Chapter 10: Functional Languages

July 9, 2015

Origins

- Alonzo Church created lambda calculus, which is where much of the original thoughts that governed functional languages originated (though Backus specifically excluded lambda calculus from his formulation of functional language theory!)

Characteristics and Concepts

- Pure functionality: a program is a mathematical function over the input. No state and no side effects.
- Haskell is purely functional, but few other functional languages are. Most others have imperative characteristics (particularly with regard to state).

Other common characteristics and concepts

- First-class function values and higher-order functions. (Think "map()", for instance.)
- Extensive polymorphism
- Native list types and operators
- Constructors for structured objects
- Garbage collection

Scheme

- Quoting convention '(something) — or just use the quote function
- True is #t, false is #f
- Functions are created by evaluating lambda expressions

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

Let it be

- let, letrec bindings are only local to a nested scope

```
gosh> (let ( (a 3)
             (b 4)
             (square (lambda(x) (* x x)))
             (plus +))
        (sqrt (plus (square a) (square b))))
5
gosh> b
*** ERROR: unbound variable: b
Stack Trace:
_______________________________________
```

The car and cdr

```
gosh> (car '(1 2 3 4 5))
1
gosh> (cdr '(1 2 3 4 5))
(2 3 4 5)
gosh> (cons 1 '(2 3 4 5))
(1 2 3 4 5)
```

Other useful bits

```
gosh> (null? () )
#t
gosh> (null? '(2))
#f
```

Other useful bits

```
gosh> (memq 'a '(a b c d))
(a b c d)
gosh> (memq 'b '(a b c d))
(b c d)
gosh> (memq 'c '(a b c d))
(c d)
```

Other useful bits

```
gosh> (member '(a) '((a) b c d))
((a) b c d)
gosh> (member '(b) '((a) b c d))
#f
gosh> (member '(b) '((a) (b) c d))
((b) c d)
```

Conditionals

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

Assignment, sequencing, and iteration

```
gosh> (define iter-fib (lambda (n)
        (do ((i 0 (+ i 1))
             (a 0 b)
             (b 1 (+ a b)))
          ((= i n) b)
          (display b)
          (display " "))))
gosh> (iter-fib 10)
1 1 2 3 5 8 13 21 34 55 89
```

Programs as lists

- Lisp is self-representing; a program is a list, and lists are what programs manipulate.
- quote and eval: the opposite of a quote ‘(something) is an eval’(something)

```
gosh> (eval '(+ 3 4) (interaction-environment))
7
gosh> (eval (quote (+ 3 4)) (interaction-environment))
7
```

Programs as lists

- apply (versus map)

```
gosh> (+ 3 4 5)
12
gosh> (apply + '(3 4 5))
12
gosh> (map + '(3 4 5))
(3 4 5)
gosh> (apply + '(3 4 5) '(8 9 10))
*** ERROR: operation + is not defined between (3 4 5) and 8
gosh> (map + '(3 4 5) '(8 9 10))
(11 13 15)
```

Using just eval and apply, you can write a Scheme interpreter in Scheme

- Here's the code from SICP

Evaluation order

- "Applicative-order" means evaluate your argments before you pass them to a function (typical stack machine layout)
- "Normal-order" means pass unevaluated arguments to a function

Evaluation order

- In Scheme, ordinary function calls are done in applicative-order, and special forms (such as cond, for instance) are done normal-order (which makes a lot of sense, since cond only uses the first condition that evaluates to true.)

Strictness and Laziness

- A language is strict iff if all function calls are strict; a function call is strict iff it is undefined when any of its arguments is undefined. Scheme is strict; Haskell is nonstrict.
- So, how do you achieve this nonstrict behavior? Laziness! You get normal-order evaluation, and acceptable speed, also. The trick is via "memoization", which tags an argument with a value when it becomes known (viz., dynamic programming.)

Strictness and Laziness

- Laziness is really useful also for infinite data structures, such as the Haskell infinite list [1..]
- The fly in the ointment, of course, are side-effects; however, the usual solution to that dilemma is to just forbid side-effects (hey, what about I/O? Isn't that just a bunch of side effects!?)

Streams and monads

- How about viewing I/O as a *stream*, an unbounded list of whose elements are generated *lazily*.
- But, while useful, streams are not quite enough.

Monads

- Instead, we want to have hidden state (the real world is a stateful (and entropic) place.)

Higher-order functions

- Consider the following Haskell:

```
Prelude> let plus a b = a + b
Prelude> let x = plus 3
Prelude> x 7
10
Prelude> x 100
103
Prelude> let y = plus 21
Prelude> y 200
221
Prelude> y 300
321
```

Currying...

- And now consider this application of our *plus* function:

```
Prelude> foldl plus 0 [1..10]
55
```