CSCE 314
Programming Languages

Haskell: Defining Functions, List Comprehensions, and Recursive Functions

Dr. Hyunyoung Lee

Outline

- Defining Functions (Ch. 4)
- List Comprehensions (Ch. 5)
- Recursive Functions (Ch. 6)

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions:

`if cond then e1 else e2`

- e1 and e2 must be of the same type
- else branch is always present

```
abs  :: Int -> Int
abs n = if n >= 0 then n else -n

max  :: Int -> Int -> Int
max x y = if x <= y then y else x

take  :: Int -> [a] -> [a]
take n xs = if n <= 0 then []
            else if xs == [] then []
                 else (head xs) : take (n-1) (tail xs)
```

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n >= 0     = n
      | otherwise = -n
```

Prelude:
otherwise = True

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

compare with ...

```
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

Guarded Equations (Cont.)

Guards and patterns can be freely mixed, the first equation whose pattern matches and guard is satisfied is chosen.

```
take  :: Int -> [a] -> [a]
take n _   | n <= 0 = []
take _ []           = []
take n (x:xs)       = x : take (n-1) xs
```

The underscore symbol _ is a wildcard pattern that matches any argument value.

Pattern Matching

- Many functions are defined using pattern matching on their arguments.

```
not       :: Bool -> Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

- Pattern can be a constant value or include one or more variables.

Functions can often be defined in many different ways using pattern matching. For example

```
(&&)              :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
False && _ = False
True  && b = b
```

- Patterns are matched in order. For example, the following definition always returns False:

```
_    && _    = False
True && True = True
```

- Patterns may not repeat variables. For example, the following definition gives an error:

```
b && b = b
_ && _ = False
```

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.

```
[1,2,3,4]
```

Means 1:(2:(3:(4:[]))).

Functions on lists can be defined using x:xs patterns.

```
head         :: [a] → a
head (x:_)    = x

tail         :: [a] → [a]
tail (_:xs)   = xs
```

head and tail map any non-empty list to its first and remaining elements.

is this definition complete?

Note:

- x:xs patterns only match non-empty lists:

```
> head []
Error
```

- x:xs patterns must be parenthesised, because function application has priority over (:). For example, the following definition gives an error:

```
head x:_ = x
```

- Patterns can contain arbitrarily deep structure:

```
f (_: (_, x):_) = x

g [[_]] = True
```

Totality of Functions

- (Total) function maps every element in the function's domain to an element in its codomain.
- Partial function maps zero or more elements in the function's domain to an element in its codomain and can leave some elements undefined.
- Haskell functions can be partial. For example:

```
head (x:_) = x

> head []
*** Exception: Prelude.head: empty list
```

```
> "10elements" !! 10
*** Exception: Prelude.(!!): index too large
```

Lambda Expressions

Functions can be constructed *without naming* the functions by using lambda expressions.

$\lambda x \rightarrow x+x$

This nameless function takes a number x and returns the result x+x.

- The symbol λ is the Greek letter lambda, and is typed at the keyboard as a backslash \.
- In mathematics, nameless functions are usually denoted using the $\mapsto$ symbol, as in $x \mapsto x+x$.
- In Haskell, the use of the λ symbol for nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based.

Why Are Lambda's Useful?

1. Lambda expressions can be used to give a formal meaning to functions defined using currying. For example:

```
add x y  = x + y
square x = x * x
```

means

```
add    = \x -> (\y -> x + y)
square = \x -> x * x
```

2. Lambda expressions can be used to avoid naming functions that are only referenced once. For example:

```
odds n = map f [0..n-1]
         where
             f x = x * 2 + 1
```

can be simplified to

```
odds n = map (\x -> x * 2 + 1) [0..n-1]
```

3. Lambda expressions can be bound to a name (function argument)

```
incrementer = \x -> x + 1
add (incrementer 5) 6
```

Case Expressions

Pattern matching need not be tied to function definitions; they also work with case expressions. For example:

(1)
```
take m ys = case (m, ys) of
              (n, _) | n <= 0 -> []
              (_, [])         -> []
              (n, x:xs)       -> x : take (m-1) xs
```

(2)
```
length [] = 0
length (_:xs) = 1 + length xs
```

using a case expression and a lambda:

```
length = \ls -> case ls of
                  [] -> 0
                  (_:xs) -> 1 + length xs
```

Let and Where

The let and where clauses are used to create a local scope within a function. For example:

(1)

```
reserved s =  -- using let
  let keywords = words "if then else for while"
      relops = words "== != < > <= >="
      elemInAny w [] = False
      elemInAny w (l:ls) = w `elem` l || elemInAny w ls
  in elemInAny s [keywords, relops]
```

```
reserved s =  -- using where
  elemInAny s [keywords, relops]
  where keywords = words "if then else for while"
        . . .
        elemInAny w (l:ls) = w `elem` l || elemInAny w ls
```

(2)

```
unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([], [])
unzip ((a, b):rest) =
  let (as, bs) = unzip rest
  in (a:as, b:bs)
```

Let vs. Where

The `let … in …` is an expression, whereas `where` blocks are declarations bound to the context. For example:

```
f x   -- using where block
   | cond1 x     = a
   | cond2 x     = g a
   | otherwise   = f (h x a)
   where a = w x
```

```
f x   -- using let-in expression
  = let a = w x
    in case () of
         _ | cond1 x    -> a
           | cond2 x    -> g a
           | otherwise -> f (h x a)
```

Sections

An operator written between its two arguments can be converted into a *curried* function written before its two arguments by using parentheses. For example:

```
> 1 + 2
3
```

```
> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses. For example:

```
> (1+) 2
3
```

```
> (+2) 1
3
```

In general, if ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called sections.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

- `(1+)` - successor function (`\y->1+y`)
- `(1/)` - reciprocation function
- `(*2)` - doubling function
- `(/2)` - halving function

Outline

- Defining Functions (Ch. 4)
- List Comprehensions (Ch. 5)
- Recursive Functions (Ch. 6)

List Comprehensions

- A convenient syntax for defining lists
- Set comprehension - In mathematics, the comprehension notation can be used to construct new sets from old sets. E.g.,

  $\{(x^2,y^2) | x \in \{1,2,...,10\}, y \in \{1,2,...,10\}, x^2+y^2 \leq 101\}$
- Same in Haskell: new lists from old lists

```
[(x^2, y^2) | x <- [1..10], y <- [1..10], x^2 + y^2 <= 101]
```

generates:

```
[(1,1),(1,4),(1,9),(1,16),(1,25),(1,36),(1,49),(1,64),(1,81),(1,100),(4,1),(4,4),(4,9),(4,16),
(4,25),(4,36),(4,49),(4,64),(4,81),(9,1),(9,4),(9,9),(9,16),(9,25),(9,36),(9,49),(9,64),
(9,81),(16,1),(16,4),(16,9),(16,16),(16,25),(16,36),(16,49),(16,64),(16,81),(25,1),(25,4),
(25,9),(25,16),(25,25),(25,36),(25,49),(25,64),(36,1),(36,4),(36,9),(36,16),(36,25),
(36,36),(36,49),(36,64),(49,1),(49,4),(49,9),(49,16),(49,25),(49,36),(49,49),(64,1),
(64,4),(64,9),(64,16),(64,25),(64,36),(81,1),(81,4),(81,9),(81,16),(100,1)]
```

List Comprehensions: Generators

- The expression x <- [1..10] is called a generator, as it states how to generate values for x.
  - generators can be infinite, e.g.,

```
> take 3 [x | x <- [1..]]
[1,2,3]
```

- Comprehensions can have multiple generators, separated by commas. For example:

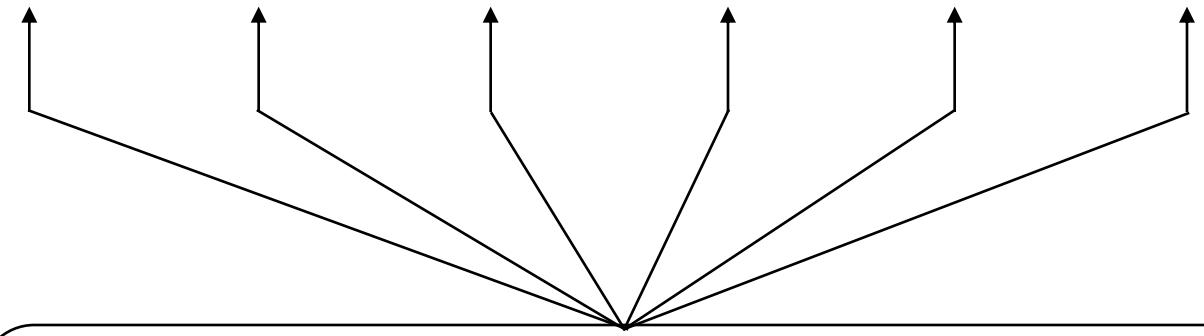
```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

- For example:

```
> [(x,y) | y <- [4,5], x <- [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```



x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x and y are elements of the list [1..3] and y ≥ x.

Using a dependant generator we can define the library function that concatenates a list of lists:

```
concat     :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

```
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x <- [1..10], even x]
```

list all numbers x s.t. x is an element of the list [1..10] and *x is even*

Example: Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

```
> factors 15
[1,3,5,15]
```

A positive integer > 1 is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime  :: Int -> Bool
prime n = factors n == [1,n]
```

```
> prime 15
False

> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes  :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] -> [b] -> [(a,b)]
```

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [0..n], x == x']
                 where n = length xs - 1
```

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

Using zip we can define a function that returns the list of all pairs of adjacent elements from a list:

```
pairs    :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Using pairs we can define a function that decides if the elements in a list are sorted:

```
sorted    :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs xs]
```

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Outline

- Defining Functions (Ch. 4)
- List Comprehensions (Ch. 5)
- Recursive Functions (Ch. 6)

A Function without Recursion

Many functions can naturally be defined in terms of other functions.

```
factorial  :: Int → Int
factorial n = product [1..n]
```

factorial maps any integer n to the product of the integers between 1 and n

Expressions are evaluated by a stepwise process of applying functions to their arguments. For example:

```
  factorial 4
= product [1..4]
= product [1,2,3,4]
= 1*2*3*4
= 24
```

Recursive Functions

Functions can also be defined in terms of themselves. Such functions are called recursive.

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other positive integer to the product of itself and the factorial of its predecessor.

```
factorial 3 = 3 * factorial 2
            = 3 * (2 * factorial 1)
            = 3 * (2 * (1 * factorial 0))
            = 3 * (2 * (1 * 1))
            = 3 * (2 * 1)
            = 3 * 2
            = 6
```

Note:

- The base case factorial 0 = 1 is appropriate because 1 is the identity for multiplication.

- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> factorial (-1)

Error: Control stack overflow
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.
- As we shall see, however, many functions can naturally be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

Recursion on Lists

Lists have naturally a recursive structure. Consequently, recursion is used to define functions on lists.

```
product        :: [Int] → Int
product []      = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

```
product [2,3,4]  =  2 * product [3,4]
                 =  2 * (3 * product [4])
                 =  2 * (3 * (4 * product []))
                 =  2 * (3 * (4 * 1))
                 =  24
```

Using the same pattern of recursion as in product we can define the length function on lists.

```
length        :: [a] → Int
length []      = 0
length (_:xs) = 1 + length xs
```

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

```
   length [1,2,3]
=  1 + length [2,3]
=  1 + (1 + length [3])
=  1 + (1 + (1 + length []))
=  1 + (1 + (1 + 0))
=  3
```

Using a similar pattern of recursion we can define the reverse function on lists.

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

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

```
   reverse [1,2,3]
=  reverse [2,3] ++ [1]
=  (reverse [3] ++ [2]) ++ [1]
=  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=  (([] ++ [3]) ++ [2]) ++ [1]
=  [3,2,1]
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the elements of two lists:

```
zip             :: [a] → [b] → [(a,b)]
zip []     _     = []
zip _      []    = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Remove the first n elements from a list:

```
drop              :: Int → [a] → [a]
drop n xs | n <= 0 = xs
drop _ []          = []
drop n (_:xs)      = drop (n-1) xs
```

- Appending two lists:

```
(++)         :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Laziness Revisited

Laziness interacts with recursion in interesting ways. For example, what does the following function do?

```
numberList xs = zip [0..] xs
```

```
> numberList “abcd”
[(0,’a’),(1,’b’),(2,’c’),(3,’d’)]
```

Laziness with Recursion

Recursion combined with lazy evaluation can be tricky; stack overflows may result in the following example:

```
expensiveLen [] = 0
expensiveLen (_:xs) = 1 + expensiveLen xs
```

```
stillExpensiveLen ls = len 0 ls
          where len z [] = z
                len z (_:xs) = len (z+1) xs
```

```
cheapLen ls = len 0 ls
          where len z [] = z
                len z (_:xs) = let z' = z+1
                               in z' `seq` len z' xs
```

```
> expensiveLen [1..10000000]  -- takes quite long
> stillExpensiveLen [1..10000000]  -- also takes long
> cheapLen [1..10000000]  -- less memory and time
```

Quicksort

The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values $\leq$ the head, sorting the tail values $>$ the head, and then appending the resulting lists on either side of the head value.

Using recursion, this specification can be translated directly into an implementation:

```
qsort          :: [Int] -> [Int]
qsort []        = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a <- xs, a <= x]
      larger  = [b | b <- xs, b > x]
```

Note:

- This is probably the simplest implementation of quicksort in any programming language!

For example (abbreviating qsort as q):