbox` throughout).
- With what I've said about `filter`, it takes a list of items—i.e., something of type [`a`].
- It also takes a function that takes an item—an `a` thing—and returns true or false—i.e., it returns a `Bool`. So, this function will have type `a` $\to$ `Bool`.
- ∴ the type should be:

```
myfilter :: [a] -> (a -> Bool) -> [a]
```

Step 2: Fill in the type template & Load the module.

- In this case, we have a function with two arguments. The second argument of type `a->Bool` does not have a matchable form like the first argument.
- This leaves us with:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f   = undefined
myfilter (x:xs) = undefined
```

Step 2: Fill in the type template & Load the module.

- In this case, we have a function with two arguments. The second argument of type `a->Bool` does not have a matchable form like the first argument.
- This leaves us with:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f   = undefined
myfilter (x:xs) = undefined
```

- Reloading module reveals an error in the template as typed:

```
> ghci Sandbox.hs
[1 of 1] Compiling Sandbox          ( Sandbox.hs, interpreted )
Sandbox.hs:6:1:
    Equations for ‘myfilter’ have different numbers of arguments
      Sandbox.hs:6:1-27
      Sandbox.hs:7:1-27
```

Step 2 (continued)

Debugging via Type-checking!

- Fix the error NOW! Forgot 2nd `f` param. in 2nd clause.
  - Type error just committed can be fixed *now*.
  - Alternative: wait until I have "first draft" of the whole program and check it.
  - Incremental approach: can check each new part of the program as I create it so that I'm not surprised by errors.
  - Identifying the source of error easier—i.e., it is the line you just typed.
- Fixing the error yields:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f     = undefined
myfilter (x:xs) f = undefined
```

Step 3: Fill in the clauses one-by-one reloading as you go.

The [] case is obvious because there is nothing to filter out:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f     = []
myfilter (x:xs) f = undefined
```

No problems with this last bit:

```
> ghci Sandbox.hs
[1 of 1] Compiling Sandbox
Ok, modules loaded: Sandbox.
*Sandbox>
```

Step 3 (continued).

- The second clause should only include `x` if `f x` is `True`; one way to write that is with an `if–then–else`:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f     = []
myfilter (x:xs) f = if f x
                     then x : myfilter f xs
                     else myfilter f xs
```

- Loading this into GHC reveals a problem:

```
> ghci Sandbox.hs
[1 of 1] Compiling Sandbox          ( Sandbox.hs, interpreted )
Sandbox.hs:8:46:
    Couldn't match expected type '[a]' with actual type 'a -> Bool'
    In the first argument of 'myfilter', namely 'f'
    In the second argument of '(:)', namely 'myfilter f xs'
    In the expression: x : myfilter f xs
Failed, modules loaded: none.
Prelude>
```

Step 3 (continued).

- This error occurs on line 8 of the module, which is the line "`then x : myfilter f xs`". GHCi is telling us that it expects that `f` would have type [`a`] but that it can see that `f` has type `a` $\rightarrow$ `Bool`. After a moment's pause, we can see that the order of the arguments is incorrect in both recursive calls. The corrected version works:

```
myfilter :: [a] -> (a -> Bool) -> [a]
myfilter [] f     = []
myfilter (x:xs) f = if f x
                      then x : myfilter xs f
                      else myfilter xs f
```