Chapter 14
Functional Programming

It is better to have 100 functions operate one one data structure, than 10 functions on 10 data structures. A. Perlis

Contents
14.1 Functions and the Lambda Calculus
14.2 Scheme
14.3 Haskell
14.3.1 Introduction
14.3.2 Expressions
14.3.3 Lists and List Comprehensions
14.3.4 Elementary Types and Values
14.3.5 Control Flow
14.3.6 Defining Functions
14.3.7 Tuples
14.3.8 Example: Semantics of Cmite
14.3.9 Example: Symbolic Differentiation
14.3.10 Example: Eight Queens

Overview of Functional Languages

• They emerged in the 1960’s with Lisp
• Functional programming mirrors mathematical functions: domain = input, range = output
• Variables are mathematical symbols: not associated with memory locations.
• Pure functional programming is state-free: no assignment
• Referential transparency: a function’s result depends only upon the values of its parameters.

Lambda Calculus

A lambda expression is a particular way to define a function:
\[ \text{LambdaExpression} \rightarrow \text{variable } \mid (\text{M N}) \mid (\lambda \text{variable . M}) \]

\[ M \rightarrow \text{LambdaExpression} \]
\[ N \rightarrow \text{LambdaExpression} \]

E.g., \((\lambda x . x^2)\) represents the \textit{Square} function.

Properties of Lambda Expressions

In \((\lambda x . M), x \text{ is bound.} \) Other variables in \(M\) are free.
A substitution of \(N\) for all occurrences of a variable \(x\) in \(M\) is written \(M[x \leftarrow N]\). Examples:

\[ x[x \leftarrow y] = y \]
\[ (ax)[x \leftarrow y] = (yy) \]
\[ (ax)[x \leftarrow y] = (xy) \]
\[ (ax)[x \leftarrow y] = (zy) \]
\[ (\lambda x \cdot zx)[x \leftarrow y] = (\lambda u \cdot (zu))[x \leftarrow y] = (\lambda u \cdot (zu)) \]
\[ (\lambda x \cdot zx)[x \leftarrow y] = (\lambda u \cdot (zu))[y \leftarrow z] = (\lambda u \cdot (zu)) \]
Lambda Expressions

A beta reduction \(((\lambda . M)N)\) of the lambda expression \((\lambda . M)\) is a substitution of all bound occurrences of \(x\) in \(M\) by \(N\).

E.g.,
\[((\lambda . x^2)5) = 5^2\]

Function Evaluation

In pure lambda calculus, expressions like \(((\lambda . x^2)5) = 5^2\) are uninterpreted.

In a functional language, \(((\lambda . x^2)5)\) is interpreted normally (25).

Lazy evaluation = delaying argument evaluation in a function call until the argument is needed.

Advantage: flexibility

Eager evaluation = evaluating arguments at the beginning of the call.

Advantage: efficiency

Status of Functions

In imperative and OO programming, functions have different (lower) status than variables.

In functional programming, functions have same status as variables; they are first-class entities.

They can be passed as arguments in a call.

They can transform other functions.

Functional Form

A function that operates on other functions is called a functional form. E.g., we can define
\[g(f, [x1, x2, ...]) = [f(x1), f(x2), ...],\] so that
\[g(Square, [2, 3, 5]) = [4, 9, 25]\]

Minimal Syntax

-- equivalent definitions of factorial -- comment

| fact1 n = if n==0 then 1 else n*fact1(n-1)
| fact2 n |
|        | otherwise = n*fact2(n-1)
| fact3 0 = 1
| fact3 n = n * fact3(n - 1)

14.3 Haskell

A more modern functional language
Many similarities with Lisp and Scheme

Key distinctions:
  - Lazy Evaluation
  - An Extensive Type System
  - Cleaner syntax
  - Notation closer to mathematics
  - Infinite lists

Copyright © 2006 The McGraw-Hill Companies, Inc.
Infinite Precision Integers

Infinite precision integers:
> fact2 30
> 26525285981219105863630848000000

14.3.2 Expressions

Infix notation. E.g.,
5*(4+6)-2 -- evaluates to 48
5^4*2-2 -- evaluates to 78
… or prefix notation. E.g.,
(·)((·)5 (((+)4 6)) 2

Operators

<table>
<thead>
<tr>
<th>!</th>
<th>!!</th>
<th>^</th>
<th>**</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td><code>div</code></td>
<td><code>mod</code></td>
<td><code>rem</code></td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>=</td>
<td>&lt;=</td>
</tr>
</tbody>
</table>
Quotes needed only if used as infix operators.

14.3.3 Lists and List Comprehensions

A list is a series of expressions separated by commas and enclosed in brackets.
The empty list is written [].
evens = [0, 2, 4, 6, 8] declares a list of even numbers.
evens = [0, 2 .. 8] is equivalent.
AtoC = [‘A’, ‘B’, ‘C’] printed as “ABC”
replicate 3 ‘A’ “AAA”

List Generator

A list comprehension can be defined using a generator:
moreevens = [2*x | x <- [0..10]]
The condition that follows the vertical bar says, “all integers x from 0 to 10.”
The symbol <- suggests set membership (∈).

Infinite Lists

Generators may include additional conditions, as in:
factors n = [f | f <- [1..n], n `mod` f == 0]
This means “all integers from 1 to n that divide f evenly.”
List comprehensions can also be infinite. E.g.:
mostevens = [2*x | x <- [0..10]]
mostevens = [0,2..]
List Transforming Functions

Suppose we define \( \text{evens} = [0, 2, 4, 6, 8] \). Then:
- head \( \text{evens} \) -- gives 0
- tail \( \text{evens} \) -- gives \([2, 4, 6, 8]\)
- head (tail \( \text{evens} \)) -- gives 2
- tail (tail \( \text{evens} \)) -- gives \([4, 6, 8]\)
- tail \([6,8]\) -- gives \([8]\)
- tail \([8]\) -- gives \([],\]

The operator \( +\) concatenates a new element onto the head of a list. E.g.,
- \(4: [6, 8]\) gives the list \([4, 6, 8]\).
- \([6, 8]:4\) -- illegal

The operator \( ++\) concatenates two lists. E.g.,
- \([2, 4]++[6, 8]\) gives the list \([2, 4, 6, 8]\).
- \(4++[6, 8]\) -- illegal
- \([4]++[6, 8]\) -- ???

List Transforming Functions

Here are some more functions on lists:
- null [] -- gives True
- null \( \text{evens} \) -- gives False
- \([1,2]==[1,2]\) -- gives True
- \([1,2]==[2,1]\) -- gives False
- \(5==[5]\) -- gives an error (mismatched args)
- type \( \text{evens} \) -- gives \([\text{Int}]\) (a list of integers)
- all even \([1,2]\) -- False
- any odd \([1,2]\) -- True
- words "You are welcome" = \["You", "are", "welcome"]

14.3.4 Elementary Types and Values

Numbers
- integers types \text{Int} (finite; like int in C, Java) and \text{Integer} (infinitely many)
- floats type \text{Float}

Numerical Functions
- \text{abs, acos, atan, ceiling, floor, cos, sin log, logBase, pi, sqrt}

Booleans
- type \text{Bool}; values \text{True} and \text{False}

Characters
- type \text{Char}; e.g., ‘a’, ‘?’

Strings
- type \text{String} = \text{[Char]}; e.g., "hello"

14.3.5 Control Flow

Conditional
- if \(x\geq y \&\& x\geq z\) then \(x\)
  - else if \(y\geq x \&\& y\geq z\) then \(y\)
  - else \(z\)

Guarded command (used widely in defining functions)
- \(|x\geq y \&\& x\geq z| = x\)
- \(|y\geq x \&\& y\geq z| = y\)
- \(|\text{otherwise} = z\)

14.3.6 Defining Functions

A Haskell Function is defined by writing:
- its prototype (name, domain, and range) on the first line, and
- its parameters and body (meaning) on the remaining lines.

\[
\text{max3} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{max3} x y z \\
| x\geq y \&\& x\geq z | = x \\
| y\geq x \&\& y\geq z | = y \\
| \text{otherwise} = z \\
\]

Note: if the prototype is omitted, Haskell interpreter will infer it.
Iterative Factorial

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

Using Pattern Matching

\[
\text{mysun } [ ] = 0 \\
\text{mysun } (x:xs) = x + \text{mysum } xs
\]

Functions are polymorphic

Omitting the prototype gives the function its broadest possible meaning. E.g.,

\[
\text{max} \begin{cases} 
  x \geq y \& \& x \geq z & \Rightarrow x \\
  y \geq x \& \& y \geq z & \Rightarrow y \\
  \text{otherwise} & \Rightarrow z
\end{cases}
\]

is now well-defined for any argument types:

\[
> \text{max3 6 4 1} \\
6
> \text{max3 "alpha" "beta" "gamma"} \\
"gamma"
\]

The member Function

\[
\text{member} :: \text{Eq } a \Rightarrow [a] \to a \to \text{Bool} \\
\text{member } \text{alist } \text{elt} \\
\begin{cases} 
  \text{alist} == [ ] & \Rightarrow \text{False} \\
  \text{elt} == \text{head } \text{alist} & \Rightarrow \text{True} \\
  \text{otherwise} & \Rightarrow \text{member } (\text{tail } \text{alist}) \text{ elt}
\end{cases}
\]

Pattern Matching

\[
\text{member } [ ] \text{ elt } = \text{False} \\
\text{member } (x:xs) \text{ elt } = \text{elt} == x \hspace{1em} \text{member } xs \text{ elt}
\]

Re: the latter can also be written:

\[
\text{member (elt:xs) elt } = \text{True} \\
\text{member (x:xs) elt } = \text{member } xs \text{ elt}
\]

\[
\text{member (x:xs) elt } = \text{if elt == } x \text{ then True} \\
\hspace{1em} \text{else member } xs \text{ elt}
\]

Functions

\[
\text{flip } f \ x \ y = f \ y \ x \\
\text{member } xs \ x = \text{elem } x \ xs \\
\text{member } = \text{elem } . \text{flip} -- \text{composition}
\]
maphead

maphead :: (a -> b) -> [ a ] -> [ b ]
maphead  func xs = [ func x | x <- xs ]

square x = x * x
maphead square [2, 3, 5, 7] = [4, 9, 25, 49]
maphead (\x -> x * x) [2, 3, 5, 7] = [4, 9, 25, 49]

map (*2) [3,6,9] = [6,12,19]

:: (a > b) -> a > [a]
++ [a] -> a->[a]         Concatenate
!! [a] -> Int -> a
   xs!!i is xs[i] in C/Java
length [a] -> Int
head [a] -> a
first element
tail [a] -> [a]
drop first element
last [a] -> a
last element of list
init [a] -> [a]
list without last element
take Int -> [a] -> [a]
take n elements from front
drop Int -> [a] -> [a]
drop n elements from front
reverse [a] -> [a]
list in reverse order
elem a -> [a] -> Bool
is element in list

14.3.7 Tuples

A tuple is a collection of values of different types. Its values are surrounded by parens and separated by commas. E.g.,
("Bob", "2771234") is a tuple.

Tuple types can be defined by the types of their values. E.g.,
type Entry = (Person, Number)
type Person = String
    type Number = String
And lists of tuples be defined as well:
type Phonebook = [(Person, Number)]

Functions on Tuples

Standard functions on tuples (first and second members):

    fst ("Bob", "2771234") returns "Bob"
    snd ("Bob", "2771234") returns "2771234"

We can also define new functions like find to search a list of tuples:

    find :: Phonebook -> Person -> [ Number ]
    find pb p = [n | (person, n) <- pb, person == p]
Program setup

list = [0, 5, 10]

main = do
    print list
    print (last list)