Student Online Teaching Advice Notice

0.1 Class 7

0.1.1 Lists in Haskell

Lists [H2010 Sec 3.7]

- The type "list of `t`" is written as `[t]`.
- Fundamentally lists are built from "nil" (`[]`) and "cons" (`:`).
- A *value* of type `[t]` is either:
  - The empty list, indicated by the (nullary)*data constructor* `[]`.
  - A non-empty list which is built from a value of type `t` (`v` say) and a (pre-existing) value of type`[t]` (`vs` say), using the (binary, infix) *data constructor* `(:)` (giving `(v:vs)`)
- The two list constructors have the following types (for arbitrary type `a`)

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

- Note that the constructors `[]` and `:` are hard-wired into the syntax of Haskell and get special treatment

List Syntactic Sugar [H2010 Sec 3.7]

We use square brackets to provide syntactical sugar in a variety of ways:

- Enumeration: `[a,b,c,d]` for `a:(b:(c:(d:[])))`, also written as `a:b:c:d:[]`
- Ranges: `[4..9]` for `[4,5,6,7,8,9]` also `[4,7..20]` for `[4,7,10,13,16,19]`
- Comprehension: `[ x*x | x <- [1..10], even x]` for `[4,16,36,64,100]`
  Comprehensions are more complex than this (see later, or [H2010 Sec 3.11])
- Strings are a special notation of lists of characters `"Hello"` for `['H','e','l','l','o']`

Special Treatment

Why does this kind of list get special treatment in Haskell?

- It's basically historical, traced back to the ML language developed specifically to write theorem provers.
- ML needs to have a rigourous formal semantics, so a mathematical notion of *algebraic datatypes*, from the field of Abstract Algebras was used.
- The lists in ML and Haskell are such algebraic types. Their use has become ubiquitous in ML/Haskell-style languages
- We shall see later how to "grow our own" in Haskell
- Serious application development in Haskell may not use these lists, and there are packages for static and dynamic arrays available.

0.1.2 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	Prolog

`s0 = []`
`s1 = 1:[]   or   [1]`
`s2 = 1:2:[]   or   [1,2]`
`s3 = 1:2:3:[] or [1,2,3]`
	`S0 = []`
	`S1 = [1]`
	`S2 = [1,2]`

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

Patterns

```
[]  (x:xs)   (x:y:xs)           []  [X|Xs]  [X,Y|Xs]
    [x]      [x,y]                  [X]     [X,Y]
```

0.1.3 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)

- 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!

0.1.4 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"
```

0.2 Class 8

0.2.1 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)`

0.3 Class 9

Observations on Exercise00 - 2020

From an "official" class size of 128, there were 119 submissions (93%)

- Grading:

  Username Marks: +1 (obtained by 117 submissions)

  Test Passed Marks: +9 (obtained by 118 submissions)

  Test Failed Marks: +6 (obtained by 0 submissions)

  Did not compile Marks: +4 (obtained by 1 submission)

  If this was Exercise01 or later, Passing the tests: +10, failing the tests: 0..9, failing to compile: 0.

- Reasons for loss of marks:

  Usernames `memyselfi`

  Failed to Compile Submitted `Main.hs` instead of `Ex00.hs`

0.3.1 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`)

0.4 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
```

0.4.1 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

0.4.2 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.

0.4.3 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)
    = let (xs1,xs2) = splitAt (n-1) xs
      in  (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?