Haskell Crash Course Part I

From the Lambda Calculus to Haskell

What is Haskell?

A typed, lazy, purely functional programming language

Haskell = λ-calculus ++

- better syntax ✓
- types

\[
\begin{align*}
\text{one} &= \lambda f \ x \ f \ x \\
\text{two} &= \lambda f \ x \ f \ (f \ x) \\
\end{align*}
\]
• built-in features
  ○ booleans, numbers, characters
  ○ records (tuples)
  ○ lists
  ○ recursion
  ○ ...

---

**Programming in Haskell**

Computation by Calculation

Substituting equals by equals

\[
\begin{align*}
(2+3) \times (5-1) & \\
= 5 \times (5-1) & \\
= 5 \times 4 & \\
= 20 &
\end{align*}
\]
Computation via Substituting Equals by Equals

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

-- subst \(1 + 3 = 4\)

\[\Rightarrow 4 \times (4 + 5)\]

-- subst \(4 + 5 = 9\)

\[\Rightarrow 4 \times 9\]

-- subst \(4 \times 9 = 36\)

\[\Rightarrow 36\]

Computation via Substituting Equals by Equals

Equality-Substitution enables Abstraction via Pattern Recognition
Abstraction via Pattern Recognition

Repeated Expressions

1. $31 \times (42 + 56)$
2. $70 \times (12 + 95)$
3. $90 \times (68 + 12)$

Recognize Pattern as $\lambda$-function

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

Equivalent Haskell Definition

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

Function Call is Pattern Instance

\[
\text{pat} \; 31 \; 42 \; 56 = \Rightarrow 31 \times (42 + 56) = \Rightarrow 31 \times 98 = \Rightarrow 3038
\]
\[
\text{pat} \; 70 \; 12 \; 95 = \Rightarrow 70 \times (12 + 95) = \Rightarrow 70 \times 107 = \Rightarrow 7490
\]
\[
\text{pat} \; 90 \; 68 \; 12 = \Rightarrow 90 \times (68 + 12) = \Rightarrow 90 \times 80 = \Rightarrow 7200
\]

Key Idea: Computation is substitute equals by equals.
Programming in Haskell

Substitute Equals by Equals

That's it! \textit{(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** λ-calculus or Python ...

```
weirdo = \( \begin{pmatrix} 1 & 0 \end{pmatrix} \)  -- 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'

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

ex7 :: Int
ex7 = 4 * 5

ex8 :: Bool
ex8 = 5 > 4

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

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

A -> B
A function that

- takes input of type \( A \)
- returns output of type \( B \)

For example

\[
\text{isPos} :: \text{Int} \rightarrow \text{Bool} \\
\text{isPos} = \ n \rightarrow (x > 0)
\]

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

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

\[
\text{isPos} :: \text{Int} \rightarrow \text{Bool} \\
\text{isPos} \ n = (x > 0)
\]

(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

```haskell
pat :: Int -> Int -> Int -> Int
pat = \x y z -> x * (y + z)
```

which we can write with the params on the left as

```haskell
pat :: Int -> Int -> Int -> Int
pat x y z = x * (y + z)
```

---

**QUIZ**

What is the type of `quiz`?

```haskell
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

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

```haskell
>>> isPos 12
True

>>> 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

```haskell
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 \( 0 + 1 + 2 + \ldots + n \)

`sumTo :: Int -> Int`

`sumTo n = ???`

When you are done you should see the following behavior:

````
>>> sumTo 3
6
>>> sumTo 4
10
>>> sumTo 5
15
```