Functional Programming

CS 1025 Computer Science Fundamentals I

Stephen M. Watt

University of Western Ontario

When the Function is the Thing

- In O-O programming, you typically know *where* an action is needed, but *what* is to be done depends on the particulars.
- In *functional* programming, you typically know *what* action is needed, but *where* it is to be done depends on the particulars.
- Some programming languages make passing functions around and combining them easy.
- These are known as *functional programming languages.*

Functional Programming

- Some believe the need to use concurrency for future hardware speed-up as "the end of the free lunch" and see FP as the solution.

- Advocates say

  "If you aren't programming functionally,
  you are programming dysfunctionally"

- FP here stands for "functional programming", but is also the name of a particular functional programming language by John Backus, of Fortran fame.

Related Concepts

- All functional programming languages allow you to pass functions as parameters, return them as results, construct new functions by composing others, etc.
- Some do *not* allow variable update or structure modification.
- Some have *lazy evaluation.*
- When you do this some things get easier and some things get harder.

A Language with a Functional Subset

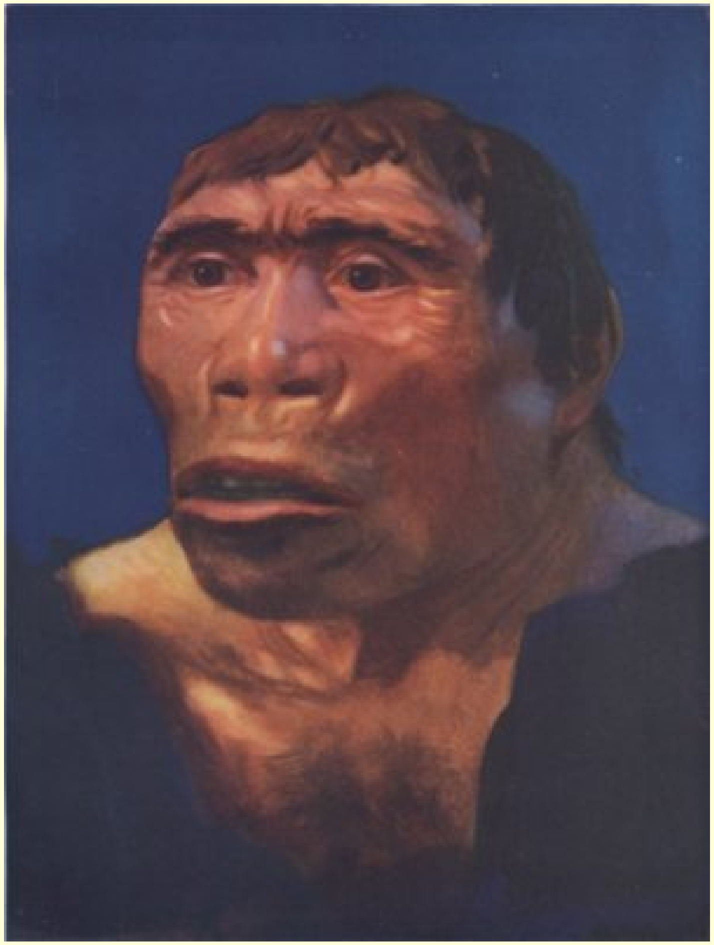
- Scheme is a multi-paradigm programming language in the Lisp family with a nice functional subset.

- Developed in the 1970s by Guy Steele and his PhD Supervisor Gerald Sussman at MIT.

- Used as a first language of instruction at MIT in the *pre-Java* era.

- Used as a first language of instruction at Waterloo in the *post-Java* era.

Java Man



Pithecanthropus erectus

from “The Outline of Science” J. Arthur Thomson (1922)

Elements of Scheme

- Syntax: (operator arg ...)
- Some operators are built-in, others programmer defined.

- lambda: create a function `(lambda (n) (+ n 1))`

- if: conditional evaluation `(if (> n 0) n (- n))`

- define: introduce a name (valid at top level and certain other places)

```
(define n 7)

(define factorial (lambda (n)
      (if (= n 1) 1 (* n (factorial (- n 1)))) )
```

List Operations

- (cons *a b*) create a “pair” data structure

- (car *p*) first element of a pair
- (cdr *p*) second element of a pair

- (null? *x*) test whether x is a null pointer.
- ’( ) special syntax for the null pointer.

- (list *a1 ...*) short-hand for some cons-es ending with null.

```
(cons 1 (cons 2 (cons 3 (cons 4 ’( ) )))) ⇔ (list 1 2 3 4)
```

Recursive Structures

- With recursive list data structures, it is natural to write recursive programs.

- Make a new list by adding 3 to each element of an input list:

```
(define add-3-to-each (lambda (l)
        (if (null? l)
             ’()
            (cons (+ 3 (car l))
                  (add-3-to-each (cdr l)) ) ) ))
```

- Make a new list by squaring each element of an input list:

```
(define square-each (lambda (l)
       (if (null? l)
            ’()
           (cons (* (car l) (car l))
                 (square-each (cdr l)) ) ) ))
```

Functions Can Be Arguments

```
(define call-my-function (lambda (f a)  (f a)))


(define call-fun-on-list-elements (lambda (f l)
       (if (null? l)
           ’()
           (cons (f (car l))
                 (call-fun-on-list-elements f (cdr l)) ) )


(define zipper (lambda (f l1 l2)
         (if (or (null? l1) (null? l2))
             ’()
             (cons (f (car l1) (car l2))
                   (zipper f (cdr l1) (cdr l2)) ) ) ))
```

Local Bindings

- Local variables may be introduced with “let”
- It has the form

```
(let ( (var1 initial-value1) (var2 initial-value2) ...)
     expr1
     expr2 ...)
```

- E.g.

```
(define factorial (lambda (n)
    (let ((nm1 (- n 1)))

        (if (< n 2)
            1
            (* n (factorial nm1)) ) ) ))
```

Lexical Scoping

- An inner function use all the local names of the functions that enclose it.

```
(define outer-fn (lambda (n)
   (let ( (inner-fn  (lambda (m) (+ m n)) ) )

       (inner-fn (+ n 2) ) ))
```

Returning Functions: Closures

- E.g.

```
(define add (lambda (a)
    (lambda (b) (+ a b))  ))
```

- What is the value of “a” when the inner function is returned?

- It is the value of “a” that “add” was called with.

  E.g. `(add 3)  =>  (lambda (b) (+ a b))  ; with a = 3`

Returning Functions: Closures

- E.g. A counter...

```
(define make-counter (lambda ()
    (let ((count 0))
       (lambda (n)
          (set! count (+ count n))
          count ) ) ))

  (define counter1 (make-counter))
  (counter1 7)  ; yields 7
  (counter1 8)  ; yields 15

  (define counter2 (make-counter))
  (counter2 9)  ; yields 9
  (counter2 3)  ; yields 12
  (counter1 3)  ; yields 18
```

Functional Programming Tricks

- Functional composition

```
(define compose (lambda (f g) (lambda (a) (f (g a))) ))
```

- E.g.

```
(define negative-inverse (compose - /))

(negative-inverse 9)   ;  Yields – 1/9
```

Functional Programming Tricks

- Convert make a unary function from a binary function:

```
(define curry (lambda (f) (lambda (a) (lambda (b) (f a b)

(define plus  (curry +))
(define plus5 (plus 5))
(define nine  (plus5 4))

((plus 5) 4)    ; Yields 9
```

Functional Programming Tricks

- Changing the order of arguments:

```
(define twist (lambda (f) (lambda (a b) (f b a)) ))


(define subtract-from (twist -))

(subtract-from 9 11)   ; Yields 2

(define minus1 ((curry subtract-from) 1))

(minus1 9)             ; Yields 8
```

Composing Functional Elements

- Very powerful
- Complex ideas can be expressed with short programs

- Be careful not to write unreadable code.

Functional Programming with Lists

- map: `(map f (list a b c d))`

  gives `(list (f a) (f b) (f c) (f d))`

  This is built in in Scheme.

Functional Programming with Lists

- reduce: `(reduce f (list a b c d))`
  gives `(f a (f b (f c d)))`

  E.g. `(reduce + (list 1 2 3 4 5))` ; Yields 15

```
(define dot-product (lambda (u v)
    (reduce + (zipper * u v)) ))


(define eval-line (lambda (x) (lambda (b a) (+ b (* a x)) )))

(define eval-poly (lambda (x) (lambda (l) (reduce (eval-line x) l))))

((eval-poly 2) (list 5 4 3 2 1))   ; Yields 57
```

Functional Programming with Lists

- Spread: `(spread (list f g h) x)`
  gives `(list (f x) (g x) (h x))`

- Question: Write the “spread” function using list operations.

- Question: Write the “spread” function using “map” and “lambda.”

Lazy Evaluation: Force and Delay

- "delay" creates a *promise* ... An object that may be evaluated later.
- "force" causes the promise to be evaluated to give a value.

- E.g.

```
(define make-five (lambda () (write "Hello") (+ 2 3)))

(define five (delay (make-five))) ; make-five not called
...
...
(define fiveno (force five))      ; make-five called here
```