

Lecture 13: Introduction to Functional Programming

COMP 524 Programming Language Concepts

Aaron Block

March 1, 2007

Based on notes by N. Fisher, F. Hernandez-Campos, and D. Stotts

The University of North Carolina at Chapel Hill



Goals

- Discuss functional languages



Functional Features

- Most functional languages Provide
 - Functions as first-class values
 - Higher-order functions
 - List Type (operators on lists)
 - Recursion
 - Structured function return
 - Garbage collection
 - Polymorphism and type inference



No hard dividing line

- You can write in a “functional” style in imperative languages.
- Most functional languages have imperative subsets:

```
(define iter-fib (lambda (n)
  (do ((i 0 (+ i 1)) ; init 0
      (a 0 b) ; init 0, set to b in each iter
      (b 1 (+ a b))) ; init 1, set to sub of a & b
      ((= i n) b) ; termination test and final value
      (display b)
      (display " "))))

(iter-fib 10)
```



So Why Functional?

- Teaches Truly recursive algorithms
- Easy Polymorphism
 - How do you write in Java a method that will operate equally properly on reals as well as strings?
- Natural expressiveness for symbolic and algebraic computations
 - Algorithms vs system Munging



History

- Lambda-calculus as semantic model (Church)
- LISP (1958, MIT, McCarthy)

```
(defun fib (n)
  (if (or (= n 0) (= n 1))
      1
      (+ (fib (- n 1))
          (fib (- n 2))))))
```



History

- Lisp
 - Dynamic Scoping
- Common Lisp (CL), Scheme
 - Static scoping
- ML, Haskell, Miranda
 - Typing, type inference, pure functional
- FP (Backus): Sort of functional APL



LISP Properties

- Homogeneity of programs and data
 - Programs are lists and a program can examine/change itself while running
- Self-Definition
 - Easy to write a Lisp interpreter in Lisp



Can Programming be Liberated from the von Neumann Style?

- This is the title of a lecture given by John Backus when he received the Turing Award in 1977.
- In this, he pointed out that the program should be abstract description of algorithm rather than sequences of changes in the state of the memory.
 - He called for raising the level of abstraction
 - A way to realize this goal is functional programming
- Programs written in modern functional programming languages are a set of mathematical relationships between objects
 - No explicit memory management takes place



An Overview of Scheme

- Book uses Scheme;
 - We'll study this and ML
- Scheme is a particularly elegant Lisp
 - Interpreter runs a read-eval-print loop
 - Things typed into the interpreter are evaluated (recursively) once
 - Anything in parentheses is a function call (unless quoted)
 - Parentheses are NOT just grouping, as they are in Algo-family languages
 - Adding a level of parentheses changes meaning.



An Overview of Scheme

- Scheme:
 - Boolean values `#t` and `#f`
 - Numbers
 - Lambda Expressions
 - Quoting

```
(+ 3 4) => 7  
(quote (+ 3 4)) => (+ 3 4)  
'(+ 3 4) => (+ 3 4)
```

-



Functions

- The lambda call defines functions

```
(lambda (x) (* x x))
```

- Evaluates within scope

```
(lambda (x) (* x x) 3) => 9
```



Conditional

- Scheme:

- Conditional expressions

```
(if (< 2 3) 4 5) => 4  
(cond  
  ((< 3 2) 1)  
  ((< 4 3) 2)  
  (else 3)) => 3
```

- Imperative stuff

- Assignments
- sequencing (begin)
- iteration
- I/O (read, display)



Binding

- You can bind values to names

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +))
  (sqrt (plus (square a) (square b)))) =>5
```



Binding

- Scoping

```
(let ((a 3)
      (let ((a 4)
            (b a)) ;uses the outer value of "a" (a=3)
      (+ a b))) => 7 ;uses the inner value of "a" (a=4)
```



Binding

- Scoping (“all at once”)

```
(letrec ((fact
  (lambda (n)
    (if (= n 1) 1
        (* n (fact (-n 1)))))))
  ;uses the “inner” fact
  (fact 5))
```



Lists and Numbers

- The functions `car`, `cdr`, and `cons` allow us to manipulate lists

```
(car '(2 3 4)) => 2 ;Returns the head  
(cdr '(2 3 4)) => (3 4) ;Returns the tail  
(cdr 2 '(3 4)) => (2 3 4) ;Amend the List
```



Overview

- Scheme standard functions

```
;arithmetic  
;boolean operators  
;equivalence  
;list operators  
(symbol? x)  
(number? x)  
(complex? x)  
(real? x)  
(rational? x)  
(integer? x)
```



Overview of Scheme

- We'll invoke the program by calling a function called 'simulate', passing it a DFA description and an input string
- The automaton description is a list of three items:
 - Start state
 - Transition function
 - set of final states
- The transition function is a list of pairs
 - The first element of each pair is a pair, whose first element is a state and whose second element is an input symbol
 - If the current state and next input symbol match the first element of a pair, then the finite automaton enters the state given by the second element of the pair



Overview of Scheme

```
(define simulate
  (lambda (dfa input)
    (cons (car dfa)
          (if (nul? input)
              (if (infinal? dfa) '(accept) '(reject))
              (simulate (move dfa (car input)) (cdr input))))))
```

```
(define move
  (lambda (dfa symbol)
    (let ((curstate (car dfa)) (trans (cadr dfa)) (finals (caddr dfa)))
      (list
        (if (eq? curstate 'error)
            'error
            (let ((pair (assoc (list curstate symbol) trans)))
              (if pair (cadr pair) 'error)))
        trans
        finals))))
```



Evaluation Order

- Functional programs are evaluated following a reduction (or evaluation or simplification) process
- There are two common ways of reducing expressions
 - Application order
 - Impatient evaluation
 - Normal order
 - Lazy evaluation



Applicative Order

- In applicative order, expressions are evaluated following the parsing tree (deeper expressions are evaluated first)

```
square (3 + 4)
= { definition of + }
  square 7
= { definition of square }
  7 * 7
= { definition of * }
  49
```



Normal Order

- In Normal order, expressions are evaluated only as their value is needed

```
square (3 + 4)
= { definition of square }
  (3 + 4) * (3 + 4)
= { definition of + applied to first term }
  7 * (3 + 4)
= { definition of + applied to second term }
  7 * 7
= { definition of * }
  49
```



Evaluation Order and Infinity

- Normal is sometimes more efficient than applicative order (Why?)
- Normal order can handle expressions that never converge to normal forms

```
fun looper x = x*looper(x):int;
fun doit (flag, arg, func) =
  if flag
  then func(arg): int
  else 1;
fun do2(flag, arg) = if flag then arg else 1;
doit(true, 5, looper);
doit(false, 5, looper);
do2(false, looper(5))
```



Haskell Evaluation Order

- Haskell is a lazy functional programming language
 - Expressions are evaluated in normal order
 - Identical expressions are evaluated only once

```
square (3 + 4)
= { definition of square }
  (3 + 4) * (3 + 4)
= { definition of + applied both terms }
  7 * 7
= { definition of * }
  49
```

