Monad P3 : Non-terminating 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)**.

$$\begin{tikzcd}
\textbf{Expressions} \arrow[d] & \\
\textbf{Value} \arrow[d] & \textbf{Type} \\
\textbf{Atomic values} \\ \textbf{Structured values} &
\end{tikzcd}$$

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

Type expressions and types

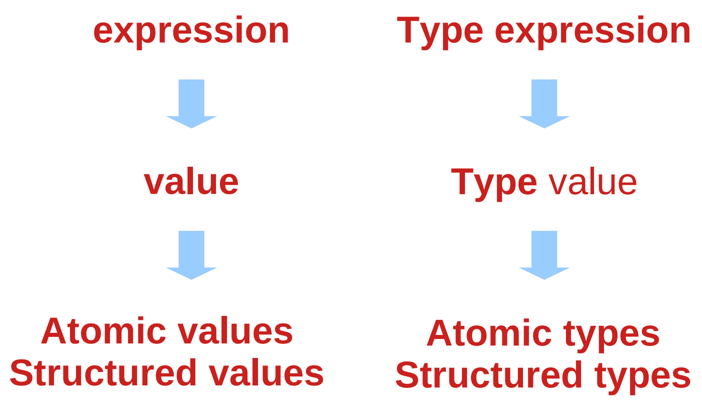
Just as **expressions** denote **values**,
type expressions are **syntactic terms**
that denote **type values** (or just **types**).

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).



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.

$$\begin{tikzcd}
\textbf{Expression} \arrow[d, "\textbf{Evaluate}"] \\
\textbf{Value}
\end{tikzcd}$$

FSM not needed
for sequencing

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

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)

```haskell
{-# 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 not **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

Non-terminating Expressions

Denotational semantics

Semantics is about defining the "**meaning**" of a **program**.

denotational semantics In Haskell

– the **value** is a mathematical object of some sort

the **expression 10** (but also the **expression 9 + 1**)

have **denotations** of the **number 10**

(rather than the Haskell **value 10**).

We usually write that **⟦9 + 1⟧ = 10** meaning that

the **denotation** of the Haskell **expression 9 + 1**

is the **number 10**.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Semantic map and Strachey brackets

Haskell **expressions** denote **mathematical values**.

Strachey brackets ⟦·⟧

to denote the "**semantic mapping**"

from **Haskell** to **Math**.

we want our **semantic brackets** to be compatible

with **semantic operations**.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Semantic map example

⟦x + y⟧ = ⟦x⟧ + ⟦y⟧

on the left side + is the Haskell function

(+) :: Num a => a -> a -> a

and on the right side it's the binary operation

in a **commutative group**.

we can use the properties from the **semantic map**

to know how our Haskell functions should work.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Commutative property example

the commutative property "**in Math**"

⟦x⟧ + ⟦y⟧ == ⟦y⟧ + ⟦x⟧
= ⟦x + y⟧ == ⟦y + x⟧
= ⟦x + y == y + x⟧

where the third step also indicates that the Haskell
(==) :: Eq a => a -> a -> a
ought to have the properties of a
mathematical equivalence relationship.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Irrecoverable / recoverable errors

expressions that result in some kind of a **run-time error**,
such as dividing by zero, have the **value _|_** (read "**bottom**").

Such an **error** is not recoverable: *irrecoverable errors*
programs will not continue past these errors.

errors encountered by the **I/O system**, *recoverable errors*
such as an **end-of-file error**, are recoverable
and are handled in a different manner.

Such an **I/O error** is really not an **error** at all
but rather an **exception**.

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

Value in the semantic sense

The **value** is ⊥, usually pronounced "**bottom**".

It is a **value** in the *semantic sense*
-- it is not a normal Haskell value per se.

It represents **computations**
that do not produce a normal Haskell **value**:

exceptions and infinite loops, for example.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Denotational semantics and ⊥

denotational semantics, where ⊥ lives, is

a mapping **Haskell values**

to some **other space of values**.

in order to give meaning to programs

in a more formal manner

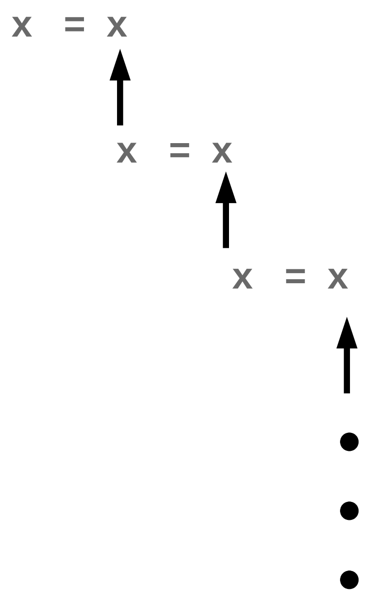
than just talking about what programs should do

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

let x = x in x

Consider an expression like **let x = x in x**

- there is no Haskell **value** for this expression.
- If you tried to evaluate it, it would simply never finish.
- not obvious what **mathematical object** this corresponds to.



https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

⊥ for computations that does not return

in order to reason about programs

that have the following characteristics,

we need to give some **denotation** for it.

- with no Haskell **value**
- never finishing upon evaluation
- not obvious **mathematical object**

So, essentially, we just *make up a value* ⊥ (**bottom**)

for all these computations

So ⊥ is just a way to define

what a computation that doesn't return "means".

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

⊥ for throwing exceptions

We also define other computations like

undefined and error "some message" as ⊥

because they also do not have obvious normal **values**.

So throwing an exception corresponds to ⊥.

This is exactly what happens with a failed pattern match.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Lifted type

every Haskell **type** is "lifted" -- it *contains* ⊥.

That is, **Bool** corresponds to {⊥, **True**, **False**}
rather than just {**True**, **False**}.

This represents the fact that Haskell programs are
not guaranteed to **terminate** and can have **exceptions**.

This is also true when you define *your own type*
-- the type contains every value you defined for it as well as ⊥.

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Bottom value in normal code

interestingly, since Haskell is non-strict,

⊥ can exist in normal code.

So you could have a value like Just ⊥ ,

and everything will work fine, unless you **evaluate** it,

A good example of this is const:

```
        const 1 ⊥                                        -- 1
```

this works for failed pattern matches as well:

```
        const 1 (let Just x = Nothing in x)              -- 1
```

constant function

const :: a -> b -> a

Input: **const 12 3**

Output: **12**

Input: **const 12 (3/0)**

Output: **12**

```
aaa x y = let    r = 3 *x
                 s = 6 *y
                 in  r + s
```

Input: **aaa 2 4**

Output: **30**

https://stackoverflow.com/questions/14698414/haskell-pattern-match-diverge-and-%e2%8a%a5/14698510#14698510

Pattern match in **let** expression (1)

```
let
  Just x = (binom (n-1) (k-1))
  Just y = (binom (n-1) k)
in
  Just (x + y)
```

It is fine from the type-checking point of view

extracting the underlying values from the **Just wrapper**
(these are **x** and **y**), adding them up and rewrapping them.

https://stackoverflow.com/questions/68240639/why-cant-you-use-just-syntax-without-let-in-block-in-haskell

Pattern match in **let** expression (2)

pattern matches in the **let**... **in** expression

assume that the results of **binom (n-1) (k-1)**

the results of the form **Just x**

but they could also be **Nothing** -

in which case your program will crash at runtime!

The "assignment" **Just x = ...**

matches ... against **Just x**,

binding **x** to the wrapped value *if the match succeeds*.

It doesn't apply **Just** to anything.

```
let
  Just x = (binom (n-1) (k-1))
  Just y = (binom (n-1) k)
in
  Just (x + y)
```

https://stackoverflow.com/questions/68240639/why-cant-you-use-just-syntax-without-let-in-block-in-haskell

Non-strict semantics (1)

An **expression language** is said to have **non-strict semantics**

if **expressions** can have a **value**

even if some of their **subexpressions** do not

Haskell is one of the few modern languages

to have **non-strict semantics** by default:

nearly every other language has **strict semantics**,

if any **subexpression** fails to have a **value**,

the whole expression fails with it.

https://wiki.haskell.org/Non-strict_semantics

Non-strict semantics (2)

non-strict semantics is one of the most important features in Haskell:

it is what allows programs

to work with conceptually **infinite** data structures,

and it is why people say that

Haskell lets you write *your own* **control** structures.

It's also one of the motivations

behind Haskell being a **pure language**

(though there are several other good ones).

https://wiki.haskell.org/Non-strict_semantics

Pure functions (1)

A **function** is called **pure**

if it corresponds to a function in the mathematical sense:

it associates each possible **input** value with an **output** value,

and does nothing else. In particular, it has no **side effects**

that is to say, invoking it produces no observable effect

other than the result it returns;

it cannot also e.g. write to disk, or print to a screen.

https://wiki.haskell.org/Pure

Pure functions (2)

A pure function is trivially referentially transparent

it does not depend on anything other than its parameters,
so when invoked
in a different **context** or
at a different **time**
with the same **arguments**,
it will produce the same **result**.

A programming language may be called purely functional
if evaluation of expressions is pure.

https://wiki.haskell.org/Pure

Non-strict vs. strict evaluation (1)

Non-strictness means that

reduction (the mathematical term for **evaluation**)

proceeds from the outside in,

(a+(b*c)) : first +, then (b*c)

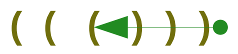
Strict languages work the other way around,

from the inside out

(a+(b*c)) : first (b*c), then +

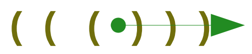
Non-strictness

from the outside in,



Strict

from the inside out



https://wiki.haskell.org/Lazy_vs._non-strict

Non-strict vs. strict evaluation (2)

With **non-strictness**

the outer reduction may eliminate some of the sub-expressions

and does not evaluate them

so "bottom" can be eliminated and don’t get be evaluated

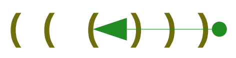
With **strictness**

if any sub-expression evaluates to bottom

then the bottom will *propagate outwards*.

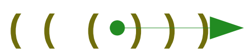
Non-strictness

from the outside in,



Strict

from the inside out



https://wiki.haskell.org/Lazy_vs._non-strict

Lazy evaluation (1)

Technically, **lazy evaluation** means **call-by-name** plus **Sharing**.
A kind of opposite is **eager evaluation**.

Lazy evaluation is part of **operational semantics**, i.e.
how a Haskell program is evaluated.

The counterpart in **denotational semantics**, i.e.
what a Haskell program computes, is called **Non-strict semantics**.
This semantics allows one to bypass undefined values
(e.g. results of infinite loops) and in this way it also allows
one to process formally infinite data.

https://wiki.haskell.org/Lazy_evaluation

Lazy evaluation (2)

Lazy evaluation is a method to evaluate a Haskell program.

It means that expressions are not evaluated
when they are bound to variables,
but their evaluation is deferred
until their results are needed by other computations.

In consequence, arguments are not evaluated
before they are passed to a function,
but only when their values are actually used.

https://wiki.haskell.org/Lazy_evaluation

Lazy evaluation (3)

While lazy evaluation has many advantages,
its main drawback is that memory usage
becomes hard to predict.

The thing is that while two expressions, like **2+2 :: Int** and **4 :: Int**,
may denote the same value 4,
they may have very different sizes and
hence use different amounts of memory.

https://wiki.haskell.org/Lazy_evaluation

Lazy evaluation (4)

An extreme example would be the infinite list 1 : 1 : 1 …
and the expression let x = 1:x in x.

The latter is represented as a cyclic graph,
and takes only finite memory, but its denotation is the former infinite list.

https://wiki.haskell.org/Lazy_evaluation

Evaluation models of a function

Call-by-value:

arguments are evaluated before a function is entered

Call-by-name:

arguments are passed unevaluated

Call-by-need:

arguments are passed unevaluated

but an expression is only evaluated once

and shared upon subsequent references

http://dev.stephendiehl.com/fun/005_evaluation.html

Reductions in the expression **f x**

Given an **expression f x**

Call-by-value:	Evaluate **x** to **v**
	Evaluate **f** to **λy.e**
	Evaluate **[y/v]e**
Call-by-name:	Evaluate **f** to **λy.e**
	Evaluate **[y/x]e**
Call-by-need:	Allocate a thunk v for **x**
	Evaluate **f** to **λy.e**
	Evaluate **[y/v]e**

http://dev.stephendiehl.com/fun/005_evaluation.html

Lambda calculus (1)

The central concept in the **lambda calculus** is
an **expression** which we can think of as a program
that when evaluated returns a result
consisting of *another* **lambda calculus expression**.

Here is the grammar for lambda expressions:

expr → λ variable . expr | expr expr | variable | (expr) | constant

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (2)

Here is the grammar for lambda expressions:

expr → λ variable . expr | expr expr | variable | (expr) | constant

A **variable** is an identifier.

A **constant** is a built-in function such as addition or multiplication,
or a constant such as an integer or boolean.

all programming language constructs can be represented
as **functions** with the pure **lambda calculus**
so these **constants** are unnecessary.
However, we will use some constants for notational simplicity.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (3) – function abstraction

A **function abstraction**, often called a **lambda abstraction**,
is a **lambda expression** that defines a **function**.

A **function abstraction** consists of *four parts*:
a **lambda** followed by a **variable**, a **period**,
and then an **expression** as in **λx.expr**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (4) – function abstraction

For example, the function abstraction **λx. + x 1**

defines a **function of x** that *adds* **x** to **1**.

Parentheses can be added to lambda expressions for clarity.

Thus, we could have written this function abstraction

as **λx.(+ x 1)** or even as **(λx. (+ x 1))**.

In C this function definition might be written as

```
    int addOne (int x) {
      return (x + 1);     }
```

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (4) – function abstraction

the **function abstraction λx. + x 1**

C function definition

```
        int addOne (int x) {
          return (x + 1);     }
```

Note that unlike C the **lambda abstraction**

does not give a **name** to the function.

The **lambda expression** itself is the **function**.

We say that **λx.expr** binds the **variable x** in **expr**

and that **expr** is the **scope** of the **variable**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (5) – beta reduction

A **function application λx.e f** is evaluated

by substituting the argument **f**

for all free occurrences of the formal parameter **x**

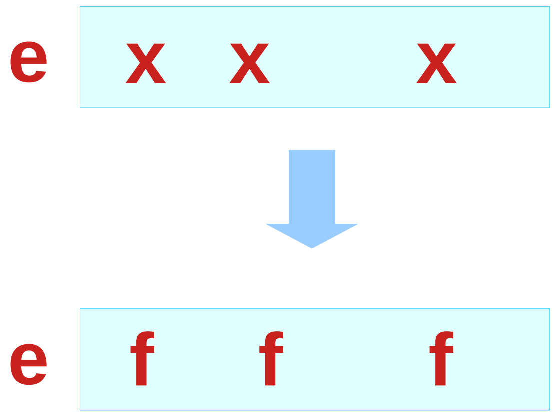
in the body **e** of the **function definition**.

We will use the notation **[f/x]e** to indicate

that **f** is to be substituted for all free occurrences of **x**

in the expression **e**.

[f/x]e



http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (5) – beta reduction

Examples:

(λx.x)y → [y/x]x = y.

(λx.xzx)y → [y/x]xzx = yzy.

(λx.z)y → [y/x]z = z

since the formal parameter x does not appear in the body z.

This substitution in a function application is called
a beta reduction and we use a right arrow to indicate it.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (5) – beta reduction

If **expr1** → **expr2**, we say **expr1** reduces to **expr2** in one step.

In general, **(λx.e)f** → **[f/x]e** means that
applying the function **(λx.e)** to the argument expression **f**
reduces to the expression [f/x]e where the argument expression **f**
is substituted for the function's formal parameter **x** in the function body **e**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Lambda calculus (5) – beta reduction

A lambda calculus expression (aka a "program") is "run" by computing a final result by repeatly applying beta reductions. We use $\rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$; that is, zero or more applications of beta reductions.

Examples:

(λx.x)y $\rightarrow$ y (illustrating that λx.x is the identity function).

(λx.xx)(λy.y) $\rightarrow$ (λy.y)(λy.y) $\rightarrow$ (λy.y); thus, we can write (λx.xx)(λy.y) $\rightarrow^*$ (λy.y). Note that here we have applied a function to a function as an argument and the result is a function.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Call by value (2)

Call by value is an extremely common evaluation model. Many programming languages both imperative and functional use this evaluation strategy. The essence of call-by-value is that there are two categories of expressions: terms and values. Values are lambda expressions and other terms which are in normal form and cannot be reduced further. All arguments to a function will be reduced to normal form before they are bound inside the lambda and reduction only proceeds once the arguments are reduced.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by value (2)

For a simple arithmetic expression, the reduction proceeds as follows. Notice how the subexpression (2 + 2) is evaluated to normal form before being bound.

```
(\x. \y. y x) (2 + 2) (\x. x + 1)
=> (\x. \y. y x) 4 (\x. x + 1)
=> (\y. y 4) (\x. x + 1)
=> (\x. x + 1) 4
=> 4 + 1
=> 5
```

http://dev.stephendiehl.com/fun/005_evaluation.html

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

Lazy vs. non-strict (1)

only evaluating an expression *when its results are needed*
(note the shift from "reduction" to "evaluation").

when the evaluation engine sees an expression
it builds a **thunk** data structure containing
whatever **values** are needed to evaluate the expression,
plus a **pointer** to the expression itself.

when the result is actually needed
the evaluation engine calls the **expression** and
then replaces the **thunk** with the result for future reference.

https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict (2)

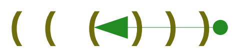
Obviously there is a strong correspondence
between a **thunk** and a partly-evaluated expression.

in most cases the terms "**lazy**" and "**non-strict**"
seem to be synonyms.

but not quite, for instance
imagine an evaluation engine
on highly parallel hardware
that fires off sub-expression evaluation *eagerly*,
but then *throws away* results that are not needed.

With **non-strictness**
if you start from the outside and
work in, then some of the
sub-expressions are *eliminated*
by the outer reductions,
so they *don't get evaluated*
and you *don't get* "bottom".

Non-strictness
from the outside in,



https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict (3)

In practice Haskell is not a purely **lazy** language:

for instance **pattern matching** is *usually* **strict**

So trying a **pattern match** forces **evaluation** to happen

at least far enough to accept or reject *the match*.

You can prepend a **~** in order

to make **pattern matches lazy**

https://wiki.haskell.org/Lazy_vs._non-strict

Lazy vs. non-strict (4)

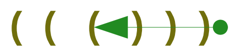
The **strictness analyzer** also looks for cases where **sub-expressions** are always required by the **outer expression**, and converts those into **eager evaluation**.

It can do this because the semantics (in terms of "bottom") don't change.

Programmers can also use the **seq** primitive to force an **expression** to evaluate regardless of whether the result will ever be used. **$!** is defined in terms of **seq**.

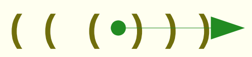
Non-strictness

from the outside in,



Strict

from the inside out



With **non-strictness** reduction from the outside in then some sub-expressions are *eliminated* by the outer reductions, so they *don't get evaluated* and you *don't get* "bottom".

https://wiki.haskell.org/Lazy_vs._non-strict

Terminating expression

Intuitively,

a specific **function evaluation** is **terminating**,

where the **value** of every **argument** is supplied

if the Haskell **evaluation strategy** needs

finite number of steps to compute the result completely.

http://termination-portal.org/wiki/Functional_Programming

Non-terminating expression

the **function zeros** is considered **non-terminating**.

RHS is to be evaluated recursively, infinitely

```
zeros :: [Integer]
zeros = 0:  zeros
```

the **evaluation** does not stop

when reaching a **term** headed by a **constructor**:

it will continue evaluating the **arguments** of this **constructor**.

```
zeros = 0:  zeros
            0:  zeros
                  0:  zeros
                        0:  zeros
```

http://termination-portal.org/wiki/Functional_Programming

repeat

```
repeat :: a -> [a]
```

it creates an *infinite* list where all items are the first argument

```
take 4 (repeat 3)
[3,3,3,3]

take 6 (repeat 'A')
"AAAAAA"

take 6 (repeat "A")
["A","A","A","A","A","A"]
```

http://zvon.org/other/haskell/Outputprelude/repeat_f.html

foldr (1)

foldr will execute the callback **function** once
for each element in the structure.

The result will be passed
to the next invocation of the callback.

For the initial call to callback,
previousValue will be initialValue,
currentValue will be the last element of the structure.

https://wiki.haskell.org/Data.Foldable.foldr

foldr (2)

foldr (+) 4 [0, 1, 2, 3]

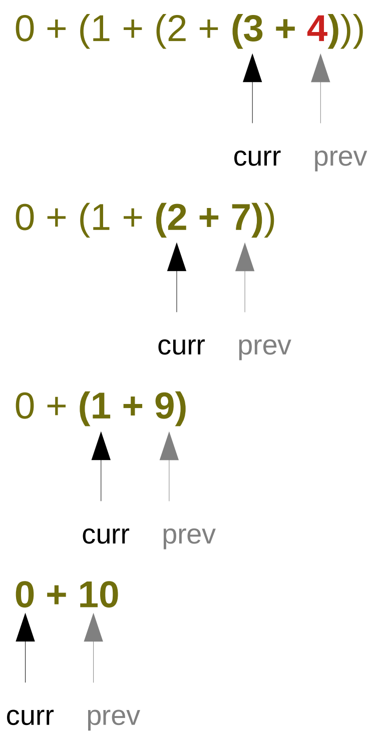
-- alternatively written without syntactic sugar for lists:

foldr (+) 4 (0 : (1 : (2 : (3 : []))))

would be equivalent to:

0 + (1 + (2 + **(3 + 4**)))

PreviousValue	= initValue	= 4
CurrentValue	= last value	= 3



https://wiki.haskell.org/Data.Foldable.foldr

foldr (3)

foldr :: (a -> b -> b) -> b -> [a] -> b

it takes the second argument	**b**
and the last item of the list	**a** in **[a]**
and applies the function,	**(a -> b -> b)**

then it takes the penultimate item from the end
and the result, and so on.

last but one in a series of things; second last.

http://zvon.org/other/haskell/Outputprelude/foldr_f.html

foldr (4)

```
foldr :: (a -> b -> b) -> b -> [a] -> b

Input: foldr (+) 5 [1,2,3,4]          1 + (2 + (3 + (4 + 5)))
Output: 15

Input: foldr (/) 2 [8,12,24,4]        8 / (12 / (24 / (4 / 2)))
Output: 8.0
```

```
1 + (2 + (3 + (4 + 5)))
1 + (2 + (3 + 9))
1 + (2 + 12)
1 + 14
15

8 / (12 / (24 / (4 / 2)))
8 / (12 / (24 / 2))
8 / (12 / 12)
8 / 1
8
```

http://zvon.org/other/haskell/Outputprelude/foldr_f.html

Non-terminating expression (1)

```
foldr (||) True $ repeat False              -- never terminates
```

```
False || (False || (False || …           ))
                                   || True
```

Infinitely check if there is any True,

But never reach the end

```
foldr (||) False $ repeat True              -- terminates with True
```

```
True || (True || (True || …              ))
                                   || False
```

There is at least one True,

Therefore return with true

https://stackoverflow.com/questions/7960543/why-does-this-haskell-code-not-terminate

Non-terminating expression (2)

```
foldr (||) True $ repeat False          -- never terminates
```

```
foldr (||) False $ repeat True          -- terminates with True
```

The first expands to **False || (False || (False || ...))**,
while the second expands to **True || (True || (True || ...))**.

The second argument to **foldr** is a red herring -
it occurs in the innermost application of **||**, not the **outermost**,
so it can never actually be reached.

The 2nd argument **True** is occurs
In the innermost application of **||**
The 2nd argument **False** is occurs
In the innermost application of **||**

A red herring is something that misleads or distracts from a relevant or important question.

https://stackoverflow.com/questions/7960543/why-does-this-haskell-code-not-terminate

Non-terminating expression (2)

bot **=** **bot**

bot is a **non-terminating expression**.

Abstractly, we denote the **value**
of a **non-terminating expression**
as _|_ (read "**bottom**").

$$\begin{tikzcd}
\text{bot = bot} & & \\
 & \text{bot = bot} \arrow[ul] & \\
 & & \text{bot = bot} \arrow[ul] \\
 & & \vdots \arrow[u]
\end{tikzcd}$$

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

Termination Checkers

Does function f terminate?

A) {Yes, Don’t know}

Typically look for decreasing size

- **Primitive recursive**
- **Walther recursion**
- **Size change termination**

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Termination Checkers

```
fib :: Integer -> Integer
fib(1) = 1
fib(2) = 1
fib(n) = fib(n-1) + fib(n-2)
```

fib(0) = $\perp^{NT}$

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Values

- **A function only stops terminating when its given a value**
- **Perhaps the question is wrong:**

Q) Given a function f and a value x,

Does f(x) terminate?

Q) Given a function f, for what values of x does

f(x) terminate?

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Non-terminate

```
fib n | n <= 0 =
error “bad programmer!”
```

- A function should never non-terminate
- It should give an helpful error message
- There may be a few exceptions
  - But probably things that can’t be proved
  - i.e. A Turing machine simulator

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Laziness

Haskell is:

- **A functional programming language**
  - **Lazy – not strict**
  - **Only evaluates what is required**
- **Lazy allows:**
  - **Infinite data structures**

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Productivity

[1..] = [1,2,3,4,5,6, ...

- Not terminating
- But is productive
  - Always another element
  - Time to generate “next result” is always finite

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

Evaluation

The blame game

- **last [1..] is ⊥NT**
- **last is a useful function**
- **[1..] is a useful value**

- **Who is at fault?**
    - **The caller of last**

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

A lazy termination checker

- All data/functions must be productive
- Can easily encode termination

```
isTerm :: [a] -> Bool
isTerm [] = True
isTerm (x:xs) = isTerm xs
```

https://ndmitchell.com/downloads/slides-catch-16_mar_2006.pdf

References

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

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