

# PROGRAMMING LANGUAGES: FUNCTIONAL PROGRAMMING

## 2. INTRODUCTION TO HASKELL: SIMPLE DATATYPES & FUNCTIONS ON LISTS

---

Shin-Cheng Mu

Spring 2022

National Taiwan University and Academia Sinica

## SIMPLE DATATYPES

---

# BOOLEANS

The datatype *Bool* can be introduced with a *datatype declaration*:

```
data Bool = False | True
```

(But you need not do so. The type *Bool* is already defined in the Haskell Prelude.)

## DATATYPE DECLARATION

- In Haskell, a **data** declaration defines a new type.

$$\begin{array}{l} \mathbf{data} \text{ Type} = \text{Con}_1 \text{ Type}_{11} \text{ Type}_{12} \dots \\ \quad \quad \quad | \text{Con}_2 \text{ Type}_{21} \text{ Type}_{22} \dots \\ \quad \quad \quad | \quad \quad \quad : \end{array}$$

- The declaration above introduces a new type, *Type*, with several cases.
- Each case starts with a constructor, and several (zero or more) arguments (also types).
- Informally it means “a value of type *Type* is either a *Con*<sub>1</sub> with arguments *Type*<sub>11</sub>, *Type*<sub>12</sub>..., or a *Con*<sub>2</sub> with arguments *Type*<sub>21</sub>, *Type*<sub>22</sub>...”
- Types and constructors begin in capital letters.

## FUNCTIONS ON BOOLEANS

Negation:

*not* :: *Bool* → *Bool*

*not False* = *True*

*not True* = *False*

- Notice the definition by *pattern matching*. The definition has two cases, because *Bool* is defined by two cases. The shape of the function follows the shape of its argument.

## FUNCTIONS ON BOOLEANS

Conjunction and disjunction:

$(\wedge), (\vee) \quad :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\text{False} \wedge x = \text{False}$

$\text{True} \wedge x = x$

$\text{False} \vee x = x$

$\text{True} \vee x = \text{True}$

I use the symbols  $\wedge$  and  $\vee$  due to mathematical convention. In your Haskell code,  $\wedge$  should be written `&&`, and  $\vee$  should be `||`.

## FUNCTIONS ON BOOLEANS

Equality check:

$$(\equiv), (\neq) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$$
$$x \equiv y = (x \wedge y) \vee (\text{not } x \wedge \text{not } y)$$
$$x \neq y = \text{not } (x \equiv y)$$

- $=$  is a definition, while  $\equiv$  is a function.
- $\equiv$  and  $\neq$  are written respectively written `==` and `/=` in ASCII.

## EXAMPLE

*leapyear*  $:: \text{Int} \rightarrow \text{Bool}$

*leapyear* *y* = (*y* 'mod' 4 == 0) ∧  
(*y* 'mod' 100 ≠ 0 ∨ *y* 'mod' 400 == 0)

- Note: *y* 'mod' 100 could be written *mod* *y* 100. The backquotes turns an ordinary function to an infix operator.
- It's just personal preference whether to do so.



## CHARACTERS

- You can think of *Char* as a big **data** definition:

**data** *Char* = 'a' | 'b' | ...

with functions:

*ord* :: *Char* → *Int*

*chr* :: *Int* → *Char*

- Characters are compared by their order:

*isDigit* :: *Char* → *Bool*

*isDigit* *x* = '0' ≤ *x* ∧ *x* ≤ '9'

## EQUALITY CHECK

- Of course, you can test equality of characters too:

$(==) :: Char \rightarrow Char \rightarrow Bool$

- $(==)$  is an *overloaded* name — one name shared by many different definitions of equalities, for different types:
  - $(==) :: Int \rightarrow Int \rightarrow Bool$
  - $(==) :: (Int, Char) \rightarrow (Int, Char) \rightarrow Bool$
  - $(==) :: [Int] \rightarrow [Int] \rightarrow Bool \dots$
- Haskell deals with overloading by a general mechanism called *type classes*. It is considered a major feature of Haskell.
- While the type class is an interesting topic, we might not cover much of it since it is orthogonal to the central message of this course.

# TUPLES

- The polymorphic type  $(a, b)$  is essentially the same as the following declaration:

**data** *Pair* *a b* = *MkPair* *a b*

- Or, had Haskell allow us to use symbols:

**data**  $(a, b) = (a, b)$

- Two projections:

*fst*  $:: (a, b) \rightarrow a$

*fst*  $(a, b) = a$

*snd*  $:: (a, b) \rightarrow b$

*snd*  $(a, b) = b$

## FUNCTIONS ON LISTS

---

## LISTS IN HASKELL

- Traditionally an important datatype in functional languages.
- In Haskell, all elements in a list must be of the same type.
  - `[1, 2, 3, 4] :: List Int`
  - `[True, False, True] :: List Bool`
  - `[[1, 2], [], [6, 7]] :: List (List Int)`
  - `[] :: List a`, the empty list (whose element type is not determined).
- *String* is an abbreviation for *List Char*; `"abcd"` is an abbreviation of `['a', 'b', 'c', 'd']`.

## LIST AS A DATATYPE

- $[] :: \text{List } a$  is the empty list whose element type is not determined.
- If a list is non-empty, the leftmost element is called its *head* and the rest its *tail*.
- The constructor  $(:) :: a \rightarrow \text{List } a \rightarrow \text{List } a$  builds a list. E.g. in  $x : xs$ ,  $x$  is the head and  $xs$  the tail of the new list.
- You can think of a list as being defined by

**data**  $\text{List } a = [] \mid a : \text{List } a$

- $[1, 2, 3]$  is an abbreviation of  $1 : (2 : (3 : []))$ .

## HEAD AND TAIL

- $head :: List\ a \rightarrow a$ . e.g.  $head\ [1,2,3] = 1$ .
- $tail :: List\ a \rightarrow List\ a$ . e.g.  $tail\ [1,2,3] = [2,3]$ .
- $init :: List\ a \rightarrow List\ a$ . e.g.  $init\ [1,2,3] = [1,2]$ .
- $last :: List\ a \rightarrow a$ . e.g.  $last\ [1,2,3] = 3$ .
- They are all partial functions on non-empty lists. e.g.  $head\ [] = \perp$ .
- $null :: List\ a \rightarrow Bool$  checks whether a list is empty.

## LIST GENERATION

- `[0..25]` generates the list `[0, 1, 2..25]`.
- `[0, 2..25]` yields `[0, 2, 4..24]`.
- `[2..0]` yields `[]`.
- The same works for all *ordered* types. For example *Char*:
  - `['a'..'z']` yields `['a', 'b', 'c'..'z']`.
- `[1..]` yields the *infinite* list `[1, 2, 3..]`.



## LIST COMPREHENSION

- Some functional languages provide a convenient notation for list generation. It can be defined in terms of simpler functions.
- e.g.  $[x \times x \mid x \leftarrow [1..5], \text{odd } x] = [1, 9, 25]$ .
- Syntax:  $[e \mid Q_1, Q_2..]$ . Each  $Q_i$  is either
  - a generator  $x \leftarrow xs$ , where  $x$  is a (local) variable or pattern of type  $a$  while  $xs$  is an expression yielding a list of type  $List\ a$ , or
  - a guard, a boolean valued expression (e.g.  $\text{odd } x$ ).
  - $e$  is an expression that can involve new local variables introduced by the generators.

## LIST COMPREHENSION

Examples:

- $[(a, b) \mid a \leftarrow [1..3], b \leftarrow [1..2]] = [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]$
- $[(a, b) \mid b \leftarrow [1..2], a \leftarrow [1..3]] = [(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2)]$
- $[(i, j) \mid i \leftarrow [1..4], j \leftarrow [i + 1..4]] = [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]$
- $[(i, j) \mid i \leftarrow [1..4], \text{even } i, j \leftarrow [i + 1..4], \text{odd } j] = [(2, 3)]$

## TWO MODES OF PROGRAMMING

- Functional programmers switch between two modes of programming.
  - Inductive/recursive mode: go into the structure of the input data and recursively process it.
  - Combinatorial mode: compose programs using existing functions (combinators), process the input in stages.
- We will try the latter style today. However, that means we have to familiarise ourselves to a large collection of library functions.
- In the next lecture we will talk about how these library functions can be defined, in the former style.

## LENGTH AND INDEXING

- $(!!) :: List\ a \rightarrow Int \rightarrow a$ . List indexing starts from zero. e.g.  $[1, 2, 3]!!0 = 1$ .
- $length :: List\ a \rightarrow Int$ . e.g.  $length\ [0..9] = 10$ .

## APPEND AND CONCATENATION

- Append:  $(++) :: \text{List } a \rightarrow \text{List } a \rightarrow \text{List } a$ . In ASCII one types  $(++)$ .
  - $[1, 2] ++ [3, 4, 5] = [1, 2, 3, 4, 5]$
  - $[] ++ [3, 4, 5] = [3, 4, 5] = [3, 4, 5] ++ []$
- Compare with  $(:) :: a \rightarrow \text{List } a \rightarrow \text{List } a$ . It is a type error to write  $[] : [3, 4, 5]$ .  $(++)$  is defined in terms of  $(:)$ .
- $\text{concat} :: \text{List } (\text{List } a) \rightarrow \text{List } a$ .
  - e.g.  $\text{concat } [[1, 2], [], [3, 4], [5]] = [1, 2, 3, 4, 5]$ .
  - $\text{concat}$  is defined in terms of  $(++)$ .

## TAKE AND DROP

- *take*  $n$  takes the first  $n$  elements of the list.
  - For example, *take* 0  $xs = []$
  - *take* 3 "abcde" = "abc"
  - *take* 3 "ab" = "ab"
- Working with infinite list: *take* 5 [1..] = [1,2,3,4,5]. Thanks to normal order (lazy) evaluation.
- Dually, *drop*  $n$  drops the first  $n$  elements of the list.
  - For example, *drop* 0  $xs = xs$
  - *drop* 3 "abcde" = "cd"
  - *drop* 3 "ab" = ""
- *take*  $n$   $xs ++ \text{drop } n$   $xs = xs$ , as long as  $n \neq \perp$ .

## MAP AND $\lambda$

- $map :: (a \rightarrow b) \rightarrow List\ a \rightarrow List\ b$ . e.g.  
 $map\ (1+) [1, 2, 3, 4, 5] = [2, 3, 4, 5, 6]$ .
- $map\ square [1, 2, 3, 4] = [1, 4, 9, 16]$ .
- Every once in a while you may need a small function which you do not want to give a name to. At such moments you can use the  $\lambda$  notation:
  - $map\ (\lambda x \rightarrow x \times x) [1, 2, 3, 4] = [1, 4, 9, 16]$
  - In ASCII  $\lambda$  is written `\`.
- $\lambda$  is an important primitive notion. We will talk more about it later.

## FILTER

- $filter :: (a \rightarrow Bool) \rightarrow List\ a \rightarrow List\ a$ .
  - e.g.  $filter\ even\ [2, 7, 4, 3] = [2, 4]$
  - $filter\ (\lambda n \rightarrow n \text{ 'mod' } 3 == 0)\ [3, 2, 6, 7] = [3, 6]$
- Application: count the number of occurrences of ' $a$ ' in a list:
  - $length \cdot filter\ ('a' ==)$
  - Or  $length \cdot filter\ (\lambda x \rightarrow 'a' == x)$
- **Note** a list comprehension can always be translated into a combination of primitive list generators and *map*, *filter*, and *concat*.



## ZIP

- $zip :: List\ a \rightarrow List\ b \rightarrow List\ (a, b)$
- e.g.  $zip\ "abcde"\ [1,2,3] = [('a',1), ('b',2), ('c',3)]$
- The length of the resulting list is the length of the shorter input list.

## POSITIONS

- Exercise: define *positions*  $:: \text{Char} \rightarrow \text{String} \rightarrow \text{List Int}$ , such that *positions* *x xs* returns the positions of occurrences of *x* in *xs*. E.g. *positions* 'o' "roodo" = [1, 2, 4].
- *positions* *x xs* = *map snd (filter ((x ==) · fst) (zip xs [0..]))*
- Or,  
*positions* *x xs* = *map snd (filter (λ(y, i) → x == y) (zip xs [0..]))*
- What if you want only the position of the *first* occurrence of *x*?

*pos*  $:: \text{Char} \rightarrow \text{String} \rightarrow \text{Int}$   
*pos* *x xs* = *head (positions x xs)*

- Due to lazy evaluation (normal order evaluation), *positions* of the other occurrences are *not* evaluated!

## MORALS OF THE STORY

- Lazy evaluation helps to improve modularity.
  - List combinators can be conveniently re-used. Only the relevant parts are computed.
- The combinator style encourages “wholemeal programming”.
  - Think of aggregate data as a whole, and process them as a whole!

## $\lambda$ EXPRESSIONS

---

## $\lambda$ EXPRESSIONS

- $\lambda x \rightarrow e$  denotes a function whose argument is  $x$  and whose body is  $e$ .
- $(\lambda x \rightarrow e_1) e_2$  denotes the function  $(\lambda x \rightarrow e_1)$  applied to  $e_2$ . It can be reduced to  $e_1$  with its *free* occurrences of  $x$  replaced by  $e_2$ .
- E.g.

$$\begin{aligned} & (\lambda x \rightarrow x \times x) (3 + 4) \\ = & (3 + 4) \times (3 + 4) \\ = & 49 . \end{aligned}$$

- $\lambda$  expression is a primitive and essential notion. Many other constructs can be seen as syntax sugar of  $\lambda$ 's.
- For example, our previous definition of *square* can be seen as an abbreviation of

*square* ::  $Int \rightarrow Int$

*square* =  $\lambda x \rightarrow x \times x$  .

- Indeed, *square* is merely a value that happens to be a function, which is in turn given by a  $\lambda$  expression.
- $\lambda$ 's are like all values — they can appear inside an expression, be passed as parameters, returned as results, etc.
- In fact, it is possible to build a complete programming language consisting of only  $\lambda$  expressions and applications. Look up “ $\lambda$  calculus”.

- $\lambda x \rightarrow \lambda y \rightarrow e$  is abbreviated to  $\lambda x y \rightarrow e$ .
- The following definitions are all equivalent:

*smaller*  $x\ y = \text{if } x \leq y \text{ then } x \text{ else } y$

*smaller*  $x = \lambda y \rightarrow \text{if } x \leq y \text{ then } x \text{ else } y$

*smaller*  $= \lambda x \rightarrow \lambda y \rightarrow \text{if } x \leq y \text{ then } x \text{ else } y$

*smaller*  $= \lambda x y \rightarrow \text{if } x \leq y \text{ then } x \text{ else } y$  .