## Lexical Syntax

### Comments
- `--`  end-of-line comment
- `(/-)`  mutli-line comment [with nesting- `)-`
- `{-# #}#`  [Pragma usually a helpful hint to the compiler 

### Identifier names
- `eat3Chars`  functions, variables and type variables start with lowercase
- `Double`  concrete typenames / constructors start with uppercase
- `a`  typically, variable names in argument positions are short
- `foo_Bar34baz`  underscores _ , primes ', digits, mixed case, are permitted

### Symbols
- `++`  symbols are infix operator names, ++ takes two arguments
- `::`  symbols starting with a colon : are infix constructor names
- `(++)`  an infix symbol can be used prefix, by enclosing in parents
- `foo`  a prefix name can be used infix, by enclosing in backquotes

### Strings
- `'c'`  character constants use single quotes
- `(||)`  symbols are infix operator names, ++ takes two arguments

### Lists
- `[]`  a list with element type a (a stands for any type)
- `(a,b)`  a pair of types a and b (a and b are type variables)
- `[2..16]`  list containing a stepped numeric range
- `[40,39 .. 0]`  a list containing a stepped numeric range
- `[x,y,z]`  a list of three things
- `x`  a paired value - in round parentheses with commas
- `x,y,z`  a triple of values

### Numbers
- `42`  value of any number type: Int, Integer, Float, Double, etc
- `42.0`  value of any fractional type: Float, Double, Rational, Complex

### Equals Symbols
- `=`  single = is a definition of a value
- `==`  double == is a comparison operator returning a Boolean

### Lambda notation
- `(x-> foo)`  backslash is a poor ASCII version of the lambda symbol
- `->`  ASCII version of a right arrow (used in lambdas, case discrimination, and types of functions)

### Layout
- `def`  indentation is used intuitively to indicate logical structuring: anything indented right to the right "belongs" in this group
- `def; defn`  Indentation can be overridden by using explicit braces and semicolons.

## Expressions

### Function application
- `f x`  space between function name f and argument expression x
- `f $ x`  function f applied to expression x (but right-associative)
- `x ++ y`  operators (symbols) are applied in infix
- `(+) x y`  an infix operator can be applied prefix by enclosing in parents
- `f (3+4) (not y)`  a prefix function can be applied infix, enclosed in backquotes
- `f (x+1)`  a function/operator can be partially applied to only some args

### Anonymous functions
- `lx -> expr`  a lambda function
- `\(x -> x+3) 5`  this anonymous function names its argument x
- `\(x+1) -> expr`  this anonymous function pattern-matches its list argument

### Data construction
- `Build (1+2) True`  Values are built by applying a data constructor as a function

### Local naming
- `let `  a function which can only be used within the given
- `let expr in expr`  the rhs, whose result is a list. Pattern-match the components of the list, then use the names x and xs within the expr

### Conditionals
- `if (then expr else expr)`  a catch-all default case is called
- `case expr of case of expr | a = rhs1 | b = rhs2 | ...`  the condititons are equivalent to a single `if` statement, but the `case` form can be used in `where` clauses

### Pattern-matching and binding
- `f (C a b)`  pattern-matches their arguments. A pattern is an application of a constructor to either literal values, fresh variable names, or other patterns.
- `f (C x:y:z:)`  patterns are nested. The value of the rest of the list after application of a constructor to either literal values, fresh variable names, or other patterns.

### Sequencing evaluation
- `do pat <- iocomp`  evaluate the side-effecting computation iocomp, and pattern-match its result against pat. for use in later actions. subsequent actions are indented to match the first one. actions can use variables bound by patterns higher up.
- `f (x:xs)`  function pattern-matches the arguments. A pattern is an application of a constructor to either literal values, fresh variable names, or other patterns.

## Definitions

### Function definition (function names start with a lower-case letter)
- `f t`  the function named f "has type" t. Known as a type signature.
- `f arg0 arg1 = rhs`  function named f has two named arguments, result is rhs
- `f (x:y:z) = rhs`  function-pattern-matches on its list argument, naming its parts
- `f x = rhs
  where rhs = expr`  an equational definition can have local definitions contained in an indented "where" clause
- `f n = n + 0 = rhsNeg
  | n + 0 = rhsPos`  guards on equations: tests are indented with vertical bar. there are multiple right-hand-sides, each guarded by a test

### Type definition (type names and constructors start with an Upper-case letter)
- `data T = C a b`  user-defined datatype T takes a type parameter 'a' values of type T are constructed using C values of type T contain one value of type 'a' and an Int
- `data U = V | W | X`  user-defined datatype U values of type U can be either a V construction, W, or X
- `type M = T Bool`  M is a synonym for T Bool - the names are interchangeable
- `newtype N = N (T U)`  N is like a synonym for (T, U), except the names are not interchangeable

### Other top-level definitions
- `module M where`  every module has a capitalised name
- `import Data.Word`  import and use functions from another module
- `class C where method : type`  define a predicate over types. class methods are indented, and must give a type signature
- `instance C (Int where method = imp)`  instance of a class predicate for a specific type. the class method definition is indented - no type signature

### Basic types
- `Int`  limited precision signed integers (e.g. 32 bit)
- `Integer`  arbitrary precision signed integers
- `Rational`  arbitrary precision fractional numbers
- `Float`  floating-point limited-precision fractional numbers
- `Double`  double-word floating-point limited-precision fractional numbers
- `Bool`  Boolean constants: True, False
- `Char`  single Unicode characters
- `String`  textual sequence of characters (+ [Char ])

### Bigger types
- `[a,b]`  pair of types a and b (a and b are type variables)
- `[a]`  list with element type a (a stands for any type)
- `a -> b`  function with argument type a, result type b
- `a -> b -> c`  function with two arguments, of types a and b, result type c