Haskell Crash Course Part I

From the Lambda Calculus to Haskell
What is Haskell?

A typed, lazy, purely functional programming language

Haskell = \(\lambda\)-calculus ++

- better syntax
- types
- built-in features
  - booleans, numbers, characters
  - records (tuples)
  - lists
  - recursion
  - ...
Programming in Haskell

Computation by Calculation

Substituting equals by equals
Computation via Substituting Equals by Equals

\[(1 + 3) \times (4 + 5)\]

\[
\begin{align*}
\text{\textbackslash \textbackslash} & \quad \text{-- subst } 1 + 3 = 4 \\
\end{align*}
\]

\[
\begin{align*}
4 \times (4 + 5) & \quad \text{-- subst } 4 + 5 = 9 \\
\end{align*}
\]

\[
\begin{align*}
4 \times 9 & \quad \text{-- subst } 4 \times 9 = 36 \\
\end{align*}
\]

\[
\begin{align*}
36 & \\
\end{align*}
\]
Equality-Substitution enables Abstraction via Pattern Recognition

\[ \text{pat} = \lambda x \rightarrow x \]

Abstraction via Pattern Recognition

Repeated Expressions

\[ \text{pat} = \lambda x y z \rightarrow x \ast (y + z) \]

\[ 31 \ast (42 + 56) \]

\[ \text{pat} 90 12 95 = 70 \ast (12 + 95) \]

\[ \text{pat} 90 68 12 = 90 \ast (68 + 12) \]

Recognize Pattern as \( \lambda \)-function

\[ \text{pat} = \lambda x y z \rightarrow x \ast (y + z) \]
Equivalent Haskell Definition

\[ \text{pat } x \ y \ z = x \times (y + z) \]

Function Call is Pattern Instance

\[ \begin{align*}
\text{pat } 31 \ 42 \ 56 & \Rightarrow 31 \times (42 + 56) \Rightarrow 31 \times 98 \Rightarrow 3038 \\
\text{pat } 70 \ 12 \ 95 & \Rightarrow 70 \times (12 + 95) \Rightarrow 70 \times 107 \Rightarrow 7490 \\
\text{pat } 90 \ 68 \ 12 & \Rightarrow 90 \times (68 + 12) \Rightarrow 90 \times 80 \Rightarrow 7200
\end{align*} \]

Key Idea: Computation is substitute equals by equals.

*Programming in Haskell*
Substitute Equals by Equals

That's it! *(Do not think of registers, stacks, frames etc.)*

Elements of Haskell
• Core program element is an **expression**
• Every *valid* expression has a **type** (determined at compile-time)
• Every *valid* expression reduces to a **value** (computed at run-time)

Ill-typed* expressions are rejected at *compile-time* before execution

• *like in* Java
• *not like* $\lambda$-calculus or Python

```
weirdo = 1 0  -- rejected by GHC
```

**Why are types good?**

• Helps with program *design*
• Types are contracts (ignore ill-typed inputs!)
• Catches errors early
• Allows compiler to generate code
• Enables compiler optimizations

The Haskell Eco-System

• Batch compiler: ghc Compile and run large programs

• Interactive Shell ghci Shell to interactively run small programs online (https://repl.it/languages/haskell)

• Build Tool stack Build tool to manage libraries etc.
Interactive Shell: ghci

$ stack ghci

:load file.hs
:type expression
:info variable
A Haskell Source File

A sequence of top-level definitions \( x_1, x_2, \ldots \)

- Each has type \( \text{type}_1, \text{type}_2, \ldots \)

- Each defined by expression \( \text{expr}_1, \text{expr}_2, \ldots \)

\[
\begin{align*}
  x_1 & :: \text{type}_1 \\
  x_1 & = \text{expr}_1 \\
  x_2 & :: \text{type}_2 \\
  x_2 & = \text{expr}_2
\end{align*}
\]
Basic Types
ex1 :: Int
ex1 = 31 * (42 + 56)  -- this is a comment

ex2 :: Double
ex2 = 3 * (4.2 + 5.6)  -- arithmetic operators "overloaded"

ex3 :: Char
ex3 = 'a'  -- 'a', 'b', 'c', etc. built-in `Char` values

ex4 :: Bool
ex4 = True  -- True, False are builtin Bool values

ex5 :: Bool
ex5 = False
QUIZ: Basic Operations

ex6 :: Int
ex6 = 4 + 5

ex7 :: Int
ex7 = 4 * 5

ex8 :: Bool
ex8 = 5 > 4

quiz :: ???
quiz = if ex8 then ex6 else ex7

What is the type of quiz?

A. Int
B. Bool
C. Error!
QUIZ: Basic Operations

ex6 :: Int
ex6 = 4 + 5 \Rightarrow 9

ex7 :: Int
ex7 = 4 * 5

ex8 :: Bool
ex8 = 5 > 4 \Rightarrow TRUE

quiz :: ???
quiz = if ex8 then ex6 else ex7
     TRUE
What is the value of quiz?

A. 9

B. 20

C. Other!

Function Types

In Haskell, a function is a value that has a type

\[ \text{In} \rightarrow \text{Out} \]

A function that
• takes *input* of type A
• returns *output* of type B

For example

```haskell
isPos :: Int -> Bool
isPos = \n -> (x > 0)
```

Define **function-expressions** using \ like in \(\lambda\)-calculus!

But Haskell also allows us to put the parameter on the *left*

```haskell
isPos :: Int -> Bool
isPos n = (x > 0)
```

(Meaning is **identical** to above definition with \(\n \rightarrow \ldots\) )
Multiple Argument Functions

A function that

- takes three inputs $A_1$, $A_2$, and $A_3$
- returns one output $B$ has the type

$A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow B$

For example

$\text{pat} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$\text{pat} = \lambda x \ y \ z \rightarrow x \ast (y + z)$

which we can write with the params on the left as

$\text{pat} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$\text{pat} \ x \ y \ z = x \ast (y + z)$
What is the type of quiz?

quiz :: ???

quiz x y = (x + y) > 0

A. Int -> Int
B. Int -> Bool
C. Int -> Int -> Int
D. Int -> Int -> Bool
E. (Int, Int) -> Bool
Function Calls

A function call is exactly like in the $\lambda$-calculus

\[
\begin{array}{c}
e_1 \ e_2
\end{array}
\]

where $e_1$ is a function and $e_2$ is the argument. For example

```python
>>> isPos 12
True
```

```python
>>> isPos (0 - 5)
False
```
Multiple Argument Calls

With multiple arguments, just pass them in one by one, e.g.

(((e e1) e2) e3)

For example

>>> pat 31 42 56
3038
EXERCISE

Write a function `myMax` that returns the maximum of two inputs

```
myMax :: Int -> Int -> Int
myMax = ???
```

When you are done you should see the following behavior:

```
>>> myMax 10 20
20

>>> myMax 100 5
100
```
**EXERCISE**

Write a function `sumTo` such that `sumTo n` evaluates to $0 + 1 + 2 + \ldots + n$

\[
\text{sumTo} :: \text{Int} \rightarrow \text{Int}
\]

\[
\text{sumTo} \ n = \text{???} \quad 0 + 1 + 2 + \ldots + n
\]

When you are done you should see the following behavior:

\[
\begin{align*}
\text{>>> sumTo 3} & = 0 + 1 + 2 + 3 = 6 \\
\text{6} & \\
\text{>>> sumTo 4} & = 0 + 1 + 2 + 3 + 4 = 10 \\
\text{10} & \\
\text{>>> sumTo 5} & = 0 + 1 + 2 + 3 + 4 + 5 = 15 \\
\text{15} & 
\end{align*}
\]
How to Return **Multiple** Outputs?

**Tuples**

A type for packing \( n \) different kinds of values into a single "struct"

\((T_1, \ldots, T_n)\)

For example

\[
\begin{align*}
\text{FST} \ (\text{Pair apple banana}) \\
\Rightarrow \text{apple} \\
\text{SND} \ (\text{banana}) \\
\Rightarrow \text{banana}
\end{align*}
\]
tup1 :: ???
tup1 = ('a', 5)

tup2 :: (Char, Double, Int)
tup2 = ('a', 5.2, 7)

**QUIZ**

What is the type ??? of tup3?

tup3 :: ???
tup3 = ((7, 5.2), True)

A. (Int, Bool)
Extracting Values from Tuples

We can create a tuple of three values \( e_1, e_2, \) and \( e_3 \) ...

\[ \text{tup} = (e_1, e_2, e_3) \]

... but how to extract the values from this tuple?
Pattern Matching via case-of expressions

fst3 :: (t1, t2, t3) -> t1
fst3 t = case t of
  (x1, x2, x3) -> x1

snd3 :: (t1, t2, t3) -> t2
snd3 t = case t of
  (x1, x2, x3) -> x2

thd3 :: (t1, t2, t3) -> t3
thd3 t = case t of
  (x1, x2, x3) -> x3
**QUIZ**

What is the value of quiz defined as

\[
\text{tup2 :: (Char, Double, Int)} \\
\text{tup2} = ('a', 5.2, 7)
\]

\[
\text{snd3 :: (t1, t2, t3) -> t2} \\
\text{snd3} \ t = \text{case} \ t \ \text{of} \\
\quad (x1, x2, x3) \rightarrow x2
\]

\[
\text{quiz} = \text{snd3} \ \text{tup2}
\]

A. 'a'

B. 5.2

C. 7

D. ('a', 5.2)

E. (5.2, 7)
Lists

Unbounded Sequence of values of type \( T \)

For example

\[
\text{chars :: [Char]}
\]
\[
\text{chars} = ['a', 'b', 'c']
\]

\[
\text{ints :: [Int]}
\]
\[
\text{ints} = [1, 3, 5, 7]
\]

\[
\text{pairs :: [(Int, Bool)]}
\]
\[
\text{pairs} = [(1,True), (2,False)]
\]
QUIZ

What is the type of things defined as

```
things :: ???
things = [[1], [2, 3], [4, 5, 6]]
```

A. [Int]
B. ([Int], [Int], [Int])
C. [(Int, Int, Int)]
D. [[Int]]
E. List
List’s Values Must Have The SAME Type!

The type \([T]\) denotes an unbounded sequence of values of type \(T\)

Suppose you have a list

\[
\text{oops} = [1, 2, 'c']
\]

There is no \(T\) that we can use

- As last element is not \(\text{Int}\)
- First two elements are not \(\text{Char}\)!

Result: Mysterious Type Error!
Constructing Lists

There are two ways to construct lists

```
"Nil" [] -- creates an empty list
"Cons" h:t -- creates a list with "head" 'h' and "tail" t
```

For example
>>> 3 : []
[3]

>>> 2 : (3 : [])
[2, 3]

>>> 1 : (2 : (3 : []))
[1, 2, 3]

Cons Operator: is Right Associative

\[ x_1 : x_2 : x_3 : x_4 : t \text{ means } x_1 : (x_2 : (x_3 : (x_4 : t))) \]

So we can just avoid the parentheses.

Syntactic Sugar

Haskell lets you write \([x_1, x_2, x_3, x_4]\) instead of \(x_1 : x_2 : x_3 : x_4 : []\)
Functions Producing Lists

Lets write a function `copy3` that

- takes an input `x` and
- returns a list with *three* copies of `x`

```
copy3 :: ???
copy3 x = ???
```

When you are done, you should see the following

```
>>> copy3 5
[5, 5, 5]

>>> copy3 "cat"
["cat", "cat", "cat"]
```
*Let's write some Functions*

A Recipe (https://www.htdp.org/)

Step 1: Write some tests

Step 2: Write the type

Step 3: Write the code
**PRACTICE: Clone**

Write a function `clone` such that `clone n x` returns a list with `n` copies of `x`.

1. Tests

When you are done you should see the following behavior
2. Types

clone :: ???

3. Code

clone n x = ???
How does \texttt{clone} execute?

(Substituting equals-by-equals!)

\begin{verbatim}
clone 3 100
   => ???
\end{verbatim}
EXERCISE: Range

Write a function range such that range i j returns the list of values [i, i+1, ..., j]

range :: ???
range i j = ???

1. Tests
>>> range 4 3
[]

>>> range 3 3
[3]

>>> range 2 3
[2, 3]

>>> range 1 3
[1, 2, 3]

>>> range 0 3
[0, 1, 2, 3]

2. Type

range :: ???

3. Code

range = ???
Functions Consuming Lists

So far: how to produce lists.

Next how to consume lists!
EXERCISE

Let's write a function `firstElem` such that `firstElem xs` returns the first element `xs` if it is a non-empty list, and `0` otherwise.

**HINT:** How to extract values from a list?

1. Tests

When you are done you should see the following behavior:
>>> firstElem []
0

>>> firstElem [10, 20, 30]
10

>>> firstElem [5, 6, 7, 8]
5

2. Type

firstElem :: ???

3. Code

firstElem = ???
QUIZ

Suppose we have the following mystery function

\[
\text{mystery} :: [a] \rightarrow \text{Int} \\
\text{mystery}\ l = \text{case } l\ \text{of} \\
\quad [] \rightarrow 0 \\
\quad (x:xs) \rightarrow 1 + \text{mystery} \ xs
\]

What does \text{mystery} [10, 20, 30] evaluate to?

A. 10  
B. 20  
C. 30  
D. 3  
E. 0
EXERCISE: Summing a List

Write a function sumList such that sumList [x₁, ..., xₙ] returns x₁ + ... + xₙ

1. Tests

When you are done you should get the following behavior:
2. Type

\[
sumList :: [Int] \rightarrow \text{Int}
\]

3. Code

\[
\text{sumList } = ???
\]

**Functions on lists: take**

Let’s write a function to take first \( n \) elements of a list \( xs \).
1. Tests

```none
-- >>> ???
```

2. Type

```haskell
-- | Length of the list
length :: [t] -> Int

-- | Append two lists
(++) :: [t] -> [t] -> [t]

-- | Are two lists equal?
(==) :: [t] -> [t] -> Bool
```

You can search for library functions on Hoogle (https://www.haskell.org/hoogle/)!
**3. Code**

```
```
haskell

```haskell
take = ???
```
You can search for library functions on Hoogle (https://www.haskell.org/hoogle/)

Recap

- Core program element is an **expression**
- Every *valid* expression has a **type** (determined at compile-time)
- Every *valid* expression reduces to a **value** (computed at run-time)
Execution

- Basic values & operators
- Execution / Function Calls just **substitute equals by equals**
- Pack data into *tuples & lists*
- Unpack data via *pattern-matching*

(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)