Chapter 10 :: Functional Languages

Programming Language Pragmatics

Michael L. Scott

Historical Origins

- The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
  - different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics
- These results led Church to conjecture that *any* intuitively appealing model of computing would be equally powerful as well
  - this conjecture is known as *Church's thesis*

Historical Origins

- Turing's model of computing was the *Turing machine* a sort of pushdown automaton using an unbounded storage "tape"
  - the Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables

Historical Origins

- Church's model of computing is called the *lambda calculus*
  - based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter λ—hence the notation's name.
  - Lambda calculus was the inspiration for functional programming
  - one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

Historical Origins

- Mathematicians established a distinction between
  - *constructive* proof (one that shows how to obtain a mathematical object with some desired property)
  - *nonconstructive* proof (one that merely shows that such an object must exist, e.g., by contradiction)
- Logic programming is tied to the notion of constructive proofs, but at a more abstract level:
  - the logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs

Functional Programming Concepts

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language
- The key idea: do everything by composing functions
  - no mutable state
  - no side effects

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - serious polymorphism
  - powerful list facilities
  - structured function returns
  - fully general aggregates
  - garbage collection

Functional Programming Concepts

- So how do you get anything done in a functional language?
  - Recursion (especially tail recursion) takes the place of iteration
  - In general, you can get the effect of a series of assignments

```
x := 0       ...
x := expr1   ...
x := expr2   ...
```

    from `f3(f2(f1(0))),` where each f expects the value of x as an argument, f1 returns expr1, and f2 returns expr2

Functional Programming Concepts

- Recursion even does a nifty job of replacing looping

```
x := 0; i := 1; j := 100;
while i < j do
   x := x + i*j; i := i + 1;
        j := j - 1
end while
return x
```

becomes `f(0,1,100),` where

```
f(x,i,j) == if i < j then
f (x+i*j, i+1, j-1) else x
```

Functional Programming Concepts

- Thinking about recursion as a direct, mechanical replacement for iteration, however, is the wrong way to look at things
  - One has to get used to thinking in a recursive style
- Even more important than recursion is the notion of ***higher-order functions***
  - Take a function as argument, or return a function as a result
  - Great for building things

Functional Programming Concepts

- Lisp also has (these are not necessary present in other functional languages)
  - homo-iconography
  - self-definition
  - read-evaluate-print
- Variants of LISP
  - Pure (original) Lisp
  - Interlisp, MacLisp, Emacs Lisp
  - Common Lisp
  - Scheme

Functional Programming Concepts

- Pure Lisp is purely functional; all other Lisps have imperative features
- All early Lisps dynamically scoped
  - Not clear whether this was deliberate or if it happened by accident
- Scheme and Common Lisp statically scoped
  - Common Lisp provides dynamic scope as an option for explicitly-declared *special* functions
  - Common Lisp now THE standard Lisp
    - Very big; complicated (The Ada of functional programming)

Functional Programming Concepts

- Scheme is a particularly elegant Lisp
- Other functional languages
  - ML
  - Miranda
  - Haskell
  - FP
- Haskell is the leading language for research in functional programming

A Review/Overview of Scheme

- As mentioned, 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 Algol-family languages
    - Adding a level of parentheses changes meaning

A Review/Overview of Scheme

- As mentioned earlier, 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 Algol-family languages
    - Adding a level of parentheses changes meaning

```
(+ 3 4) ⇒ 7
((+ 3 4))) ⇒ error
```

(the ' ⇒' arrow means 'evaluates to‘)

A Review/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)
```

  - Mechanisms for creating new scopes

```
(let ((square (lambda (x) (* x x))) (plus +))
(sqrt (plus (square a) (square b))))
let*
letrec
```

A Review/Overview of Scheme

- 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)

A Review/Overview of Scheme

- Scheme standard functions (this is not a complete list):
  - arithmetic
  - boolean operators
  - equivalence
  - list operators
  - symbol?
  - number?
  - complex?
  - real?
  - rational?
  - integer?

A Review/Overview of Scheme

Example program - Simulation of DFA

- 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
    - the transition function
    - the 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 in 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

A Review/Overview of Scheme
Example program - Simulation of DFA

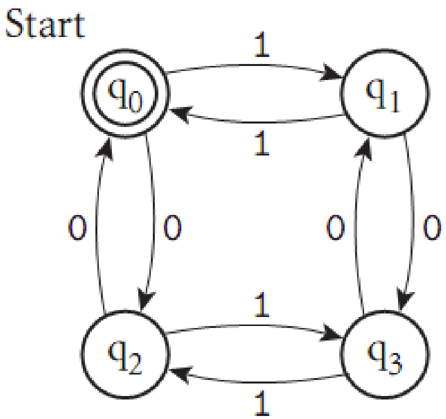
```
(define simulate
  (lambda (dfa input)
    (cons (current-state dfa)              ; start state
          (if (null? input)
              (if (infinal? dfa) '(accept) '(reject))
            (simulate (move dfa (car input)) (cdr input))))))

;; access functions for machine description:
(define current-state car)
(define transition-function cadr)
(define final-states caddr)
(define infinal?
  (lambda (dfa)
    (memq (current-state dfa) (final-states dfa))))

(define move
  (lambda (dfa symbol)
    (let ((cs (current-state dfa)) (trans (transition-function dfa)))
      (list
       (if (eq? cs 'error)
           'error
         (let ((pair (assoc (list cs symbol) trans)))
           (if pair (cadr pair) 'error)))   ; new start state
       trans                                ; same transition function
       (final-states dfa)))))               ; same final states
```

Figure 10.1 Scheme program to simulate the actions of a DFA. Given a machine description and an input symbol *i*, function `move` searches for a transition labeled *i* from the start state to some new state *s*. It then returns a new machine with the same transition function and final states, but with *s* as its "start" state. The main function, `simulate`, tests to see if it is in a final state. If not, it passes the current machine description and the first symbol of input to `move`, and then calls itself recursively on the new machine and the remainder of the input. The functions `cadr` and `caddr` are defined as `(lambda (x) (car (cdr x)))` and `(lambda (x) (car (cdr (cdr x))))`, respectively. Scheme provides a large collection of such abbreviations.

A Review/Overview of Scheme
Example program - Simulation of DFA




```
(define zero-one-even-dfa
 '(q0                                                    ; start state
   (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)     ; transition fn
    ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
   (q0)))                                                ; final states
```

Figure 10.2 DFA to accept all strings of zeros and ones containing an even number of each. At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 10.1.

Evaluation Order Revisited

- Applicative order
  - what you're used to in imperative languages
  - usually faster
- Normal order
  - like call-by-name: don't evaluate arg until you need it
  - sometimes faster
  - terminates if anything will (Church-Rosser theorem)

Evaluation Order Revisited

- In Scheme
  - functions use applicative order defined with lambda
  - special forms (aka macros) use normal order defined with syntax-rules
- A *strict* language requires all arguments to be well-defined, so applicative order can be used
- A n*on-strict* language does not require all arguments to be well-defined; it requires normal-order evaluation

Evaluation Order Revisited

- Lazy evaluation gives the best of both worlds
- But not good in the presence of side effects.
  - delay and force in Scheme
  - delay creates a "promise"

High-Order Functions

- Higher-order functions
  - Take a function as argument, or return a function as a result
  - Great for building things
  - Currying (after Haskell Curry, the same guy Haskell is named after)
    - For details see Lambda calculus on CD
    - ML, Miranda, and Haskell have especially nice syntax for curried functions

Functional Programming in Perspective

- Advantages of functional languages
  - lack of side effects makes programs easier to understand
  - lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
  - lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
  - programs are often surprisingly short
  - language can be extremely small and yet powerful

Functional Programming in Perspective

- Problems
  - difficult (but not impossible!) to implement efficiently on von Neumann machines
    - lots of copying of data through parameters
    - (apparent) need to create a whole new array in order to change one element
    - heavy use of pointers (space/time and locality problem)
    - frequent procedure calls
    - heavy space use for recursion
    - requires garbage collection
    - requires a different mode of thinking by the programmer
    - difficult to integrate I/O into purely functional model