1 Slides

1.1 Week 2

1.1.1 Lectures

1.1.2 Patterns

Pattern Matching

- Inputs: a pattern, and a Haskell expression
- Output: either a "fail", or a "success", with a binding from each variable to the part of the expression it matched.

Example:

pattern `('c':zs)`,

matches expression `'c':('a':('t':[]))`,

with `zs` bound to `'a':('t':[])`

Patterns in Haskell

- We can build patterns from atomic values, variables, and certain kinds of "constructions".
- An atomic value, such as `3`, `'a'` or `"abc"` can only match itself
- A variable, or the wildcard `_`, will match anything.
- A "construction" is either:
  1. a tuple such as `(a,b)` or `(a,b,c)`, etc.,
  2. a list built using `[]` or `:`,
  3. or a user-defined datatype.
- A construction pattern matches if all its sub-components match.

Pattern Sequences

- Function definition equations may have a sequence of patterns (e.g., the `and` function example.)
- Each pattern is matched against the corresponding expression, and all such matches must succeed. *One* binding is returned for all of the matches
- Any given variable may only occur once in any pattern sequence (it can be reused in a different equation.).

Pattern Examples

- Expect three arbitrary arguments (of the appropriate type!)

```
myfun x y z = whatever   -- binding x y z
```

- Illegal — if we want first two arguments to be the same then we need to use a conditional (somehow).

```
myfun x x z = whatever          -- Can't bind x twice here !
myfun x y z | x == y = whatever -- much better !
```

- First argument must be zero, second is arbitrary, and third is a non-empty list.

```
myfun 0 y (z:zs) = whatever   -- binding y z zs
```

- First argument must be zero, second is arbitrary, and third is a non-empty list, whose first element is character 'c'

```
myfun 0 y ('c':zs) = whatever -- binding y zs
```

- First argument must be zero, second is arbitrary, and third is a non-empty list, whose tail is a singleton.

```
myfun 0 y (z:[z']) = whatever -- binding y z z'
```

Pattern Matching (summary)

- Pattern-matching can *succeed* or *fail*.
- If successful, a pattern match returns a (possibly empty) *binding*.
- A binding is a mapping from (pattern) variables to values.
- Examples:

Patterns	Values	Outcome
x (y:ys) 3	99 [] 3	Fail
x (y:ys) 3	99 [1,2,3] 3	Ok, $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$
x (3:ys) 3	99 [3,2,1] 3	Ok, $x \mapsto 99, ys \mapsto [2,1]$

- Binding $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$ can also be written as a (simultaneous) *substitution* $[99, 1, [2,3]/x, y, ys]$ where 99, 1 and $[2,3]$ replace x, y and ys respectively.

1.1.3 Function Notation (again)

Identifiers vs. Operators

Names of things in Haskell come in a number of varieties, including:

Variable Identifiers Names that start with a *lowercase* alphabet character, followed by zero or more alphanumeric characters, plus underscores (a.k.a. $varid$).

Variable Symbols Names that start with a symbolic character, followed by zero or more symbolic characters (a.k.a. $varsym$).

Function Notation (V)

We can define and use functions whose names are either $varid$ or $varsym$.

For $varid$ names, the function definition uses "prefix" notation, where the function name appears before the arguments:

```
myfun x y  =  x+y+y                    myfun 57 42
```

For $varsym$ names, the function definition uses "infix" notation, where the function has exactly two arguments and the name appears inbetween the arguments:

```
x +++ y  =   x+y+y                          57 +++ 42
```

Function Notation (VI)

For $varid$ names, with functions having two arguments, we can define and use them "infix-style" by surrounding them with backticks:

```
x `plus2` y  =  x+y+y                       57 `plus2` 42
```

For $varsym$ names, we can define and use them "prefix-style" by enclosing them in parentheses:

```
(++++) x y  =   x+y+y                       (++++) 57 42
```

We can define one way and use the other—all these are valid:

```
57 `myfun` 42            (+++) 57 42
plus2 57 42               57 ++++ 42
```

They all have type `a -> a -> a`.

leap-years again!

We can rewrite `mod y num` as `y `mod` num`, so the leap-year predicate becomes:

```
leapyear :: Int -> Bool
leapyear y | y `mod` 400 == 0  =  True   -- 2000 was
           | y `mod` 100 == 0  =  False  -- 1900 wasn't
           | y `mod` 4 == 0    =  True   -- 2020 was
           | otherwise         =  False  -- 2022 isn't
```

More about infix operators

- It is easy to define our own infix operators
- However, how should we interpret `u +++ v + x +++ y` ?
- There are many possibilites:
- ```
  (u+++v) + (x+++y)
  u +++ ((v+x) +++y )
  (u +++ (v+x)) +++ y
  ```

- Most languages use notions of operator precedence and associative "fixity" to handle this.
- Haskell allows you *specify* this:

```
infixr 8 ^    -- '^' level 8,  a^b^c  = a^(b^c)
infixl 7 *    -- '*' level 7, a*b*c = (a*b)*c
infix  4 <    -- '<' level 4, a<b<c illegal
-- by above, a*b^c^d*e*f<g is interpreted as
--  ( ((a * (b^(c^d))) * e) * f ) < g
```

The Haskell Report

- Official Reference: "Haskell 2010 Language Report"
  - Online: `http://www.haskell.org/onlinereport/haskell2010/`
- In this course we refer to sections of that report thus:
  - [H2010 Sec 3.4]
  - Haskell 2010 Language Report, Section 3.4

1.1.4 Prelude Extracts

The Haskell Prelude [H2010 Sec 9]

- The "`Standard Prelude`" is a library of functions loaded automatically (by default) into any Haskell program.
- Contains most commonly used datatypes and functions
- [H2010 Sec 9] is a specification of the Prelude the actual code is compiler dependent
```

Prelude extracts (I)

Haskell allows the programmer to define new *infix* operators. It can be useful to also specify their precedence and associativity.

- Infix declarations

```
infixr 9  .
infixr 8  ^, ^^, ..
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
infixr 5  :, ++
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

  Higher precedence numbers bind tighter. Function application binds tightest of all

Prelude extracts (II)

- Numeric Functions

```
subtract  :: (Num a) => a -> a -> a
even, odd :: (Integral a) => a -> Bool
gcd       :: (Integral a) => a -> a -> a
lcm       :: (Integral a) => a -> a -> a
(^)       :: (Num a, Integral b) => a -> b -> a
(^^)      :: (Fractional a, Integral b) => a -> b -> a
```

The `Num`, `Integral` and `Fractional` annotations have to do with *type-classes* — see later.

Prelude extracts (III)

- Boolean Type & Functions

```
data Bool  =  False | True
(&&), (||) :: Bool -> Bool -> Bool
not        :: Bool -> Bool
otherwise  :: Bool
```

Prelude extracts (IV)

- List Functions

```
map    :: (a -> b) -> [a] -> [b]
(++)   :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
concat :: [[a]] -> [a]
head   :: [a] -> a
tail   :: [a] -> [a]
null   :: [a] -> Bool
length :: [a] -> Int
(!!)   :: [a] -> Int -> a
repeat :: a -> [a]
take   :: Int -> [a] -> [a]
drop   :: Int -> [a] -> [a]
elem   :: Eq a => a -> [a] -> Bool
```

Prelude extracts (V)

- Function Functions

```
id        :: a -> a
const     :: a -> b -> a
(.)       :: (b -> c) -> (a -> b) -> a -> c
flip      :: (a -> b -> c) -> b -> a -> c
seq       :: a -> b -> b
($), ($!) :: (a -> b) -> a -> b
```

We will re-visit these later — note that type-polymorphism here means that the possible implementations of some of these are extremely constrained!

1.1.5 Writing Functions (I)

Writing Functions (I) — using other functions

(Examples from Chp 4, Programming in Haskell, 2nd Ed., Graham Hutton 2016)

- Function `even` returns true if its integer argument is even

  ```
  even n = n `mod` 2 == 0
  ```

  We use the modulo function `mod` from the Prelude

- Function `recip` calculates the reciprocal of its argument

  ```
  recip n = 1/n
  ```

  We use the division function `/` from the Prelude

- Function call `splitAt n xs` returns two lists, the first with the first `n` elements of `xs`, the second with the rest of the elements

  ```
  splitAt n xs = (take n xs, drop n xs)
  ```

  We use the list functions `take` and `drop` from the Prelude

1.2 Examples and HOFs

Higher Order Functions

What is the difference between these two functions?

```
add x y = x + y
add2 (x, y) = x + y
```

We can see it in the types; `add` takes one argument at a time, returning a function that looks for the next argument.

This concept is known as "Currying" after the logician Haskell B. Curry.

```
add :: Integer -> (Integer -> Integer)
add2 :: (Integer, Integer) -> Integer
```

The function type arrow associates to the right, so `a -> a -> a` is the same as `a -> (a -> a)`.

In Haskell functions are *first class citizens*. In other words, they occupy the same status in the language as values: you can pass them as arguments, make them part of data structures, compute them as the result of functions...

```
add3 :: (Integer -> (Integer -> Integer))
add3 = add
```

```
> add3 1 2
3

(add3) 1 2
  ==> add 1 2
  ==> 1 + 2
```

Notice that there are no parameters in the definition of `add3`.

A function with multiple arguments can be viewed as a function of one argument, which computes a new function.

```
add 3 4
  ==> (add 3) 4
  ==> ((+) 3) 4
```

The first place you might encounter this is the notion of *partial application*:

```
increment :: Integer -> Integer
increment = add 1
```

If the type of `add` is `Integer -> Integer -> Integer`, and the type of `add 1 2` is `Integer`, then the type of `add 1` is? It is `Integer -> Integer`

Some more examples of partial application:

An infix operator can be partially applied by taking a *section*:

```
increment = (1 +) -- or (+ 1)

addnewline = (++"\n")
```

```
double :: Integer -> Integer
double = (*2)
```

```
> [ double x | x <- [1..10] ]
[2,4,6,8,10,12,14,16,18,20]
```

Functions can be taken as parameters as well.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)

addtwo = twice increment
```

Here we see functions being defined as functions of other functions!

Function Composition (I)

In fact, we can define function composition using this technique:

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)

twice2 f = f `compose` f
```

We can use an infix operator definition for compose, even though it takes three arguments, rather than two.

```
(f ! g) x = f (g x)
twice3 f = f!f
```

We just bracket the infix application and apply that to the last (`x`) argument.

Function Composition (II) [H2010 Sec 9]

Function composition is in fact part of the Haskell Prelude:

```
(.) :: (b -> c) -> (a -> b) -> a -> c

(f . g) x = f (g x)
```

We can define functions without naming their inputs, using composition (and other HOFs)

```
second :: [a] -> a
second = head . tail
```

```
> second [1,2,3]
2
```

Í

1.2.1 Writing Functions (IIa)

Writing Functions (II) — using recursion

- We shall show how to write the functions `take` and `drop` using recursion.
- We shall consider what this means for the execution efficiency of `splitAt`.
- We then do a direct recursive implementation of `splitAt` and compare.

Implementing `take`

- `take :: Int -> [a] -> [a]` Let `xs1 = take n xs` below. Then `xs1` is the first `n` elements of `xs`. If `n <= 0` then `xs1 = []`. If `n >= length xs` then `xs1 = xs`.
- ```
  take n _ | n <= 0  =  []
  take _ []          =  []
  take n (x:xs)      =  x : take (n-1) xs
  ```
- How long does `take n xs` take to run? (we count function calls as a proxy for execution time)
- It takes time proportional to `n` or `length xs`, whichever is shorter.
  ```

Implementing `drop`

- `drop :: Int -> [a] -> [a]` Let `xs2 = drop n xs` below. Then `xs2` is `xs` with the first `n` elements removed. If `n <= 0` then `xs2 = xs`. If `n >= length xs` then `xs2 = []`.
- ```
  drop n xs | n <= 0  =  xs
  drop _ []           =  []
  drop n (x:xs)       =  drop (n-1) xs
  ```
- How long does `drop n xs` take to run?
- It takes time proportional to `n` or `length xs`, whichever is shorter.

1.2.2 Writing Functions (IIb)

Implementing `splitAt` recursively

- `splitAt :: Int -> [a] -> ([a],[a])` Let `(xs1,xs2) = splitAt n xs` below. Then `xs1` is the first `n` elements of `xs`. Then `xs2` is `xs` with the first `n` elements removed. If `n >= length xs` then `(xs1,xs2) = (xs,[])`. If `n <= 0` then `(xs1,xs2) = ([],xs)`.
- ```
  splitAt n xs | n <= 0  =  ([],xs)
  splitAt _ []           =  ([],[])
  splitAt n (x:xs)       = splitMerge x (splitAt (n-1) xs)
  splitMerge x (xs1,xs2) = ( x:xs1, xs2 )
  ```
- How long does `splitAt n xs` take to run?
- It takes time proportional to `n` or `length xs`, whichever is shorter, which is twice as fast as the version using `take` and `drop` explicitly!

Switcheroo!

- Can we implement `take` and `drop` in terms of `splitAt`?
- Hint: the Prelude provides the following:

  ```
  fst :: (a,b) -> a
  snd :: (a,b) -> b
  ```

- Solution:

```
take n xs = fst (splitAt n xs)
drop n xs = snd (splitAt n xs)
```

- How does the runtime of these definitions compare to the direct recursive ones?

1.2.3 Haskell vs. Prolog

Lists: `Haskell` **vs.** `Prolog`

Mathematically we might write lists as items separated by commas, enclosed in angle-brackets

$$\sigma_0 = \langle\rangle \quad \sigma_1 = \langle 1\rangle \quad \sigma_2 = \langle 1,2\rangle \quad \sigma_3 = \langle 1,2,3\rangle$$

Haskell

```
s0 = []
s1 = 1:[]   or  [1]
s2 = 1:2:[]   or  [1,2]
s3 = 1:2:3:[] or [1,2,3]
```

Prolog

```
S0 = []
S1 = [1]
S2 = [1,2]
```

```
1:2:3:[] is really (1:(2:(3:[])))
```

Patterns

```
[]  (x:xs)  (x:y:xs)
    [x]     [x,y]
```

```
[] [X|Xs] [X,Y|Xs]
   [X]    [X,Y]
```

1.2.4 Prelude Lists (A)

Function: `head`

`head xs` returns the first element of `xs`, if non-empty

Type Signature

```
head  ::  [a] -> a
```

Non-Empty List

```
head (x:_)  =  x
```

Empty List

```
head []  =  error "Prelude.head: empty list"
```

We have to fail in the last case because there is no way to generate a value `v` of type `a`, where `a` can be any possible type, if there is no such value input to the function. Empty list `[]`, of type `a`, contains no value of type `a` !

Undefinedness in Haskell

- Sometimes a Haskell function is *partial*: it doesn't return a value for some input, because it can't without violating type restrictions.
- Haskell provides two ways to explicitly define such a undefined "value":

  ```
  undefined :: a
  error :: String -> a
  ```

  Evaluating either of these results in a run-time error
- There are two ways in which "undefined" can occur implicitly:
  - If we use pattern-matching that is incomplete, so that some input values fail to match.
  - if a recursive function fails to terminate
- When talking about the meaning of Haskell, it is traditional to use the symbol $\perp$, a.k.a. "bottom", to denote undefinedness.

Why Not define a default value for `head []`?

- Why don't we define a default value for each type (`default ::a`) so that we can define (possible using Haskell classes):

```
head [] = default -- for any given type a
```

  rather than having head [] $= \perp$?

- Why not have `default` for `Int` equal to 0?
- A key design principle behind Haskell libraries and programs is to have programs (functions!) that obey nice obvious laws:

```
xs = head xs : tail xs
sum (xs ++ ys) = sum xs + sum ys
product (xs ++ ys) = product xs * product ys
```

- Consider the product law if `default = 0` and `xs = []`, and assume that both `sum` and `product` use `default` for the empty list case. Lefthand side is then `product ys` while thre righthand side is zero.

Function: `tail`

`tail xs`, for non-empty `xs` returns it with first element removed

Type Signature

```
tail  ::  [a] -> [a]
```

Non-Empty List

```
tail (_:xs)  =  xs
```

Empty List

```
tail []  =  error "Prelude.tail: empty list"
```

Here again, we have tail [] $= \perp$.

`tail [] /= []` — Why Not?

- Why don't we define `tail [] = []`? The typing allows it.
- Consider the following law, `xs = head xs : tail xs`, given that `xs` is `tail []`
- ```
    tail [] = head ( tail[]) : (tail : tail [])
  =   [] = head [] : tail []
  =   [] = ⊥ : []
  ```
  We have managed to show that the empty list is the same as a singleton list containing an undefined element.
- "Obvious" fixes can have unexpected consequences.

Function: `last`

`last xs` returns the last element of `xs`, if non-empty

Type Signature

```
last  ::  [a] -> a
```

Singleton List

```
last [x]  =  x
-- must occur before (_:xs) clause
```

Non-Empty List

```
last (_:xs)  =  last xs
```

Empty List

```
last []  =  error "Prelude.last: empty list"
```
```

Function: `init`

`init xs`, for non-empty `xs` returns it with last element removed

Type Signature

```
init  ::  [a] -> [a]
```

Singleton List

```
init [x]  =  []
```

Non-Empty List

```
init (x:xs)  =  x : init xs
```

Empty List

```
init []  =  error "Prelude.init: empty list"
```

1.2.5 Prelude Lists (B)

Function: `null`

`null xs` returns `True` if the list is empty

Type Signature

```
null  ::  [a] -> Bool
```

Empty List

```
null []  =  True
```

Non-Empty List

```
null (_:_)  =  False
```

Function: `++`

`xs ++ ys` joins lists `xs` and `ys` together.

Type Signature

```
(++)   :: [a] -> [a] -> [a]
```

Empty List

```
[] ++ ys  =  ys
```

Non-Empty List

```
(x:xs) ++ ys  =  x : (xs ++ ys)
```

Evaluating: `++`

```
   (1:2:3:[]) ++ (4:5:[])
= -- Non-Empty List, x -> 1, xs -> 2:3:[]
   1 : ( (2:3:[]) ++ (4:5:[]) )
= -- Non-Empty List, x -> 2, xs -> 3:[]
   1 : ( 2: ( (3:[]) ++ (4:5:[]) ) )
= -- Non-Empty List, x -> 3, xs -> []
   1 : ( 2: (3: ([] ++ (4:5:[]) ) ) )
= -- Empty List, ys -> 4:5:[]
   1 : ( 2: (3: (4 : 5 :[])))
```

Note that the time taken is proportional to the length of the first list, and independent of the size of the second.

Function: `reverse` (slow)

`reverse xs`, reverses the list `xs`

Type Signature

```
reverse  ::  [a] -> [a]
```

Empty List

```
reverse []  =  []
```

Non-Empty List

```
reverse (x:xs)  =  reverse xs ++ [x]
```

Evaluating: `reverse`

```
   reverse (1:2:3:[])
= -- Non-Empty List, x -> 1, xs -> 2:3:[]
   reverse (2:3:[]) ++ [1]
= -- Non-Empty List, x -> 2, xs -> 3:[]
   (reverse (3:[]) ++ [2]) ++ [1]
= -- Non-Empty List, x -> 3, xs -> []
   ((reverse [] ++ [3]) ++ [2]) ++ [1]
= -- Empty List,
   (([] ++ [3]) ++ [2]) ++ [1]
= -- after many concatenations
   3:2:1:[]
```

This is a bad way to do reverse (why?)

Function: `reverse` **(fast)**

`reverse xs`, reverses the list `xs`

Type Signature

```
reverse  ::  [a] -> [a]
```

Use Helper Function (???)

```
reverse xs = rev [] xs
```

Helper: Non-Empty List

```
rev sx (x:xs) = rev (x:sx) xs
```

Helper: Empty List

```
rev sx [] = sx
```

Evaluating: `reverse`, again

```
   reverse (1:2:3:[])
= -- ???
   rev [] (1:2:3:[])
= -- Non-Empty List, sx -> [], x -> 1, xs -> 2:3:[]
   rev (1:[]) (2:3:[])
= -- Non-Empty List, sx -> 1:[], x -> 2, xs -> 3:[]
   rev (2:1:[]) (3:[])
= -- Non-Empty List, sx -> 2:1:[], x -> 3, xs -> []
   rev (3:2:1:[]) []
= -- Empty List, sx -> 3:2:1:[]
   3:2:1:[]
```

Much faster (why?)

Function: `reverse` (Prelude Version, [H2010 Sec 9.1])

`reverse xs`, reverses the list `xs`

Type Signature

```
reverse  ::  [a] -> [a]
```

!!!! ???

```
reverse  =  foldl (flip (:)) []
```

The Prelude doesn't always give the most obvious definition of a function's behaviour !

Function: `(!!)`

`(!!)  xs n`, or `xs !!  n` selects the `n`th element of list `xs`, provided it is long enough. Indices start at 0.

Fixity and Type Signature

```
infixl 9 !!
(!!)  ::  [a] -> Int -> a
```

Negative Index

```
xs !! n | n < 0
   =  error "Prelude.!!: negative index"
```

Empty List

```
[] !! _  =  error "Prelude.!!: index too large"
```

Zero Index `(x:_) !! 0  =  x`

Non-Zero Index `(_:xs) !! n  =  xs !! (n-1)`

1.2.6 Lexical and Syntactical Matters (I)

Haskell is Case-Sensitive [H2010 Sec 2.4]

For example, the following names are all *different*:

```
ab aB Ab AB
```

Program Structure [H2010 Sec 1.1]

A Haskell script can be viewed as having four levels:

1. A Haskell program is a set of modules, that control namespaces and software re-use in large programs.
2. A module consists of a collection of declarations, defining ordinary values, datatypes, type classes, and fixity information.
3. Next are expressions, that denote values and have static types.
4. At the bottom level is the lexical structure, capturing the concrete representation of programs in text files.

(We focus on the bottom three for now).

Notational Conventions [H2010 Sec 2.1]

- The report uses the following notation for syntax:

$[syn]$	optional occurence of syn
$\{syn\}$	zero or more repetitions of syn
(syn)	grouping
$syn_1 \vert syn_2$	choice between alternatives
$syn_{\langle syn' \rangle}$	difference—elements generated by syn, except those generated by syn'
`fibonacci`	terminal syntax in `typewriter font`

- It uses BNF-like syntax, with productions of the form:

$$nonterm \quad \rightarrow \quad alt_1 \vert alt_2 \vert \ldots \vert alt_n$$

  "$nonterm$ is either an alt_1 or alt_2 or ..."

- The trick is distinguishing | (alternative separator) from `|` , the vertical bar character (and similarly for characters {} [] ()).

Comments [H2010 Sec 2.3]

A Haskell script has two kinds of comments:

1. End-of-line comments, starting with `--`.

2. Nested Comments, started with {- and ending with -}

Example, where comments are in red.

```
myfun x -- end-of-line, but -} won't end it
 = let
     y = 2 {- nested, but -- ignored here -} ; z = 3
     {-
     a = 4 {- was 42 but I changed my mind -}
     b = 5
     -}
   in y + z * x
```

Namespaces [H2010 Sec 1.4]

- Six kinds of names in Haskell:
  1. Variables, denoting values;
  2. (Data-)Constructors, denoting values;
  3. Type-variables, denoting types;
  4. Type-constructors, denoting 'type-builders';
  5. Type-classes, denoting groups of 'similar' types;
  6. Module-names, denoting program modules.
- Two constraints (only) on naming:
  - Variables (1) and Type-variables (3) begin with lowercase letters or underscore, Other names (2,4,5,6) begin with uppercase letters.
  - An identifier cannot denote both a Type-constructor (4) and Type-class (5) in the same scope.
- So the name `Thing` (*e.g.*) can denote a module, data-constructor, and either a class or type-constructor in a single scope.

Character Types (I) [H2010 Sec 2.2]

The characters can be grouped as follows:

- *special* : ( ) , ; [ ] ` { }

- *whitechar* → *newline*|*vertab*|*space*|*tab*
- *small* → a|b|...|z|_
- *large* → A|B|...|Z
- *digit* → 0|1|...|9
- *symbol* : ! # % & * + . / < = > ? @ \ ^ | - ~
- the following characters are not explicitly grouped- : " '

(There is also stuff regarding Unicode characters (beyond ASCII) that we shall ignore—so the above is not exactly as shown in [H2010 Sec 2.2]).

Lexemes (I) [H2010 Sec 2.4]

The term "lexeme" refers to a single basic "word" in the language.

- Variable Identifiers (*varid*) start with lowercase and continue with letters, numbers, underscore and single-quote. `x x' a123 myGUI _HASH very_long_Ident_indeed''`
- Constructor Identifiers (*conid*) start with uppercase letters and continue with letters, numbers, underscore and single-quote. `T Tree Tree' My_New_Datatype Variant123`
- Variable Operators (*varsym*) start with any symbol, and continue with symbols and the colon. `<+> |:| ++ + - ==> == && #!#`
- Constructor Operators (*consym*) start with a colon and continue with symbols and the colon. `:+: :~ :=== :$%&`

Identifiers (*varid*, *conid*) are usually prefix, whilst operators (*varsym*,*consym*) are usually infix.

Lexemes (II) [H2010 Sec 2.4]

- Reserved Identifiers (*reservedid*):

```
case class data default deriving do else foreign if
import in infix infixl infixr instance let module
newtype of then type where _
```

- Reserved Operators (*reservedop*): `.. : :: = \ | <- -> @ ~ =>`

Literals [H2010 Sec 2.5,2.6]

We give a simplified introduction to literals (actual basic values)

- Integers (*integer*) are sequences of digits Examples: `0 123`
- Floating-Point (*float*) has the same syntax as found in mainstream programming languages. `0.0 1.2e3 1.4e-45`
- Characters (*char*) are enclosed in single quotes and can be escaped using backslash in standard ways. `'a' '$' '\'' '"' '\64' '\n'`
- Strings (*string*) are enclosed in double quotes and can also be escaped using backslash in standard ways. `"Hello World" "I 'like' you" "\" is a dbl-quote" "line1\nline2"`

1.2.7 Lexical and Syntactical Matters (II)

Operators [H2010 Sec 3]

- Expressions can built up as expected in many programming languages

```
3    x    x+y    (x<=y)    a + c * d - (e * (a / b))
```

- Some operators are left-associative like `+ - * /` : `a + b + c` parses as `(a + b) + c`
- Some operators are right-associative like `: . ^ && ||`: `a:b:c:[]` parses as `a:(b:(c:[]))`
- Other operators are non-associative like `== /= < <= >= >` : `a <= b <= c` is illegal, but `(a <= b) && (b <= c)` is ok.
- The minus sign is tricky: `e - f` parses as "`e` subtract `f`", `(- f)` parses as "minus `f`", but `e (- f)` parses as "function `e` applied to argument minus `f`"

Function Application/Types

- Function application is denoted by juxtaposition, and is left associative

- `f x y z` parses as `((f x) y) z`
- If we want `f` applied to both `x`, and to the result of the application of `g` to `y`, we must write `f x (g y)`
- In types, the function arrow is right associative `Int -> Char -> Bool` parses as `Int -> (Char -> Bool)`
- The type of a function whose first argument is itself a function, has to be written as `(a -> b) -> c`
- Note the following types are identical: `(a -> b) -> (c -> d)` `(a -> b) -> c -> d`

Sections [H2010 Sec 3.5]

- A "section" is an operator, with possibly one argument surrounded by parentheses, which can be treated as a prefix function name.
- `(+)` is a prefix function adding its arguments (e.g. `(+) 2 3 = 5`)
- `(/)` is a prefix function dividing its arguments (e.g. `(/) 2.0 4.0 = 0.5`)
- `(/4.0)` is a prefix function dividing its single argument by 4 (e.g. `(/4.0) 10.0 = 2.5`)
- `(10.0/)` is a prefix function dividing 10 by its single argument (e.g. `(10/) 4.0 = 2.5`)
- `(- e)` is not a section, use `subtract e` instead. (e.g. `(subtract 1) 4 = 3`)

1.3 Lexical and Syntactical Matters (III)

Value Declarations [H2010 Sec 4.4.3]

- A value declaration can be either a function or a pattern.
- We have already seen function declarations, which involve patterns

```
funname patn_1 patn_2 ... patn_n = expr
```

- A declaration can also have a lhs that is a single pattern:

```
patn = expr
```

The simplest example is defining a variable to denote a (fixed) value:

```
molue = 42
```

- But we can also match complicated patterns against equally complicated expressions:

```
(a,b,c)  =  (1,2,3)
[d,e,f]  =  take 3 [1..10]
(before,after) = splitAt 42 someBigList
```

Haskell Layout Rule [H2010 Sec 2.7]

- Some Haskell syntax specifies lists of declarations or actions as follows: $\{item_1 ; item_2 ; item_3 ; \ldots ; item_n\}$
- In some cases (after keywords `where`, `let`, `do`, `of`), we can drop $\{$, $\}$ and `;`.
- The layout (or "off-side") rule takes effect whenever the open brace is omitted.
  - When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
  - For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted);
  - if it is indented the same amount, then a new item begins (a semicolon is inserted);
  - and if it is indented less, then the layout list ends (a close brace is inserted).

Layout Example

Offside rule (silly) example: consider $\textbf{let } x = y + 3 \wedge z = 10 \wedge f(a) = a + 2z \textbf{ in } f(x)$

- Full syntax:

```
let {  x = y + 3 ; z = 10; f a = a + 2 * z } in f x
```

- Using Layout:

```
let x = y + 3
    z = 10
    f a = a + 2 * z
in f x
```

- Using Layout (alternative):

```
let
  x =  y + 3
  z = 10
  f a
   = a + 2 * z
in f x
```

Local Declarations [H2010 Sec 3.12]

- A let-expression has the form:

$$\texttt{let}\ \{d_1\texttt{;}\ \ldots\texttt{;}d_n\}\texttt{in}\ e$$

  d_i are declarations, e is an expression. *The offside-rule applies.*

- Scope of each d_i is e and righthand side of all the d_is (mutual recursion)
- Example: $ax^2 + bx + c = 0$ means $x = \frac{-b \pm (\sqrt{b^2 - 4ac})}{2a}$
- ```
  solve a b c
   = let twoa = 2 * a
         discr = b*b - 2 * twoa * c
         droot = sqrt discr
     in ((droot-b)/twoa , negate ((droot+b)/twoa))
  ```

Local Declarations [H2010 Sec 3.12]

- A where-expression has the form:

$$\texttt{where}\ \{d_1\texttt{;}\ \ldots\texttt{;}d_n\}$$

  d_i are declarations. *The offside-rule applies.*
  ```

- Scope of each d_i is the expression that *precedes* `where` and righthand side of all the d_is (mutual recursion)
- ```
  solve a b c
   = ((droot-b)/twoa , negate ((droot+b)/twoa))
   where
     twoa = 2 * a
     discr = b*b - 2 * twoa * c
     droot = sqrt discr
  ```

`let` ([H2010 Sec 3.12]) vs. `where` [H2010 Sec 4.?]

- What is the difference between `let` and `where` ?
- The `let ...in ...` is a full expression and can occur anywhere an expression is expected.
- The `where` keyword occurs at certain places in *declarations*

  $$\ldots \texttt{where}\{d_1 ; \ldots ; d_n\}$$

  of
  - case-expressions [H2010 Sec 3.13]
  - `module`s [H2010 Sec 4]
  - `class`es [H2010 Sec 4.3.1]
  - `instance`s [H2010 Sec 4.3.2]
  - function and pattern righthand sides (rhs) [H2010 Sec 4.4.3]
- Both allow mutual recursion among the declarations.

Conditionals [H2010 Sec 3.6]

- For expressions, we can write a conditional using `if` ... `then`... `else`

  $$exp \quad \rightarrow \quad \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp$$

- The else-part is compulsory, and cannot be left out (why not?)
- The (boolean-valued) expression after `if` is evaluated: If true, the value is of the expression after `then` If false, the value is of the expression after `else`
  ```

Case Expression §3.13

- A case-expression has the form:

  $$\texttt{case}\ e\ \texttt{of}\ \{p_1\ \texttt{->}\ e_1\texttt{;}\ \ldots\ \texttt{;}p_n\ \texttt{->}\ e_n\}$$

  p_i are patterns, e_i are expressions. *The offside rule applies.*

```
odd x =
  case (x `mod` 2) of
    0 -> False
    1 -> True

vowel x =
  case x of
    'a' -> True
    'e' -> True
    'i' -> True
    'o' -> True
    'u' -> True
    _   -> False
```

```
empty x =
  case x of
  [] -> True
  _  -> False
```