CPSC 370 **Haskell Syntax** (v1) *UNBC*
Fall 2023

Booleans *are* `True` *and* `False`
Characters (`Char`) `'c'`, `'\n'`, `'\''`, *etc.*, ...
Strings (`String` = `[Char]`)
`"hello"`, `['h','e','l','l','o']`, . `'h':"ello"` .

`--` line comment;
`{-` ... `-}` ; block comments. `{- {-` *they nest* `-} -}`
`{-#` ... `#-}` ; language pragmas.
Haddock comments use "`-- |`", "`-- ^`", "`{- |`", or "`{- ^`".

Keywords `class`, `data`, `do`, `else`, `of`, `import`, `if`, `in`, `instance`, `let`, `module`, `newtype`, `then`, `type`, `where`

Reserved operators :
`..`, `:`, `::`, `=`, `\`, `|`, `<-`, `->`, `@`, `~`, `=>`

Constructor operators start with ':'.

Identifiers

- wild card: `_`
- variables: `[a-z][a-zA-Z0-9_']*`
- Constructors (types, modules): `[A-Z][a-zA-Z0-9_']*`
- Operators: `[!#$%&*+-./<=>?@\^|~:]+`
- A variable in back-ticks acts like an operator: "`f 3 2`" or "`3 `f` 2`".
- An operator in parenthesis acts like a prefix function: "`(+) 3 2`".

Basic Types:

`Int`, `Integer`, `Float`, `Double`, `Rational`, `Bool`, `Char`.

Type constructors: `[]` (lists); `->` (functions); `(,)`, `(,,)`, ... (tuples), `Maybe` (*e.g.*, `Maybe Int`); `Either` (*e.g.*, `Either String Int`);

Type classes

Show can convert to a string
Read can convert from a string
Eq supports `==` and `/=`
Ord supports `<`, `>=`, ...
Num supports basic arithmetic.
... `Fuctor`, `Applicative`, `Monad`, `Foldable`, `Traversable`

User Defined Types

`type` creates aliases. `type String = [Char]; type Array a = [[a]]`

`newtype` creates a distinct type for an old idea.

```
newtype Distance = Metres { unMetres :: Double }
```

`data` creates brand new types.

```
data Colour = Chartreuse | Vermillion | Fuschia
data Tree a = Node (Tree a) a (Tree a) | EmptyTree
data Box  a = B a
```

CPSC 370 Haskell Syntax (v1) *UNBC*
Fall 2023

Overall file structure

```
module ModuleName (
  export list) where
imports
definitions
```

Imports

- `import ModuleName` ..... *everything from* `ModuleName`
- `import ModuleName(mx1,mx2)` ..... *only* `mx1`, `mx2` *from* `ModuleName`
- `import ModuleName hiding(mx1,mx2)` ....*all but* `mx1`, `mx2` *from* `ModuleName`
- `import qualified ModuleName` ..... *must use* `ModuleName.mx1`, *and so on.*
- `import ModuleName as M`
- `import qualified ModuleName as M` ..... *can use* `M.mx1`, `M.mx2` *and so on.*

Syntactic blocks

- `let { pat = expr [ ; pat = expr ]* } in expr`
- `case expr of { pat -> expr [ ; pat -> expr ]* }`
- `do { [ do-stat ; ]* m-expr }`
  where `do-stat` is one of
  - `pat <- m-expr`
  - `let pat = expr`
  - `m-expr`

  and `m-expr` is an expression of a monadic type.

Lists

- all elements have the same type
- (explicit) `["cat", "house"]`
- constructors are `:` and `[]`. `("cat":"house":[])`
- enumeration: `[1 .. 4]` same as `1:[2 .. 4]` same as `[1,2,3,4]`
- comprehension: `[2*x-1 | x<-[1..50]]`, `[x | x<-[1..100], x `mod` 2==1]`