Haskell Crash Course Part I

From the Lambda Calculus to Haskell

+ types
+ compiler
+ modules/package

Programming in Haskell

Computation by Calculation

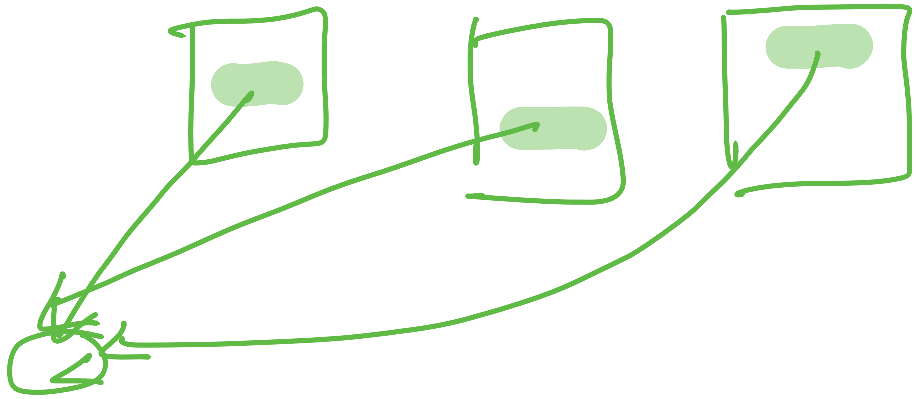
Substituting equals by equals

Computation via Substituting Equals by Equals

```
    (1 + 3) * (4 + 5)
                        -- subst 1 + 3 = 4

==>       4 * (4 + 5)
                        -- subst 4 + 5 = 9

==>       4 * 9
                        -- subst 4 * 9 = 36

==>       36
```

Computation via Substituting Equals by Equals

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



Abstraction via Pattern Recognition

Repeated Expressions

```
31 * (42 + 56)
70 * (12 + 95)
90 * (68 + 12)
```

≡ pat 31 42 56

≡ pat 70 12 95

= pat 90 68 12

Recognize Pattern as λ-function

```
let pat = \x y z -> x * ( y + z )
```

pat x y z = x*(y+z)

pat 31

Equivalent Haskell Definition

```
pat   x y z =  x  * ( y + z )
```

Function Call is **Pattern Instance**

```
pat 31 42 56 =*> 31 * (42 + 56) =*> 31 * 98  =*> 3038
pat 70 12 95 =*> 70 * (12 + 95) =*> 70 * 107 =*> 7490
pat 90 68 12 =*> 90 * (68 + 12) =*> 90 * 80  =*> 7200
```

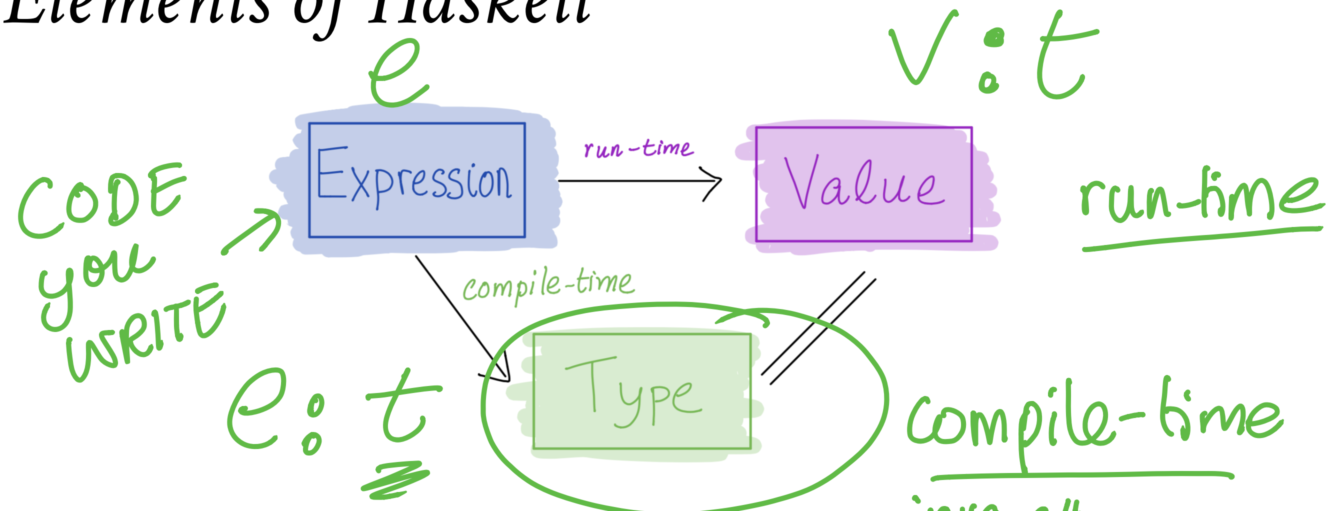
Key Idea: Computation is *substitute* equals by equals.

Programming in Haskell

Substitute Equals by Equals

Thats it! (*Do not* think of registers, stacks, frames etc.)

Elements of Haskell



java, c#

- 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~~...

↳ mypy

js → ts

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
```

VSCODE

A Haskell Source File

A sequence of **top-level definitions** `x1` , `x2` , ...

- Each has *type* `type_1` , `type_2` , ...
- Each defined by *expression* `expr_1` , `expr_2` , ...

```
x_1 :: type_1
x_1 = expr_1

x_2 :: type_2
x_2 = expr_2

.
.
.
```

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

QUIZ

```
ex6 :: Int
ex6 = 4 + 5
```

```
ex7 :: Int
ex7 = 4 * 5
```

```
ex8 :: Bool
ex8 = 5 > 4
```

BOOL is TRUE

```
quiz :: ???
quiz = (if ex8 then ex6 else ex7)
```

int

BOOL int int

What is the *type* of `quiz` ?

Bool is false

A. `Int`

B. `Bool`

C. Error!

QUIZ: Basic Operations

Quiz

```
ex6 :: Int
ex6 = 4 + 5      = 9

ex7 :: Int
ex7 = 4 * 5      = 20

ex8 :: Bool
ex8 = 5 > 4      = TRUE
```

IF FALSE e_1 e_2 => e_2

IF TRUE e_1 e_2 => e_1

```
~~quiz :: ???~~
                           ✓
quiz = if ex8 then ex6 else ex7
          TRUE     9        20
```

$\underbrace{\text{ex8 ? ex6 : ex7}}$

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

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

x

Define **function-expressions** using `\` like in λ-calculus!

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

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

(Meaning is **identical** to above definition with `\n -> ...`)

Multiple Argument Functions

A function that

- takes three *inputs* `A1`, `A2` and `A3`

- returns one *output* `B` has the type

```
A1 -> A2 -> A3 -> B
```

For example

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

which we can write with the params on the *left* as

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

QUIZ

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 λ-calculus

```
(e1 e2)
```

where `e1` is a function and `e2` is the argument. For example

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

exp exp

```
>>> isPos (0 - 5)
False
```

Multiple Argument Calls

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

```
(((e e1) e2) e3)
```

$(((e\ \ e_1)\ e_2)\ \ e_3\ )$

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
```

How to Return Multiple Outputs?

Tuples

A type for packing `n` different kinds of values into a single "struct"

```
(T1,..., Tn)
```

For example

(Char, Int)

```
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)`

B. `(Int, Double, Bool)`

C. `(Int, (Double, Bool))`

D. `((Int, Double), Bool)`

E. `(Tuple, Bool)`