Monad P3 : Haskel Expressions (1E)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

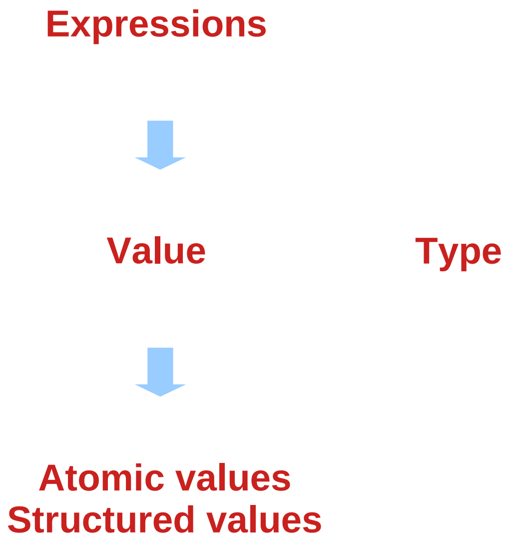
Haskell Expressions

Expressions and values

Because Haskell is a purely functional language,
all **computations** are done via the **evaluation** of
expressions (**syntactic terms**) to yield **values**

Every **value** has an associated **type**.
(Intuitively, we can think of **types** as **sets** of **values**.)

Examples of **expressions** include **atomic values**
such as the **integer 5**, the **character 'a'**,
and the **function \x -> x+1**,
as well as **structured values**
such as the **list [1,2,3]** and the **pair ('b',4)**.



Expressions

Value

Type

Atomic values
Structured values

https://www.haskell.org/tutorial/goodies.html

Type expressions and types

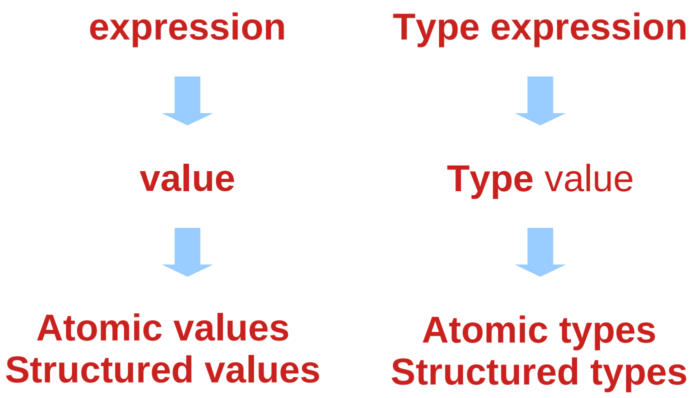
Just as **expressions** denote **values**,
type expressions are **syntactic terms**
that denote **type values** (or just **types**).

Examples of **type expressions** include the **atomic types**

- **Integer** (infinite-precision integers),
- **Char** (characters),
- **Integer->Integer** (functions mapping Integer to Integer),

as well as the **structured types**

- **[Integer]** (homogeneous lists of integers) and
- **(Char,Integer)** (character, integer pairs).



expression → **value** → **Atomic values** / **Structured values**

Type expression → **Type** value → **Atomic types** / **Structured types**

https://www.haskell.org/tutorial/goodies.html

First class values

All **Haskell values** are "**first-class**"

- they may be passed as arguments to functions,
- returned as results,
- placed in data structures, etc.

Haskell types, on the other hand, are not first-class.

https://www.haskell.org/tutorial/goodies.html

Typing

Types in a sense describe **values**, and

the **association** of a **value** with its **type** is called a **typing**.

Using the examples of values and types above,

we write **typing** as follows: (the "**::**" can be read "has type.")

```
    5  :: Integer
   'a' :: Char
   inc :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

https://www.haskell.org/tutorial/goodies.html

Function definition and declaration

Functions in Haskell are normally defined by a **series of equations**.

For example, the **function inc** can be defined by the single equation:

```
Inc  n          = n+1
```

An **equation** is an example of a **declaration**.

Another kind of **declaration** is a **type signature declaration**,

with which we can declare an **explicit typing for inc**:

```
inc           :: Integer -> Integer
```

https://www.haskell.org/tutorial/goodies.html

Expression evaluation =>

when we wish to indicate that an **expression e1 evaluates**, or "**reduces**," to *another* **expression** or **value e2**, we will write:

e1 => e2

For example, note that:

inc (inc 3) => 5

https://www.haskell.org/tutorial/goodies.html

Statements vs Expressions

Many programming languages differentiate **statements** from **expressions**.

Statement: What code does

Expression: What code is

can think the term "**statement**" very broadly to refer to anything that is not an **expression** or **type declaration**.

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Imperative vs functional languages

statements vs. **expressions** closely parallels

imperative languages vs. **functional languages**:

Imperative: A language that *emphasizes* **statements**

Functional: A language that *emphasizes* **expressions**

C lies at one end of the spectrum (imperative),

relying heavily on **statements** to accomplish everything.

Haskell lies at the exact opposite extreme (functional),

using **expressions** heavily:

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Statement examples in the imperative language C

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int elems[5] = {1, 2, 3, 4, 5};          // statement

    int total = 0;
    int i;

    for (i = 0; i < 5; i++) {                // statement
        total += elems[i];                   // statement
    }
    printf("%d\n", total);                   // statement

    return 0;
}
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Expression examples in the functional language Haskell (1)

everything in Haskell is an **expression**,

and even **statements** are **expressions**.

```
main = print (sum [1..5])            -- Expression
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Expression examples in the functional language Haskell (2)

For example, the following code might appear to be
a traditional imperative-style sequence of statements:

```
main = do
   putStrLn "Enter a number:"              -- Statement?
   str <- getLine                          -- Statement?
   putStrLn ("You entered: " ++ str)       -- Statement?
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Expression examples in the functional language Haskell (3)

but **do** notation is merely syntactic sugar

for nested applications of **(>>=),** which is itself nothing more than

an infix higher-order function:

```
main =
    putStrLn "Enter a number:"   >>= (\_   ->        -- Expression
         getLine                 >>= (\str ->        -- Sub-expression
             putStrLn ("You entered: " ++ str) ))    -- Sub-expression
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Statement-as-expression

In Haskell, "**statements**" are actually **nested expressions**,
and **sequencing statements** just builds larger and larger **expressions**.

This statement-as-expression paradigm promotes consistency
and prevents arbitrary language limitations,
such as Python's restriction of lambdas to single statements.

In Haskell, you cannot limit
the number of statements a **term** uses
any more than you can limit the number of **sub-expressions**.

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Monads

do notation works for more than just **IO**.

Any **type** that implements the **Monad class**

can be "*sequenced*" in **statement** form,

as long as it supports the following two operations:

```
class Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b

   return :: a -> m a
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Statement-like syntax using monads

This provides a uniform interface for translating imperative **statement-like** syntax into **expressions** under the hood.

For example, the **Maybe** type implements the Monad class:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    m >>= f = case m of
        Nothing -> Nothing
        Just a  -> f a
    return = Just
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

do notation using monads

This lets you assemble **Maybe-based** computations using **do** notation

```
example :: Maybe Int
example = do
    x <- Just 1
    y <- Nothing
    return (x + y)
```

```
example =
        Just 1   >>= (\x ->
        Nothing >>= (\y ->
        return (x + y) ))
```

The above code desugars to nested calls to **(>>=)**:

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Substitute **>>=** and **return**

The compiler then substitutes in our definition of **(>>=)** and **return**

```
example = case (Just 1) of
   Nothing -> Nothing
   Just x  -> case Nothing of
      Nothing -> Nothing
      Just y  -> Just (x + y)
```

```
example =
       Just 1   >>= (\x ->
       Nothing >>= (\y ->
       return (x + y) ))
```

```
instance Monad Maybe where
   m >>= f = case m of
      Nothing -> Nothing
      Just a  -> f a
   return = Just
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Evaluate the outer and inner **case** expression

We can then hand-evaluate this expression to prove that it short-circuits when it encounters Nothing:

```
-- Evaluate the outer `case`
example = case Nothing of
    Nothing -> Nothing
    Just y  -> Just (1 + y)

-- Evaluate the remaining `case`
example = Nothing
```

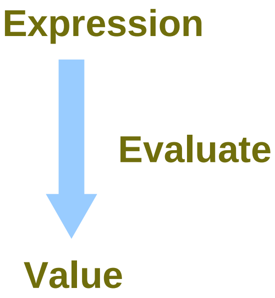
```
example = case (Just 1) of
    Nothing -> Nothing
    Just x  -> case Nothing of
        Nothing -> Nothing
        Just y  -> Just (x + y)
```

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Everything is an expression to be evaluated

Notice that we can evaluate these **Maybe** "statements"
without invoking any sort of **abstract machine**.

When everything is an **expression**,
everything is simple to **evaluate**
and does not require *understanding* or
invoking an execution model.



Expression

Evaluate

Value

FSM not needed
for sequencing

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Semantics

Semantics

In fact, the distinction between **statements** and **expressions** also closely parallels another important divide: the difference between **operational semantics** and **denotational semantics**.

Operational semantics:

Translates code to **abstract machine statements**

Denotational semantics:

Translates code to **mathematical expressions**

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Expressions and their meaning

Haskell teaches you
to think denotationally in terms of expressions and their meanings
instead of statements and an abstract machine.

This is why Haskell makes you a better programmer:
you *separate* your mental model
from the underlying execution model, … abstract machine
so you can more easily identify *common patterns*
between diverse programming languages and problem domains.

https://www.haskellforall.com/2013/07/statements-vs-expressions.html

Haskell expression

the distinction between **statements** and **expressions**

in **imperative languages**

x = 2 + 2;

the **x =** ...; part being a **statement**

the **2 + 2** part being an **expression**.

The **body** of a **Haskell function** is

always one single **expression**

although you can split that one expression apart for convenience

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Haskell expression

So if you want to "do more than one thing",
which is an **imperative** notion of a **function**
being able to change **global state**,
you solve this with **monads**, like so:

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Web service examples

Scotty is a web framework written in Haskell,

which is similar to **Ruby**'s **Sinatra**.

You can install it using the following commands:

```
$ sudo apt-get install cabal-install
$ cabal update
$ cabal install scotty
```

You can compile and start the server from the terminal

```
$ runghc hello-world.hs
Setting phasers to stun... (port 3000) (ctrl-c to quit)
```

http://shakthimaan.com/posts/2016/01/27/haskell-web-programming/news.html

hello-world.hs

```
$ runghc hello-world.hs
```

The service will run on port 3000, and
you can open localhost:3000 in a browser
to see the `Hello, World!' text.

You can also use **Curl** to make a query to the server.

```
$ sudo apt-get install curl
```

```
$ curl localhost:3000
Hello, World!
```

```haskell
-- hello-world.hs
{-# LANGUAGE OverloadedStrings #-}

import Web.Scotty

main :: IO ()
main = scotty 3000 $ do
  get "/" $ do
    html "Hello, World!"
```

http://shakthimaan.com/posts/2016/01/27/haskell-web-programming/news.html

Web service requests and responses

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import Network.HTTP.Types

main = scotty 3000 $ do
  get "/" $ do                                 -- handle GET request on "/" URL
    text "This was a GET request!"             --      send 'text/plain' response
  delete "/" $ do                              -- handle DELETE request on "/" URL
    html "This was a DELETE request!"          –       send 'text/html' response
  post "/" $ do                                -- handle POST request on "/" URL
    text "This was a POST request!"            --      send 'text/plain' response
  put "/" $ do                                 -- handle PUT request on "/" URL
    text "This was a PUT request!"             --      send 'text/plain' response
```

https://dev.to/parambirs/how-to-write-a-haskell-web-servicefrom-scratch---part-3-5en6

Overloaded Strings

{-# LANGUAGE OverloadedStrings #-}
is called a **language pragma** and
extends the languauge with nice features.

In this case, **OverloadedStrings** allows us to write a string and
it gets automatically converted to the **string type** we need
(**String**, **ByteString**, or **Text**).

{-# LANGUAGE OverloadedStrings #-}

https://www.stackbuilders.com/blog/getting-started-with-haskell-projects-using-scotty/

Entry function **scotty**

scotty is the entry function
that **Scotty** defines for running an application.

The first **parameter** is the **port** that we want it to run in, and
the rest is the **application**,
which looks like a **list** of **routes** and **handlers**.

For now, we only have one **route** (the root) and a **handler**,
which is a **GET** and returns an **HTML string** with a **title**.

```
scotty 3000 $
  get "/" $
    html "<h1>Shortener</h1>"
```

https://www.stackbuilders.com/blog/getting-started-with-haskell-projects-using-scotty/

Named and unnambed parameters

```
-- named parameters:
get "/askfor/:word" $ do
   w <- param "word"
   html $ mconcat ["<h1>You asked for ", w, ", you got it!</h1>" ]



-- unnamed parameters from a query string or a form:
post "/submit" $ do          -- e.g. http://server.com/submit?name=somename
  name <- param "name"
  text name
```

https://dev.to/parambirs/how-to-write-a-haskell-web-servicefrom-scratch---part-3-5en6

Haskell expression in scotty examples (1)

```
{-# LANGUAGE OverloadedStrings #-}
module Main (main) where
import Web.Scotty

main :: IO ()
main = scotty 3000 $
  get "/:who" $ do
    who <- param "who"
    text ("Beam " <> who <> " up, Scotty!")
```

```
Ghci> [1,2,3] <> [4,5,6]                -- concatenation
[1,2,3,4,5,6]
```

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Haskell expression in scotty examples (2)

Here, **main**'s **body** (a **monadic action**, not a function) is a single **expression**, **scotty 3000 (...)**.

While the linebreak1 after **scotty 3000 $** doesn't carry meaning and only makes the code look nicer,

the linebreak2 in the **do** block actually reduces multiple actions into one expression via **syntactic sugar**.

```
main :: IO ()
main = scotty 3000 $          -- linebreak1
  get "/:who" $ do            -- linebreak2
    who <- param "who"
    Text ("..." <> who <> " ...")
```

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Haskell expression in scotty examples (3)

So while it may seem that this **event handler**

does two things things:

- (1) **param "who"**
- (2) **text (...)**

it is still one expression equivalent to this:

```
{-# LANGUAGE OverloadedStrings #-}
module Main (main) where
import Web.Scotty

main :: IO ()
main = scotty 3000 $
  get "/:who" $ do
    who <- param "who"
    text ("Beam " <> who <> " up, Scotty!")
```

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Haskell expression in scotty examples (4)

```
main =
  scotty 3000
       (get "/:who"
            (param "who" >>=
                 (\who -> text ("Beam " <> who <> " up, Scotty!"))))
```

with **>>=** being the invisible operator between the do-block lines.

When expressions begin to grow, this becomes very inconvenient,
so you split parts of them into sub-expressions
and give those names, e.g. like:

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Haskell expression in scotty examples (5)

```
main = scotty 3000 handler
  where
    handler = do
      get "/:who" getWho
      post "/" postWho

    getWho = do
      ...
    postWho = do
      ...
```

But it is essentially equivalent to one big expression.

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Haskell expression in scotty examples (6)

There are many things in the language beyond function bodies that are not expressions; in the example above, the following are <u>not</u> **expressions**:

{-# LANGUAGE OverloadedStrings #-}	(a language pragma)
module Main (main) where	(a module, export list)
import Web.Scotty	(an import declaration)
main :: IO ()	(a type signature)
main =	(a top declaration, or a value binding)

https://stackoverflow.com/questions/63144227/what-is-an-expression-in-haskell

Haskell expression in scotty examples (7)

import Web.Scotty could be called a kind of **statement**,
since *grammatically* it's in **imperative form**,
but if we're going to be imprecise,
It would be ok to call them all **declarations**.

More interestingly, in Haskell you have
both an **expression language**
at the **value level** and one at the **type level**.

So **IO ()** isn't a **value expression**, but it's a **type expression**.
If you had the ability to mix those two **expression languages** up,
you'd have **dependent types**.

- **{-# LANGUAGE OverloadedStrings #-}**
  (a language pragma)
- **module Main (main) where**
  (a module, export list)
- **import Web.Scotty**
  (an import declaration)
- **main :: IO ()**
  (a type signature)
- **main =**
  (a top declaration, or a value binding)

https://www.haskell.org/tutorial/goodies.html

Lazy evaluation

Operational semantics

Operational semantics (1)

It is one of the key properties of **purely functional languages** like Haskell that a direct mathematical interpretation like "1+9 denotes 10" carries over to functions, too:

in essence, the denotation of a program of type **Integer -> Integer** is a mathematical function **Z** → **Z** between integers.

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

Operational semantics (2)

While we will see that this expression needs refinement generally,
to include non-termination,

the situation for **imperative languages** is clearly worse:
a **procedure** with that type denotes something
that changes the state of a machine in possibly unintended ways.

Imperative languages are tightly tied to operational semantics
which describes their way of execution on a machine.

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

Operational semantics (3)

It is possible to define a denotational semantics
for **imperative programs** and to use it
to reason about such programs,
but the semantics often has operational nature
and sometimes must be extended
in comparison to the denotational semantics
for **functional languages**.[

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

Operational semantics (4)

In contrast, the meaning of **purely functional languages** is by default completely independent from their way of execution.

The Haskell98 standard even goes as far as to specify only Haskell's non-strict denotational semantics, leaving open how to implement them.

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

Operational semantics (5)

The real quantity we're interested in formally describing is **expressions** in programming languages.

A programming language semantics is described by the operational semantics of the language.

The operational semantics can be thought of as
a description of an abstract machine
which operates over the abstract terms
of the programming language in the same way
that a virtual machine might operate over instructions.

http://dev.stephendiehl.com/fun/004_type_systems.html

Operational semantics (6)

Denotational semantics for a language provides a **function** that translates from **program syntax** into **mathematical objects** like sets, functions, lists or even some other programming language

– a denotational semantics acts like a **compiler**

Operational semantics works by rewriting or executing programs **step-by-step**

– it uses only one program syntax to explain how a program runs

https://www.cs.princeton.edu/~dpw/cos441-11/notes/slides13-lambda-calc.pdf

Operational semantics (7)

As languages become more complicated, it is often easier to define **operational semantics** than **denotational semantics**

– it requires less math to do so

– but you might not be able to prove particularly strong theorems using the semantics

https://www.cs.princeton.edu/~dpw/cos441-11/notes/slides13-lambda-calc.pdf

Operational semantics (8)

The **operational library** makes it easy to implement **monads** with tricky **control flow**.

This is very useful for:

writing web applications in a sequential style,
programming games with a uniform interface
for human and AI players and easy replay,
implementing fast parser monads,
designing monadic DSLs, etc.

Embedded Domain Specific Language means that you embed a Domain specific language in a language like Haskell.

https://apfelmus.nfshost.com/articles/operational-monad.html

Operational semantics (9)

For instance, to write a web application
where the user is guided through a sequence of tasks ("wizard").
To structure your application, you can use a custom monad
that supports an instruction **askUserInput :: CustomMonad UserInput**.

This command sends a web form to the user
and returns a result when he submits the form.
However, you don't want your server to block
while waiting for the user, so you have to suspend the computation
and resume it at some later point.
tricky to implement
This library makes it easy.

https://apfelmus.nfshost.com/articles/operational-monad.html

Operational semantics (10)

The idea is to identify a set of primitive instructions
and to specify their **operational semantics**.
Then, the library makes sure that the monad laws hold automatically.
In the web application example,
the primitive instruction would be **AskUserInput**.

Any monad can be implemented in this way.
Ditto for monad transformers.

https://apfelmus.nfshost.com/articles/operational-monad.html

Sharing (1)

Sharing means that **temporary data** is physically stored,
if it is used multiple times.

```
let x = sin 2
in  x*x
```

x is used twice as factor in the product **x*x**.

Due to **referential transparency**, it does not play a role,

whether **sin 2** is computed twice or

whether it is computed once and the result is stored and reused.

https://wiki.haskell.org/Lazy_evaluation

Sharing (2)

However, when you write **let** expression,

the **Haskell compiler** will certainly decide to store the result.

This can be the wrong way,

if a computation is cheap but its result is huge.

[0..1000000] ++ [0..1000000]

where it is much cheaper to compute the list of numbers

than to store it with full length.

https://wiki.haskell.org/Lazy_evaluation

Sharing (3)

Because the **sharing** property cannot be observed in Haskell,
it is hard to transfer the sharing property to foreign programs
when you use Haskell as an Embedded domain specific language.

You must design a **monad** or
use **unsafePerformIO** hacks, which should be avoided.

https://wiki.haskell.org/Lazy_evaluation

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf