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:

```plaintext
if cond then e1 else e2
```
- `e1` and `e2` must be of the same type
- else branch is always present

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

\[
\text{take} \ ::= \text{Int} \to [a] \to [a] \\
\text{take } n \ _ \ | \ n \leq 0 = [] \\
\text{take } \ _ \ [ ] = [] \\
\text{take } n \ (x:xs) = x :\ \text{take } (n-1) \ xs
\]

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

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

\[
\text{not} :: \text{Bool} \to \text{Bool}
\]
\[
\text{not False} = \text{True} \\
\text{not True} = \text{False}
\]

- 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:

\[(\&\&) \quad :: \quad \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}\]

<table>
<thead>
<tr>
<th>Truth Value 1</th>
<th>Truth Value 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

\[_ \quad && \quad _ \quad = \quad \text{False}\]

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

\[\text{True} \quad && \quad \text{True} \quad = \quad \text{True}\]
\[\_ \quad && \quad \_ \quad = \quad \text{False}\]
\[\text{False} \quad && \quad \_ \quad = \quad \text{False}\]
\[\text{True} \quad && \quad \text{b} \quad = \quad \text{b}\]
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.

\[
[1,2,3,4] \quad \text{Means } 1:(2:(3:(4:[]))).
\]

Functions on lists can be defined using \textit{x:xs} patterns.

\begin{align*}
\text{head} & : \ [a] \to a \\
\text{head} \ (x:_:) & = x \\
\text{tail} & : \ [a] \to [a] \\
\text{tail} \ (_:xs) & = xs
\end{align*}

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:
  
  ```haskell
  > head []
  Error
  ```

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

- Patterns can contain arbitrarily deep structure:
  
  ```haskell
  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 \( \lambda \) is the Greek letter *lambda*, and is typed at the keyboard as a backslash \( \backslash \).

- In mathematics, nameless functions are usually denoted using the \( \mapsto \) symbol, as in \( x \mapsto x+x \).

- In Haskell, the use of the \( \lambda \) 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 \( x = \) \( x \times x \)

   means

   add = \( \lambda x \rightarrow (\lambda y \rightarrow x + y) \)
   square = \( \lambda x \rightarrow x \times x \)
2. Lambda expressions can be used to avoid naming functions that are only referenced once. For example:

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

can be simplified to

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

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

```haskell
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) \[
\text{take } m \ ys = \text{case } (m, \ ys) \ of \\
\quad (n, \ _) \mid n \leq 0 \rightarrow [] \\
\quad (_, \ []) \rightarrow [] \\
\quad (n, \ x:xs) \rightarrow x : \text{take } (m-1) \ xs
\]

(2) \[
\text{length } [] = 0 \\
\text{length } (_:xs) = 1 + \text{length } xs
\]

using a case expression and a lambda:

\[
\text{length } = \backslash l s \rightarrow \text{case } l s \ of \\
\quad [] \rightarrow 0 \\
\quad (_:xs) \rightarrow 1 + \text{length } xs
\]
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]

(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:

\[
f x \quad -- \text{using where block}
\]

\[
| \text{cond1 } x \quad = \ a \\
| \text{cond2 } x \quad = \ g \ a \\
| \text{otherwise} \quad = \ f \ (h \ x \ a)
\]

where \( a = w \ x \)

\[
f x \quad -- \text{using let-in expression}
\]

\[
= \text{let } a = w \ x \\
in \text{case } () \text{ of }
\]

\[
_ \quad | \text{cond1 } x \quad -> \ a \\
| \text{cond2 } x \quad -> \ g \ a \\
| \text{otherwise} \quad -> \ f \ (h \ x \ a)
\]
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:

\[
> \begin{array}{c}
  1 + 2 \\
  3 \\
\end{array}
\quad
> \begin{array}{c}
  (+) 1 2 \\
  3 \\
\end{array}
\]

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

\[
> \begin{array}{c}
  (1+) 2 \\
  3 \\
\end{array}
\quad
> \begin{array}{c}
  (+2) 1 \\
  3 \\
\end{array}
\]

In general, if \(\oplus\) is an operator then functions of the form \((\oplus)\), \((x\oplus)\) and \((\oplus y)\) are called sections.
Why Are Sections Useful?

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

- \((1+)\) - successor function (\(y \rightarrow 1+y\))
- \((1/)\) - 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**

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

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 $x \leftarrow [1..10]$ 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:

\[
> \ [(x,y) \mid y \leftarrow [4,5], \ x \leftarrow [1,2,3]\]
\]

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

\[x \leftarrow [1,2,3]\] 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.

\[(x, y) \mid x \leftarrow [1..3], y \leftarrow [x..3]\]

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 \(y \geq x\).

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

\[
\text{concat} :: [[a]] \rightarrow [a] \\
\text{concat } xss = [x \mid xs \leftarrow xss, x \leftarrow xs]
\]

> 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.

\[ \{ x | x \leftarrow [1..10], \text{even } x \} \]

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:

\[
\text{factors} :: \text{Int} \rightarrow [\text{Int}]
\]

\[
factors \ n = \{ x | x \leftarrow [1..n], \ n \ `\text{mod}` x == 0 \}
\]

> 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:

```haskell
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:

```haskell
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:

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

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

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

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

\[
\text{factorial} :: \text{Int} \rightarrow \text{Int}  \\
\text{factorial} \ n = \text{product} \ [1..n]
\]

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:

\[
\begin{align*}
\text{factorial} \ 4 &= \text{product} \ [1..4] \\
&= \text{product} \ [1,2,3,4] \\
&= 1\times2\times3\times4 \\
&= 24
\end{align*}
\]
Recursive Functions

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

\[
\begin{align*}
\text{factorial } 0 &= 1 \\
\text{factorial } n &= n \times \text{factorial} \ (n-1)
\end{align*}
\]

\[
\begin{align*}
\text{factorial } 3 &= 3 \times \text{factorial } 2 \\
&= 3 \times (2 \times \text{factorial } 1) \\
&= 3 \times (2 \times (1 \times \text{factorial } 0)) \\
&= 3 \times (2 \times (1 \times 1)) \\
&= 3 \times (2 \times 1) \\
&= 3 \times 2 \\
&= 6
\end{align*}
\]

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

```
= 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 \texttt{length} function on lists.

\begin{verbatim}
length :: [a] \rightarrow \text{Int}
length [] = 0
length ((_:xs)) = 1 + length xs
\end{verbatim}

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

\[
\text{length [1,2,3]} \\
\quad = 1 + \text{length [2,3]} \\
\quad = 1 + (1 + \text{length [3]}) \\
\quad = 1 + (1 + (1 + \text{length []})) \\
\quad = 1 + (1 + (1 + 0)) \\
\quad = 3
\]
Using a similar pattern of recursion we can define the reverse function on lists.

\[
\text{reverse} :: [a] \rightarrow [a]
\]

\[
\text{reverse} \ [\] = [\]
\]

\[
\text{reverse} \ (x:xs) = \text{reverse} \ xs \ +\!\!\!\!\!\:+ \ [x]
\]

\[
\text{reverse} \ [1,2,3]
\]

\[
= \text{reverse} \ [2,3] \ +\!\!\!\!\!\:+ \ [1]
\]

\[
= (\text{reverse} \ [3] \ +\!\!\!\!\!\:+ \ [2]) \ +\!\!\!\!\!\:+ \ [1]
\]

\[
= (((\text{reverse} \ [3] \ +\!\!\!\!\!\:+ \ [2]) \ +\!\!\!\!\!\:+ \ [1])
\]

\[
= [3,2,1]
\]
Multiple Arguments

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

- **Zipping the elements of two lists:**
  
  \[
  \begin{align*}
  \text{zip} & \quad :: \ [a] \to [b] \to [(a,b)] \\
  \text{zip} \ [\ ] \ _ & \quad = \ [\ ] \\
  \text{zip} \ _ \ [\ ] & \quad = \ [\ ] \\
  \text{zip} \ (x:\text{xs}) \ (y:\text{ys}) & \quad = \ (x,y) : \text{zip} \ \text{xs} \ \text{ys}
  \end{align*}
  \]

- **Remove the first n elements from a list:**
  
  \[
  \begin{align*}
  \text{drop} & \quad :: \ \text{Int} \to [a] \to [a] \\
  \text{drop} \ n \ \text{xs} \ | \ n \leq 0 & \quad = \ \text{xs} \\
  \text{drop} \ _ \ [\ ] & \quad = \ [\ ] \\
  \text{drop} \ n \ (_:\text{xs}) & \quad = \ \text{drop} \ (n-1) \ \text{xs}
  \end{align*}
  \]

- **Appending two lists:**
  
  \[
  \begin{align*}
  (++) & \quad :: \ [a] \to [a] \to [a] \\
  [\ ] & \quad ++ \ \text{ys} = \ \text{ys} \\
  (x:\text{xs}) \ ++ \ \text{ys} & \quad = \ x : \ (\text{xs} \ ++ \ \text{ys})
  \end{align*}
  \]
Laziness Revisited

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

\[
\text{numberList} \, \text{xs} = \text{zip} \, [0..] \, \text{xs}
\]

\[
> \text{numberList} \, \text{"abcd"} \\
[\text{(0,'a')),(1,'b'),(2,'c'),(3,'d')]]
\]
Laziness with Recursion

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

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

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

```haskell
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 $> \text{the head}$, and then appending the resulting lists on either side of the head value.
Using recursion, this specification can be translated directly into an implementation:

\[
\begin{align*}
qsort \; :: \; [\text{Int}] \rightarrow [\text{Int}] \\
qsort \; [] \; & = \; [] \\
qsort \; (x:xs) \; = \\
& \quad \; \; \; \; \; qsort \; \text{smaller} \; ++ \; [x] \; ++ \; qsort \; \text{larger} \\
& \quad \; \; \; \; \text{where} \\
& \quad \; \; \; \; \quad \; \; \; \; \; \; \; \; \; \; \; \smaller \; = \; [a \mid a \leftarrow xs, \; a \; \leq \; x] \\
& \quad \; \; \; \; \; \; \; \; \; \; \; \; \; \text{larger} \; = \; [b \mid b \leftarrow xs, \; b \; > \; x]
\end{align*}
\]

Note:

- This is probably the simplest implementation of quicksort in any programming language!
For example (abbreviating qsort as q):

\[ q \ [3,2,4,1,5] \]

\[ q \ [2,1] \quad ++ \quad [3] \quad ++ \quad q \ [4,5] \]

\[ q \ [1] \quad ++ \quad [2] \quad ++ \quad q \ [] \]

\[ q \ [] \quad ++ \quad [4] \quad ++ \quad q \ [5] \]

\[ [1] \]

\[ [] \]

\[ [] \]

\[ [5] \]