Monad P1 : Side Effects (1A)

Young Won Lim
3/26/19

Please send corrections (or suggestions) to youngwlim@hotmail.com.
This document was produced by using LibreOffice.

Based on

Variables and functions
https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Purity
https://wiki.haskell.org/Functional_programming#Purity

Variables

Imperative programming:

- **variables** as changeable locations in a computer's memory
- **imperative programs** explicitly commands (instructs) the computer what to do

functional programming

- a way to think in higher-level mathematical **terms**
- defining how **variables** **relate** to one another
- the **compiler** will **translate** these **functions** and **variables** to **instructions** so that the computer can process.

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Haskell Language Features (I)

Haskell Functional Programming (I)

- **Immutability**
- **Recursive Definition : only in functions**
- **No Data Dependency**

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Redefinition : not allowed

imperative programming:

after setting **r = 5** and then changing it to **r = 2**.

r = 5

r = 2

Hakell programming:

an error: "multiple declarations of **r**".

within a given scope, a **variable** in Haskell

are defined only once and cannot change,

like variables in mathematics.

r = 5

~~**r = 2**~~

no mutation

in Haskell

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Variables in a file

Immutable:

they can change only based on

the data we enter to run the program.

We cannot define **r** two ways in the same code,

but we could change the value by changing the file

Vars.hs

```
a = 100
r = 5
pi = 3.14159
e = 2.7818
```

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

No Mutation

```
*Main> r = 33
<interactive>:12:3: parse error on input ‘=’


$ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> r = 333
<interactive>:2:3: parse error on input ‘=’
Prelude>
```

let r = 33

No mutation, Immutable

let r = 33

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Loading a variable definition file

```
$ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load Var1.hs
[1 of 1] Compiling Main             ( var.hs, interpreted )
Ok, modules loaded: Main.
*Main> r
5
*Main> :t r
r :: Integer
*Main>


*Main> :load Var2.hs
[1 of 1] Compiling Main             ( var2.hs, interpreted )
Ok, modules loaded: Main.
*Main> r
55
```

:load Var1.hs

:load Var1.hs

definition with initialization

Var1.hs file

```
r = 5
x = 1
y = 3.14
…
```

Var2.hs file

```
r = 55
x = 1
y = 3.14
…
```

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Incrementing by one

imperative programming:

incrementing the variable r

(**updating** the value in memory)

r = r + 1

Hakell programming:

No **compound assignment** like operations

if **r** had been defined with any value beforehand,

then **r = r + 1** in Haskell would bring an error message.

multiple definition not allowed

r = 3
~~**r = r + 1**~~

the expression **r = r + 1** is a **recursive definition**

allowed in a **function** definition

add1 x = x + 1

r = 3
~~**r = add1 r**~~

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Arguments and parameters of a function

binding an **argument** and a **parameter** of a **function**

add1 x = x + 1 → **101**

add1 100

x (parameter)

100 (argument)

add1 x = x + 1

r = 100

add1 r

~~**r = add1 r**~~

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Recursive Definition

Hakell programming:

a **recursive definition** of **r**

(defining it in terms of itself)

a += b	**(a = a + b)**
a -= b	**(a = a – b)**
a *= b	**(a = a * b)**
a /= b	**(a = a / b)**

No **compound assignment** like operations are allowed

if **a** had been defined with any value beforehand,

then **a = a + b** in Haskell would **multiply defined**

recursive function
factorial 0 = 1
factorial n = n * factorial (n – 1)

non-recursive function
add1 x = x + 1

recursive definitions are allowed
only in function definition

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Simulating imperative codes

The most primitive way of **x = v** is to use a **function** taking **x** as a **parameter**, and pass the **argument** **v** to that function.

x = v

```
i = s = 0;                          // sum 0..100
while (i <= 100) {
   s = s+i;
   i++;
}
return s;
```

```
sum = f 0 0                         -- the initial values
  where
  f i s | i <=100     = f (i+1) (s+i)     -- increment i, augment s
        | otherwise   = s                 -- return s at the end
```

```
i = (i+1)
s = (s+i)
```

This code is not pretty functional programing code,

but it is simulating imperative code

https://stackoverflow.com/questions/43525193/how-can-i-re-assign-a-variable-in-a-function-in-haskell

No Data Dependency

```
y = x * 2
x = 3
```

```
x = 3
y = x * 3
```

Hakell programming:

because the values of variables do not change

variables can be defined in any order

no mandatory : "**x** being declared before **y**"

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Evaluation examples

area 5

=> { replace the LHS area r = ... by the RHS ... = pi * r^2 }

pi * 5 ^ 2

=> { replace pi by its value }

3.141592653589793 * 5 ^ 2

=> { apply exponentiation (^) }

3.141592653589793 * 25

=> { apply multiplication (*) }

78.53981633974483

area r = pi * r^2

pi = 3.141592653589793

5^2 = 25

3.141592653589793 * 25 =

78.53981633974483

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Translation to instructions

functional programming

- making the **compiler** **translate** **functions** and **variables**
  to the step-by-step **instructions**
  that the computer can process.

replace each **function** and **variable** with its **definition**

repeatedly replace the results until a single value remains.

to apply or call a function means

to **replace** the LHS of its **definition** by its RHS.

LHS = RHS

LHS

RHS

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Scope

Scope rules define the **visibility rules** for **names** in a programming language.

What if you have references to a **variable** named **k** in different parts of the program?

Do these refer to the same variable or to different ones?

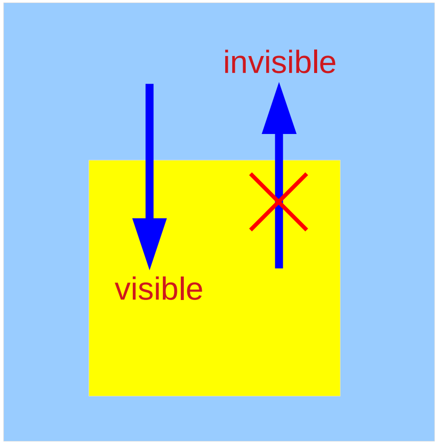
https://courses.cs.washington.edu/courses/cse341/03wi/imperative/scoping.html

Haskell Scope

Most languages, including Haskell, are **statically scoped**.

- A **block** defines a new **scope**.
- **Variables** can be declared in that scope,
  and are not visible from the outside.
- However, **variables** outside the scope (in enclosing scopes)
  are visible unless they are overridden.
- In Haskell, these scope rules also apply
  to the names of **functions**.

Static scoping is also sometimes called **lexical scoping**.



https://courses.cs.washington.edu/courses/cse341/03wi/imperative/scoping.html

Side Effects Definition

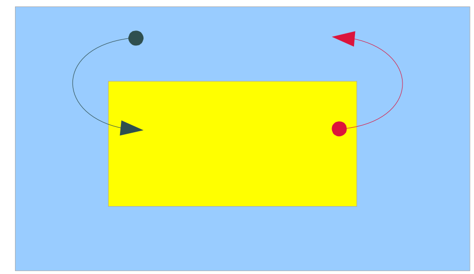
a **function** or **expression** is said to have a **side effect**

if it **modifies** some state outside its scope or

has an observable interaction

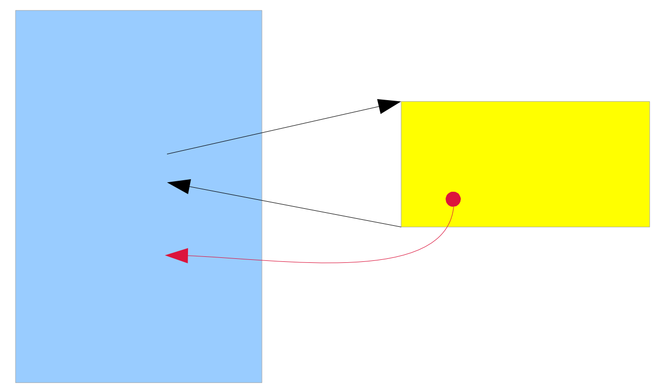
with its calling functions or the outside world

besides **returning a value**.



a particular **function** might

- modify a **global** variable or **static** variable
- modify one of its **arguments**
- raise an **exception**
- write data to a **display** or **file**
- read data from a **keyboard** or **file**
- call *other side-effecting functions*



https://en.wikipedia.org/wiki/Side_effect_(computer_science)

Some Monad types to handle side effects

State monad

manages **global variables**

Error monad

enables **exceptions**

IO monad

handles interactions with the **file system**,

and other **resources** outside the program

actions in State, Error, IO monad
have side effects

the **program** itself has no side effects

the **action** in monads does have side effects

the functional nature of the **program**

is maintained (**pure**, **no side effects**)

https://blog.osteele.com/2007/12/overloading-semicolon/

History, Order, and Context

In the presence of **side effects**,

a program's behaviour may depend on **history**;

the **order** of **evaluation** matters.

the **context** and **histories**

imperative programming : frequent utilization of **side effects**.

functional programming : **side effects** are rarely used.

The lack of side effects makes it easier

to do **formal verifications** of a program

https://en.wikipedia.org/wiki/Side_effect_(computer_science)

Side Effects Examples in C

```
int i, j;
i = j = 3;

i = (j = 3);          // j = 3 returns 3, which then gets assigned to i
```

```
// The assignment function returns 10
// which automatically casts to "true"
// so the loop conditional always evaluates to true

while (b = 10) { }
```

https://en.wikipedia.org/wiki/Side_effect_(computer_science)

Haskell Language Features (II)

Haskell Functional Programming (II)

- **Pure Function**
- **Simple IO**
- **Laziness**
- **Sequencing**

https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

Pure Languages

Haskell is a **pure** language — **no side effects**

programs are made of **functions** — **pure functions**

that cannot change

any **global state** or **variables**,

they can only

do some **computations** and **return** their **results**.

not modify **arguments** of a function

every **variable's** value does not change in time

However, some problems are inherently **stateful**

in that they rely on some state that changes over time.

immutability

st1 = 10

use a function for

stateful computations

s -> (x,s)

st1 (v,10)

a bit tedious to model

Haskell has the **state monad** features

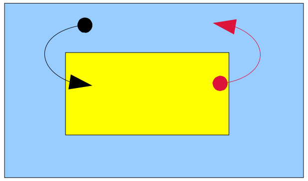
http://learnyouahaskell.com/for-a-few-monads-more

Pure Function

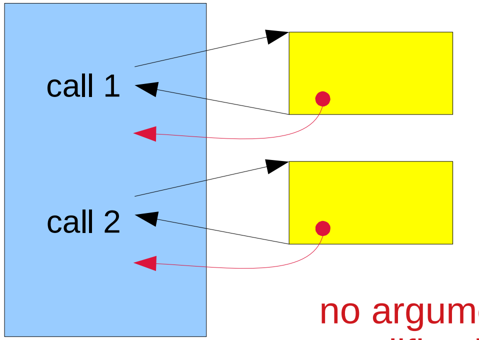
A **pure** function has no **side effects**

- **no state** nor **no** access to **external states** (global variables)
  - the function call starts from the scratch (no memory)
  - every invocation with the same set of arguments returns always the same result

- **no argument modifications**
  - calling a **pure** function is the same as
  - calling it twice and discarding the result of the first call.



no global variables



no argument modification

easily parallelizeable

no side effect means **no data races**

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Actions

Haskell **runtime**

- first evaluates **main** (an expression)
  - not to a **simple value**
  - but to an **action**. (function)
- then executes this **action**. (function)

a **function** as a **value**

IO action

- the **program** itself has no side effects
- the **action** does have side effects

stateful computation

the functional nature of the **program**
is maintained (pure, no side effects)

action

IO monad

function

evaluation - execution

main = **putStrLn** "Hello World!"

stateful computation

(side effects)

World -> ((), World)

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Simple IO

main calls functions like **putStrLn** or **print**,
which return **IO actions**.

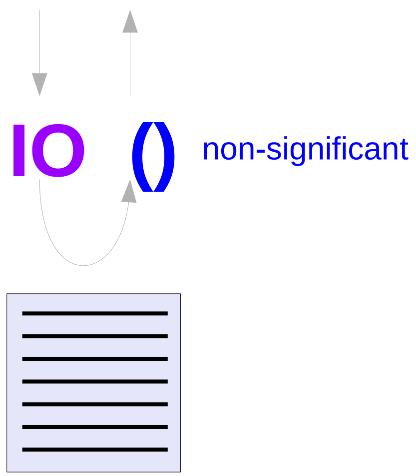
- **primitives** built into Haskell :
  the only non-trivial source of **IO actions**:

- **return** trivially converts any **value** into an **IO action**.

IO actions : **IO ()**

putStrLn :: String -> IO ()
print :: Show a => a -> IO ()

computations resulting in values



imperative code using builtin primitives

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Primitives in PutStrLn

```
...
writeCharBuffer h_ Buffer{ bufRaw=raw, bufState=WriteBuffer,
                           bufL=0, bufR=count, bufSize=sz }
...
writeCharBuffer :: Handle__ -> CharBuffer -> IO ()
writeCharBuffer h_@Handle__{..} !cbuf = do
…

-- |Write a new value into an 'IORef'
writeIORef  :: IORef a -> a -> IO ()
writeIORef (IORef var) v = stToIO (writeSTRef var v)

-- |Write a new value into an 'STRef'
writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef (STRef var#) val = ST $ \s1# ->
   case writeMutVar# var# val s1#      of { s2# ->  (# s2#, () #) }
```

s2# -> (# s2#, () #)

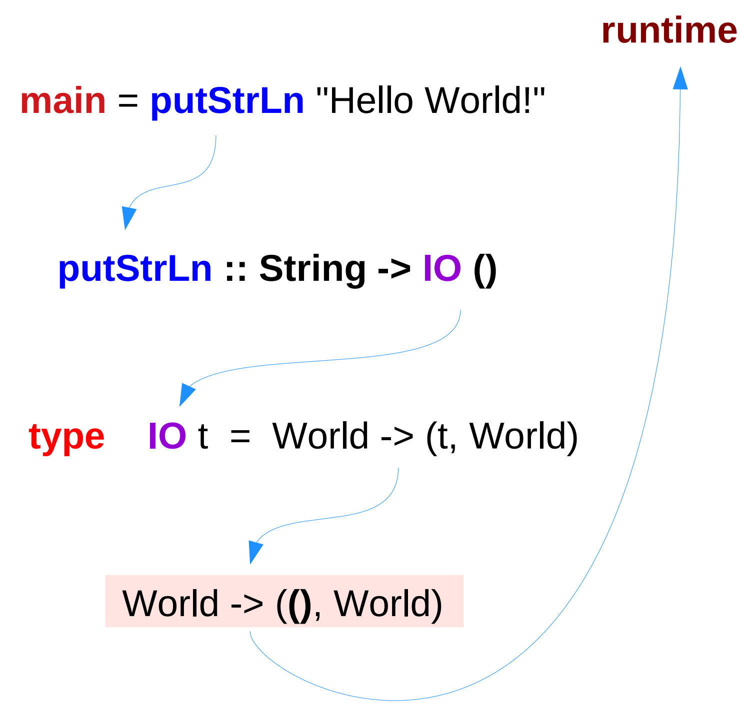
s -> (x,s)

http://hackage.haskell.org/package/base-4.11.1.0/docs/src/GHC.IO.Handle.Text.html#local-6989586621679303176

IO actions in main

IO action is <u>invoked</u>, <u>after</u> the Haskell **program** has <u>run</u>

- an **IO action** can <u>never</u> be <u>executed</u> inside the **program**
  in order to <u>execute</u> a function of the type **World -> (t, World)**
  must <u>supply</u> a **value** of the type **World**
- once <u>created</u>, an **IO action** keeps *percolating up*
  until it ends up in **main** and is <u>executed</u> by the **runtime**.
- **IO action** can be also <u>discarded</u>,
  but that means it will <u>never</u> be <u>evaluated</u>



https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness

Haskell will not **calculate** anything

unless it's strictly necessary or

is forced by the programmer

Haskell will not even **evaluate**

arguments to a function before calling it

Haskell assumes that the arguments will **not** be **used**,

so it procrastinates as long as possible.

unless proven otherwise

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness and Pure Functions

A **pure function** has no **side effects**.

Calling a **function once** is the same

as calling it **twice** and discarding the **result** of the first **call**.

not modifying its **arguments**

but modifying only the **result**

furthermore, if the **result** of any function call is not used,

Haskell will spare itself the trouble

and will never call the **function**.

exception **IO ()** **-- () non-significant result**

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness and Pure Functions

```
getChar :: RealWorld -> (Char, RealWorld)

main :: RealWorld -> ((), RealWorld)
main world0 = let (a, world1) = getChar world0

                  (b, world2) = getChar world1

              in ((), world2)
```

- not possible here to omit any call of **getChar**, just because the **result** is not used
- nor possible to reorder the **getChar**'s

world2 requires **world1**

world1 requires **world0**

the **result ()** is not used

https://wiki.haskell.org/IO_inside#Welcome_to_the_RealWorld.2C_baby

Laziness Example 1

Division by zero : **undefined -** never be evaluated.

main = print $ undefined + 1

no compile time error

but a runtime error

because of an attempt to evaluate **undefined**.

foo x = 1

main = print $ (foo undefined) + 1

Haskell calls **foo** but never evaluates its argument **undefined**

(just returns 1)

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness Example 2

this does not come from optimization:

from the definition of **foo**, the compiler

figures out that its **argument** is unnecessary.

but the result is the same

if the definition of **foo** is hidden from view in another module.

```
{-# START_FILE Foo.hs #-}
-- show
module Foo (foo) where
foo x = 1
```

```
{-# START_FILE Main.hs #-}
-- show
import Foo
main = print $ (foo undefined) + 1
```

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness with infinity

laziness allows it to deal with

- **infinity** (like an infinite list)
- the **future** that hasn't materialized yet

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness and IO action

Laziness or not, a program will be executed at some time.

why an expression should be evaluated?

among many reasons, the fundamental one is

to display its result.

without **I/O**, nothing would ever be evaluated

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation

Larger **IO actions** are composed of smaller **IO actions**.

- the **order** of **composition** matters
- **sequence IO actions**

special syntax for sequencing :

the **do** notation.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation Example

```
main = do
   putStrLn "The answer is: "
   print 43
```

sequencing two **IO actions**

- one **IO action** returned by **putStrLn**
- another **IO action** returned by **print**

inside a **do** block

proper **indentation**.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation – input action (1)

whatever you receive from the user or from a file

you assign to a variable and use it later.

```
main = do
    str <- getLine
    putStrLn str
```

when executed, creates an **action**

that will take the input from the user.

then pass this input to the rest of **actions** of the **do** block

under the **name** **str** when the rest is executed.

(not ordinary variable, but a **binding**)

immutable variable

just a binding

x <- monadic value

(only the result of the

monadic value execution)

getLine

str

binded name

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation – input action (2)

```
str <- getLine
getLine :: IO String
```

only the returned **result** is passed

- **str** is not really a variable
- **<-** is not really an assignment

- **<-** creates an **action** (execution)
- **<-** binds the name **str** to the **value** (**String**) that will be returned by executing the **action** of **getLine**.

In Haskell you never **assign** to a variable, (**immutable**) instead you **bind** a **name** to a **value**.

getLine creates an **action** that, when the **action** executed will take the **input** from the user. It will then pass that **input** to the rest of the **do block** (which is also an **action**) under the **name str** when it (the rest) is executed.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

do block operations

the **do block** is used for

sequencing a more general set of

monadic operations such as **IO actions**

IO is just one example of a **monad**

inside a **monadic do** block

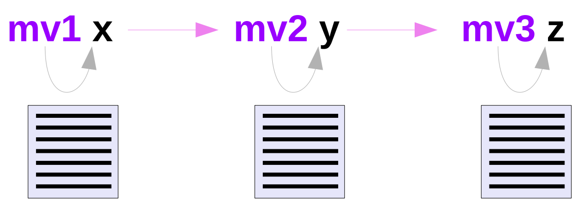
- looks like chunks of **imperative** code.
- behaves like **imperative** code

the core of **monadic operations** is built

by **imperative programming**.

```
main = do
        mv1 x
        mv2 y
        mv3 z
```

imperative code

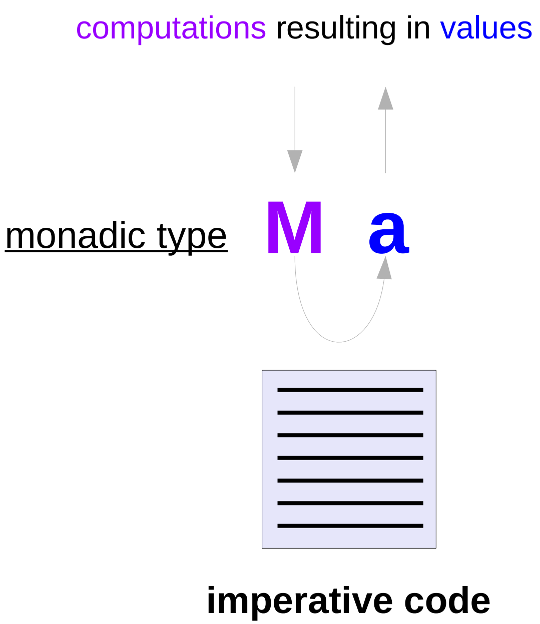


https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Monadic value

a **value** of type **M a** is interpreted — **mv** :: **M a**

as a **statement** in an imperative language **M**
that returns a **value** of type **a** as its **result**;



https://wiki.haskell.org/Functional_programming#Purity

Semicolon Overloading

The way the **actions** are glued together

is the essence of the **Monad**.

Since the glueing happens between the lines,

the **Monad** is sometimes described as

an "**overloading of the semicolon**."

Different **monads** overload it differently.

```
main = do
   putStrLn "The answer is: " ;
   print 43
```

```
main =
   putStrLn "The answer is: "  >>
   print 43
```

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Semicolon Overloading Examples

can define your own **sequencing rule**

- execute the <u>first</u> statement <u>once</u>,
  and <u>then</u> execute the <u>next</u> statement
- the <u>first</u> statement computes a <u>value</u>,
  which the <u>next</u> statement can <u>use</u>

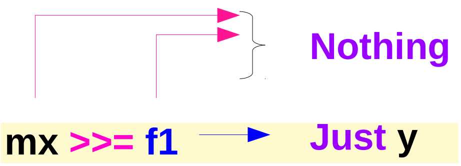
the Maybe monad

- <u>execute</u> the <u>first</u> statement, but <u>only</u> <u>execute</u>
  the <u>next</u> statement if the value so far <u>isn't</u> <u>null</u>

the List monad

- the <u>first</u> statement computes a <u>list</u> of values,
  and the <u>second</u> statement <u>runs</u> once using <u>each</u> of them

```
mx  ::     Maybe a
f1 :: a -> Maybe b
```




```
f x = [x, x+1]
g x = [x * x]
f 3 >>= g                    [9, 16]

1 : [2, 3] >>= \x -> [x *2]  [2,4,6]
```

https://blog.osteele.com/2007/12/overloading-semicolon/

Combining two statements

analogy between **statements** and **variables**

- Java and C++ have **typed variables**
- Haskell adds **typed statements**

Operators **combine values**, such as plus and times.

overload operators:

Integer+Integer, String+String, Vector+Vector

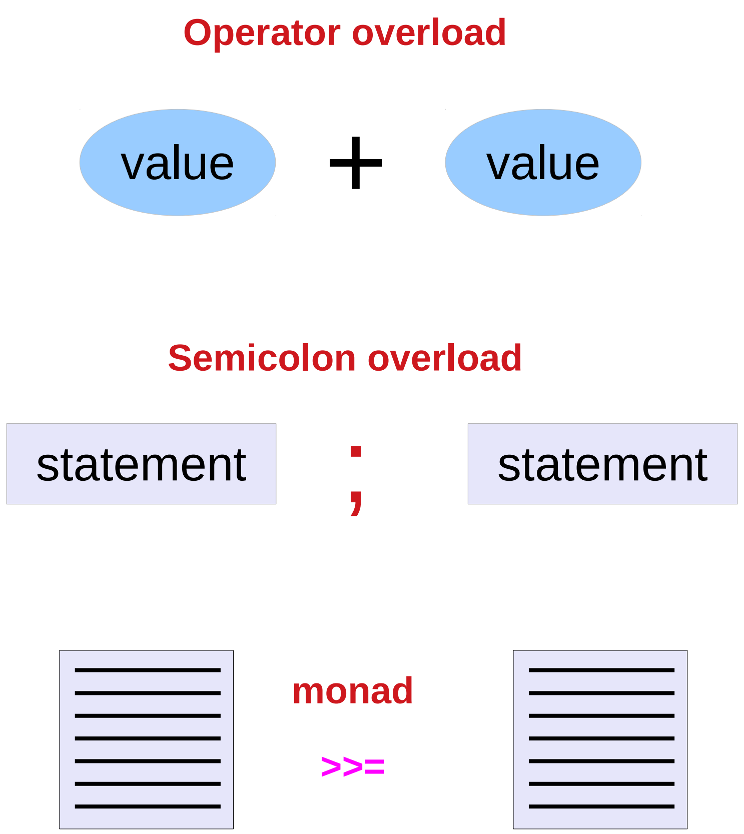
semicolon operator **combines** two **statements**.

a **monad** is a definition for the **semicolon operator**

it defines the meaning of a compound statement

composed of two simpler ones.

Haskell lets you overload semicolon.



https://blog.osteele.com/2007/12/overloading-semicolon/

Stateful Computations & IO: Side Effects in Haskell

The functional language Haskell expresses **side effects**

such as **I/O** and

other **stateful computations**

using **monadic actions**

IO monad

State monad

https://en.wikipedia.org/wiki/Side_effect_(computer_science)

Stateful Computation

a **stateful computation** is a **function** that

takes some **state** and

returns a **value** along with some **new state**.

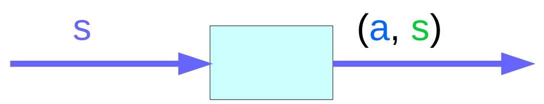
That function would have the following type:

s -> (a,s)

s is the type of the **state** and

a the **result** of the **stateful computation**.

s -> (a, s)



a **function** is an _executable_ _**data**_

when _executed_, a _**result**_ is produced

action (an executable function)

result is produced if executed

http://learnyouahaskell.com/for-a-few-monads-more

Assignment in the Haskell runtime

Assignment in an **imperative** language :

will assign the value **5** to the variable **x**

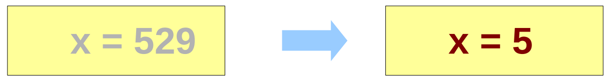
will have the value **5** *as an expression*

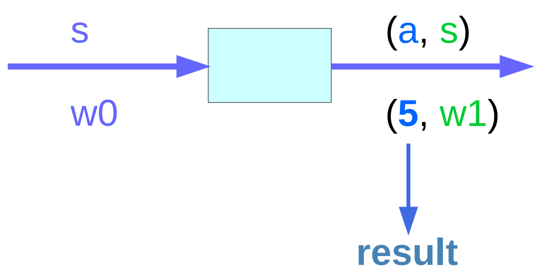
Assignment in a **functional** language

as a **function** that

takes a **state** and

returns a **result** and a **new state**





http://learnyouahaskell.com/for-a-few-monads-more

Assignment as a stateful computation

Assignment in a functional language

as a **function** that

takes a **state** and

returns a **result** and a **new state**

an input **state** :

all the variables that have been assigned previously

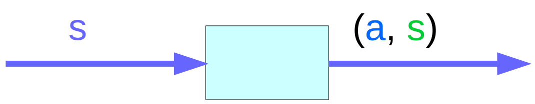
a **result** : **5**

a **new state** :

all the previous variable mappings plus

the newly assigned variable.

s -> (a, s)



all the variables that have been assigned previously

all the previous mapped variable plus the newly assigned variable

a **result** : **5**

x = 5

http://learnyouahaskell.com/for-a-few-monads-more

A value with a **context**

The **stateful computation**:

- a **function** that
  - takes a **state** and
  - returns a **result** and a **new state**
- can be considered as a **value** with a **context**

the actual **value** is the **result**

the **context** is

an **initial state** that must be provided to get the **result**

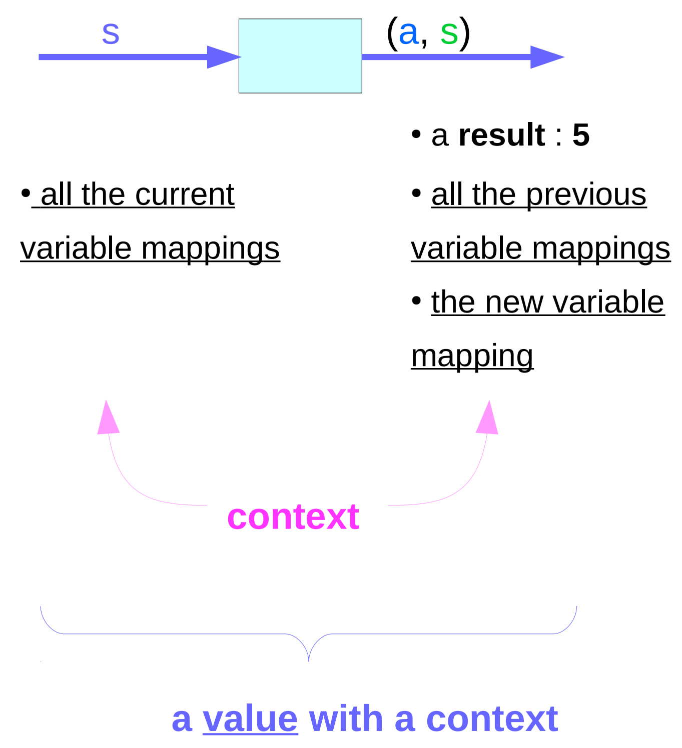
not only the **result**, but also a **new state** is obtained

through the **execution** of the function

the **result** is determined based on the **initial state**

the **result** and the **new state** depend on the **initial state**

s -> (a, s)



http://learnyouahaskell.com/for-a-few-monads-more

Stateful computations of IO Monad

Generally, a **monad** cannot perform **side effects** in Haskell.

there is a few exceptions: **IO monad, State monad**

Suppose there is a type called World,

which contains all the state of the external universe

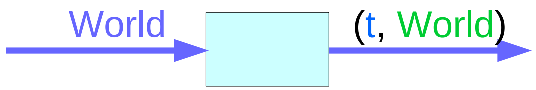
(actually a reference to such a data structure)

A way of thinking what **IO monad** does

```
type   IO t   =   World   ->   (t, World)
```

type synonym

World -> (t, World)



In Haskell, no variable changes

a state transition via a function

a collection of variables (state)

a new collection of variables (updated)

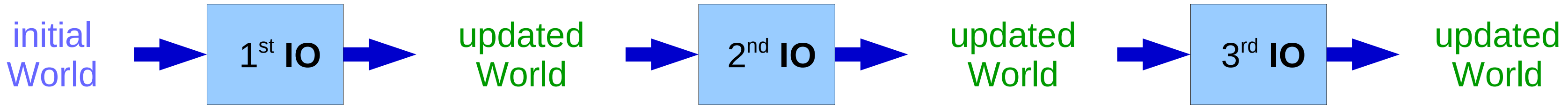
In Haskell, a function is a value

an action – an executable function

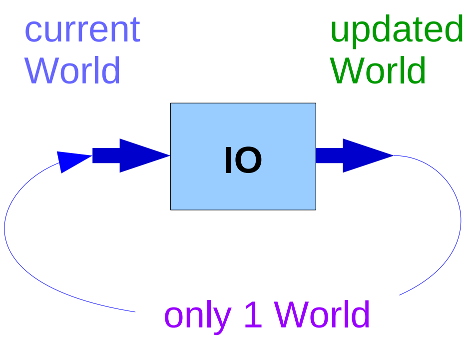
https://www.cs.hmc.edu/~adavidso/monads.pdf

Stateful computation models of IO monad

using **GHC**



using **GHCI**,



https://www.cs.hmc.edu/~adavidso/monads.pdf

Pure subset of a language

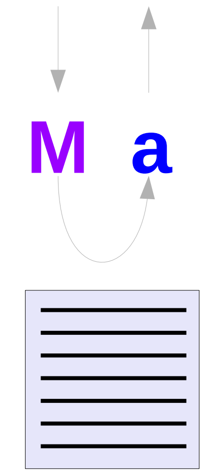
Some functional languages allow **expressions** to yield actions in addition to **return values**.

These actions are called **side effects** to emphasize that the **return value** is the most important **result** of a function

pure languages prohibit **side effects** but, **pure** subsets is still **useful**

beneficial to write a significant part of a code as **pure** and the remaining error prone **impure** part as small as possible

computations resulting in values



imperative code

actions + return values

actions may yield side effects

{ impure subset }

https://wiki.haskell.org/Functional_programming#Purity

Pure language features

Immutable Data	altered copies are used
Referential Transparency	the same result on each invocation
Lazy Evaluation	defer until needed
Purity and Effects	mutable array and IO

https://wiki.haskell.org/Functional_programming#Purity

Immutable data

Pure functional programs typically
operate on **immutable data**.

Instead of altering existing values,
altered copies are created and
the original is preserved.

Since the **unchanged parts** of the structure
cannot be modified, they can often be shared
between the old and new copies,
which saves memory.

https://wiki.haskell.org/Functional_programming#Purity

Referential Transparency

Pure computations yield the <u>same</u> <u>value</u>
each time they are <u>invoked</u>.

This property is called **referential transparency**
and makes possible to conduct
equational reasoning on the code.

no argument modification
no global variable access
: no side effects

https://wiki.haskell.org/Functional_programming#Purity

Referential Transparency Examples

y = f x

g = h y y

then we should be able

to replace the definition of **g** with

g = h (f x) (f x)

and get the same result;

only the **efficiency** might change.

https://wiki.haskell.org/Functional_programming#Purity

Lazy Evaluation

Since **pure** computations are **referentially transparent**

they can be performed at any time

and still yield the same result.

This makes it possible to defer the computation of values

until they are needed, that is, to **compute** them **lazily**.

Lazy evaluation avoids unnecessary computations

and allows **infinite data structures** to be defined and used.

https://wiki.haskell.org/Functional_programming#Purity

Purity and Effects

Even though **purely functional programming** is very beneficial,
the programmer might want to use **features**
that are not available in pure programs,
like efficient **mutable arrays** or **convenient I/O**.

There are 2 **approaches** to this problem.

1) extended impure function
2) simulating monads

https://wiki.haskell.org/Functional_programming#Purity

Using impure functions

Some functional languages **extend**

their purely functional core **with side effects**.

The programmer must be careful not to use **impure functions**

in places *where only pure functions are expected*.

https://wiki.haskell.org/Functional_programming#Purity

Using monads

Another way of **introducing side effects** to a pure language

is to **simulate** them using **monads**.

While the **language** remains **pure** and **referentially transparent**,

monads can provide **implicit state** by threading it inside them.

stateful computation

The **compiler** does not care about the **imperative features**

because the **language** itself remains **pure**,

however usually the **implementations** do care about them

due to the **efficiency reasons**,

for instance to provide **O(1) mutable arrays**.

https://wiki.haskell.org/Functional_programming#Purity

Monads enable lazy evaluation

Allowing **side effects** only through **monads**
and keeping the language **pure** makes it possible
to have **lazy evaluation** that does not conflict
with the **effects** of **impure code**.

Even though **lazy expressions** can be
evaluated **in any order**,
the **monad structure** **forces** the effects
to be executed **in the correct order**.

https://wiki.haskell.org/Functional_programming#Purity

All the effects as parameters

suppose a function **f’** has **side effects**.

if all the effects it produces are specified

as the input and output parameters (**RealWorld**),

then that function is **pure** to the outside world.

an **impure** function **f’**

f' :: Int -> Int

adding the **RealWorld** as input and output parameters

converts an **impure functon f’** into **pure function f**

f :: Int -> RealWorld -> (Int, RealWorld)

https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

Realworld parameter

f' :: Int -> Int	**impure**
f :: Int -> RealWorld -> (Int, RealWorld)	**pure**

input some states of the whole world → modify the whole world because of the a side effects → then return the new world.

https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

Use a parameterized data type IO

f :: Int -> RealWorld -> (Int, RealWorld) **pure**

define a parametrized data type

IO a = RealWorld -> (a, RealWorld)

f :: Int -> IO Int

https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

Encapsulation

```
f :: Int -> IO Int
IO a = RealWorld -> (a, RealWorld)
```

handling a **RealWorld** directly is too dangerous—
in particular, if a programmer gets their hands
on a value of type **RealWorld**,
they might try to copy it, which is basically impossible.

The definition of **IO encapsulates** the states of the whole world.

https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

Chaining

a chaining pattern of function calls

(<result-of-f>, worldY) = f worldX

(<result-of-g>, worldZ) = g <result-of-f> worldY

https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

Strictness Declaration with !

Thunk

Delayed Computation

Strictness Evaluation !

Weak Head Normal Form

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Thunk

strictness declaration

it must be evaluated to what's called

"weak normal head form"

when the data structure value is created.

data Foo = Foo Int Int !Int !(Maybe Int)

f = Foo (2+2) (3+3) (4+4) (Just (5+5))

The function **f** above, when evaluated, will return a "**thunk**":

that is, the code to execute to figure out its value.

At that point, a **Foo** doesn't even exist yet, just the **code**.

delayed computation

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Delayed Computation

```
data Foo = Foo Int Int !Int !(Maybe Int)
f = Foo (2+2) (3+3) (4+4) (Just (5+5))
```

But at some point someone may try to look inside it

```
case f of
     Foo 0 _ _ _ -> "first arg is zero"
     _           -> "first arge is something else"
```

This is going to execute enough code to do what it needs

So it will create a **Foo** with four parameters

The first parameter, we need to evaluate all the way to 4,

where we realize it doesn't match.

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Strict Evaluation !

```
data Foo = Foo Int Int !Int !(Maybe Int)
f = Foo (2+2) (3+3) (4+4) (Just (5+5))
```

The second parameter doesn't need to be evaluated,
because we're not testing it.
Thus, instead of storing the computation results 6,
store the code (3+3) that will turn into a 6
only if someone looks at it.

The third parameter, however, has a **!** in front of it,
so is ***strictly evaluated***: (4+4) is executed,
and 8 is stored in that memory location.

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Weak Normal Head Form

data Foo = Foo Int Int !Int !(Maybe Int)

f = Foo (2+2) (3+3) (4+4) (Just (5+5))

The fourth parameter is also ***strictly evaluated***.

we're evaluating not fully, but only to **weak normal head form**.

figure out whether it's **Nothing** or **Just** something,

and store that, but we go no further.

That means that we store not **Just 10** but actually **Just (5+5)**,

leaving the **thunk** inside **unevaluated**.

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

References

[1] https://en.wikibooks.org/wiki/Haskell/Variables_and_functions

[2] https://stackoverflow.com/questions/43525193/how-can-i-re-assign-a-variable-in-a-function-in-haskell

[3] https://courses.cs.washington.edu/courses/cse341/03wi/imperative/scoping.html

[4] https://en.wikipedia.org/wiki/Side_effect_(computer_science)

[5] https://blog.osteele.com/2007/12/overloading-semicolon/

[6] http://learnyouahaskell.com/for-a-few-monads-more

[7] https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

[8] http://hackage.haskell.org/package/base-4.11.1.0/docs/src/GHC.IO.Handle.Text.html#local-6989586621679303176

[9] https://wiki.haskell.org/IO_inside#Welcome_to_the_RealWorld.2C_baby

[10] https://wiki.haskell.org/Functional_programming#Purity

[11] https://www.cs.hmc.edu/~adavidso/monads.pdf

[12] https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

[13] https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration