Background – Expressions (1D)

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

This document was produced by using LibreOffice.

Guard operator

patterns are a way of
making sure a **value** conforms to some **form**
and **deconstructing** it

guards are a way of
testing whether some **property** of a **value**
(or several of them) are **true** or **false**.

http://learnyouahaskell.com/syntax-in-functions

while and **let** bindings

Where bindings are a **syntactic construct**
that let you **bind** to **variables** at the end of a function
and the whole function can see them, including all the **guards**.

Let bindings are **expressions** themselves,
let you bind to variables anywhere,
but are very local, so they don't span across **guards**.
Just like any construct in Haskell
that is used to bind values to names,
let bindings can be used for pattern matching.

http://learnyouahaskell.com/syntax-in-functions

while bindings

put the keyword **where** after the **guards**
and define several **names** or **functions**.

all the **names** are aligned at a single column.

These names are visible across the **guards**
and removes redundancy.

The **names** we define in the where section of a function
are only visible to that function,
the namespace of other functions are not contaminated.

where bindings aren't shared
across **function bodies** of **different patterns**.
If you want several **patterns** of one function
to access some shared name, you have to define it **globally**.

You can also use where bindings to **pattern match**!

http://learnyouahaskell.com/syntax-in-functions

let bindings

let <bindings> in <expression>

The **names** that you define in the **let** part
are accessible to the expression after the **in** part.

the **names** are also aligned in a single column

let puts the bindings first and
the **expression** that uses them later
whereas **where** is the other way around.

let bindings are **expressions** themselves.
where bindings are just **syntactic constructs**.

http://learnyouahaskell.com/syntax-in-functions

case expression

case expressions are, well, **expressions**,
much like **if else expressions** and **let** bindings.

Not only can we evaluate **expressions**
based on the possible cases of the value of a variable,
we can also do **pattern matching**

taking a variable,
pattern matching it,
evaluating pieces of code based on its value,
where have we heard this before?

pattern matching on parameters in function definitions!
Well, that's actually just syntactic sugar for case expressions.

http://learnyouahaskell.com/syntax-in-functions

guard as an **expression**

```
case () of
  _ | a >= x     -> 1
    | a == b     -> 333
    | otherwise -> 5
```

guard as an expression

```
    | a >= x     = 1
    | a == b     = 333
    | otherwise = 5
```

guard

```
if  | a >= x     -> 1
    | a == b     -> 333
    | otherwise -> 5
```

multi-way if

using the **MultiWayIf** extension

https://stackoverflow.com/questions/10370346/using-guards-in-let-in-expressions

Guard operator

```
f x
  | predicate1 = expression1
  | predicate2 = expression2
  | predicate3 = expression3
```

Examples)

```
absolute x = if (x<0) then (-x) else x
```

```
absolute x
  | x<0          = -x
  | otherwise    = x
```

no equals sign on the first line of the function definition
but an equals sign after each guard.

https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27226

Guard operator – otherwise

The **otherwise** guard should always be last, it's like the default in a C switch statement.

catch all guard

more readable than **if/then/else** for more than two conditional cases

```
score :: Int -> String
score x
  | x > 90 = show (x) ++ ": A"
  | x > 80 = show (x) ++ ": B"
  | x > 70 = show (x) ++ ": C"
  | otherwise = show(x) ++ ": F"
```

https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27226

Guard operator – where

```
holeScore :: Int -> Int -> String
holeScore strokes par
  | strokes < par       = show (par-strokes) ++ " under par"
  | strokes == par      = "level par"
  | strokes > par       = show(strokes-par) ++ " over par"
```

```
holeScore :: Int -> Int -> String
holeScore strokes par
  | score < 0       = show (abs score) ++ " under par"
  | score == 0      = "level par"
  | otherwise       = show(score) ++ " over par"
 where score = strokes-par
```

https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27226

Case expression

```
data Pet = Cat | Dog | Fish

hello :: Pet -> String
hello x =
  case x of
    Cat   -> "meeow"
    Dog   -> "woof"
    Fish  -> "bubble"
```

```
case x of
  pattern -> value
  pattern -> value
  pattern -> value
```

https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27226

Case expression – a pattern having a variable

```
data Pet = Cat | Dog | Fish | Parrot String

hello :: Pet -> String
hello x =
  case x of
    Cat    -> "meeow"
    Dog    -> "woof"
    Fish   -> "bubble"
    Parrot name  -> "pretty"  + name


hello (Parrot "polly")
"pretty polly"
```

https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27226

Case expression – a default pattern

```
data Pet = Cat | Dog | Fish | Parrot String

hello :: Pet -> String
hello x =
  case x of
    Parrot name  -> "pretty " ++ name
    _            -> "grunt"
```

https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27226

Select expression

a function implemented in Haskell:

```
select :: a -> [(Bool, a)] -> a
select def = maybe def snd . List.find fst
        -- = fromMaybe def . lookup True
        -- = maybe def id . lookup True

select exDefault
    [(cond1, ex1),
     (cond2, ex2),
     (cond3, ex3)]
```

Unfortunately this function is <u>not</u> in the **Prelude**.

It is however in the utility-ht package.

https://wiki.haskell.org/Case

Advantages of **let**

P1

```
f :: s -> (a,s)
f x = y
  where y = ... x ...
```

P2

```
f :: State s a
f = State $ \x -> y
  where y = ... x ...
```

Using Control.Monad.State monad

P2 will not work, because **where** refers to the **pattern matching** **f =**, where no **x** is in scope.

with **let**, there is no problem.

P3

```
f :: s -> (a,s)
f x =
  let y = ... x ...
  in  y
```

P4

```
f :: State s a
f = State $ \x ->
  let y = ... x ...
  in  y
```

https://wiki.haskell.org/Let_vs._Where

Advantages of **while**

Because "**where**" blocks are bound to a syntactic construct, they can be used to share bindings between parts of a function that are not syntactically **expressions**.

```
f x
  | cond1 x    = a
  | cond2 x    = g a
  | otherwise = f (h x a)
  where
    a = w x
```

```
f x
  = let a = w x
    in case () of
      _ | cond1 x    -> a
        | cond2 x    -> g a
        | otherwise -> f (h x a)
```

an expression style

```
f x =
  let a = w x
  in  select (f (h x a))
        [(cond1 x, a),
         (cond2 x, g a)]
```

a functional equivalent:

```
f x =
   let a = w x
   in if cond1 x
     then a
     else if cond2 x
     then g a
     else f (h x a)
```

a series of if-then-else expressions:

these alternatives are arguably less readable and hide the structure of the function more than simply using **where**

https://wiki.haskell.org/Let_vs._Where

Lambda Lifting

let or **where** can often be implemented
using **lambda lifting** and **let floating**,
incurring at least the cost of introducing a new name.

```
f x
  | cond1 x   = a
  | cond2 x   = g a
  | otherwise = f (h x a)
  where
    a = w x
```

a : a free variable

```
f x = f' (w x) x
```

```
f' a x
  | cond1 x   = a
  | cond2 x   = g a
  | otherwise = f (h x a)
```

a : an argument

lambda lifting:

turning free variables into arguments

The auxiliary definition can either be a top-level binding,
or included in f using **let** or **where**

https://wiki.haskell.org/Let_vs._Where

Let-floating transformation

let-oating transformations:

floating inwards moves bindings as far inwards as possible

```
let x = y+1
in case z of
  [] -> x*x
  (p:ps) -> 1
```

```
case z of
  [] -> let x = y+1
              in x*x
  (p:ps) -> 1
```

the **full laziness** transformation floats selected bindings
outside enclosing lambda abstractions

```
f = \xs -> letrec
              g = \y -> let n = length xs
                          in ...g...n...
              in  ...g...
```

```
f = \xs -> let n = length xs
              in letrec g = \y -> ...g...n...
                    in ...g...
```

local transformations fine-tune" the location of bindings

https://www.microsoft.com/en-us/research/wp-content/uploads/1996/05/float.pdf

Eta Conversion

An **eta conversion** (**η-conversion**) is
adding or dropping of **abstraction** over a function.

the following two values are equivalent under **η-conversion**:
\x -> abs x
abs

an eta reduction

\x -> abs x ⇒ **abs**

an eta abstraction (expansion)

abs ⇒ **\x -> abs x**

Extensive use of η-reduction can lead to **Pointfree** programming.
It is also typically used in certain **compile-time optimisations**.

https://wiki.haskell.org/Let_vs._Where

Eta Expansion

```
fib = (map fib' [0 ..] !!)
  where
    fib' 0 = 0
    fib' 1 = 1
    fib' n = fib (n - 1) + fib (n - 2)
```

```
fib x = map fib' [0 ..] !! x
  where
    fib' 0 = 0
    fib' 1 = 1
    fib' n = fib (n - 1) + fib (n - 2)
```

the second one runs considerably slower than the first.
You may wonder why simply adding an **explicit argument** to **fib**
(known as **eta expansion**) degrades performance so dramatically.

In the first version
fib' is a **global constant** that never changes,
and you're just indexing into that.

In the second version,
fib is a function that constructs
a new and different **fib'** for every value of **x**.

```
Prelude> [11, 22, 33, 44, 55] !! 0
11
Prelude> [11, 22, 33, 44, 55] !! 1
22
Prelude> [11, 22, 33, 44, 55] !! 4
55
```

https://wiki.haskell.org/Let_vs._Where

Problems with where (2)

```
fib =
    let fib' 0 = 0
        fib' 1 = 1
        fib' n = fib (n - 1) + fib (n - 2)
    in  (map fib' [0 ..] !!)
```

```
fib x =
    let fib' 0 = 0
        fib' 1 = 1
        fib' n = fib (n - 1) + fib (n - 2)
    in  map fib' [0 ..] !! x
```

In the second case, **fib'** is redefined for every argument **x**
The compiler cannot know whether you intended this –
while it increases **time complexity** it may reduce **space complexity**.
Thus it will not float the definition out from under the binding of x.

In contrast, in the first function, **fib'**
can be moved to the top level by the compiler.
The **where** clause hid this structure and made the application to x
look like a plain **eta expansion**, which it is not.

https://wiki.haskell.org/Let_vs._Where

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf