$(\lambda\ (\lambda)\ ((\lambda\ \lambda)))$

Functional programming

Contents

A session with Scheme

```
% scm

> '( ( person Jack  ( married Jill ) )
     ( person Jim   ( single ) )
     ( person Jerry ( alimony 800 ) )
   )
((person jack (married jill)) (person jim
(single)) (person jerry (alimony 800)))

> ( cons 'alpha '( beta ) )
(alpha beta)

> ( symbol? 'alpha )
#t

> ( symbol? '( alpha ) )
#f

> ( symbol? alpha )
ERROR: unbound variable:  alpha
; in expression: (... alpha)
; in top level environment.

> ( null? 'alpha )
#f

> ( null? () )
#t

> ( cdr ( cons 'x '( y z ) ) )
(y z)

> ( cons 'x ( cdr '( y z ) ) )
(x z)
```



```
> ( define ( addOne x )
    ( + x 1 )
  )
#<unspecified>

> ( addOne 10 )
11

> ( addOne ( addOne 15 ) )
17

> ( define ( conj x y )
    ( if x y #f )
  )
#<unspecified>

> ( conj ( symbol? '(a) )  ( eq? 'a 'a ) )
#f

> ( define ( disj x y )
    ( if x #t y )
  )
#<unspecified>

> ( disj ( symbol? '(a) )  ( eq? 'a 'a ) )
#t


> ( eq? 'a 'a )
#t
> ( eq? 'a 'b )
#f
> ( eq? '( a ) '( a ) )
#f
```

```
>
( define ( eqExpr? x y )
  ( if  ( symbol? x )
        ( if  ( symbol? y )
              ( eq? x y )
              #f
        )
        ( if  ( null? x )
              ( null? y )
              ( if  ( eqExpr? ( car x )
                              ( car y ) )
                    ( eqExpr? ( cdr x )
                              ( cdr y ) )
                    #f
)  )    )     )
#<unspecified>

> ( eqExpr?  '( a b ( c d ) )
             '( a b ( c d ) )  )
#t

> ( eqExpr?  '( a b ( c d ) )
             '( a b ( c d e) )  )
#f

>
( define ( eqExpr? x y )
           ; the same as built-in "equal?"
  ( cond
    ( ( symbol? x )  ( eq? x y ) )
    ( ( null? x )  ( null? y ) )
    ( ( eqExpr? ( car x ) ( car y ) )
      ( eqExpr? ( cdr x ) ( cdr y ) ) )
    ( else  #f )
) )
#<unspecified>
```



```
> ( eqExpr?  '( a b ( c d ) )
             '( a b ( c d ) )  )
#t
> ( eqExpr?  '( a b ( c d ) )
             '( a b ( c d e) )  )
#f

> ( define ( member? K L )
  ( cond
    ( (null? L )  #f )
    ( ( eqExpr? K ( car L ) )  #t )
    ( else  ( member? K ( cdr L ) ) )
) )
#<unspecified>

> ( member? 'aa '( bb cc aa ee rr tt ) )

#t

> ( member? 'aa '( bb cc (aa) ee rr tt ) )

#f

> ( define ( append L1 L2 )    ; built-in!
    (if  ( null? L1 )
         L2
         ( cons ( car L1 )
                ( append ( cdr L1 ) L2 ) )
  ) )
WARNING: redefining built-in append
#<unspecified>

> ( append '( ab bc cd )
           '( de ef fg gh ) )
(ab bc cd de ef fg gh)

> ( exit )
;EXIT
```

Lisp in general

There were many dialects, starting with Lisp 1.5 (1960), through Scheme (1975) to Common Lisp (1985).

[Other important functional programming languages are Hope, ML, Miranda.]

The mathematical basis of many functional programming languages is λ-calculus (it allows expressions that have functions as values).

Fundamental control mechanisms:

- function application,
- function composition,
- conditional schema,
- recursion.

Data structures are very simple:

- lists,
- atoms.

Programs and data are expressed in the same syntax:

- function applications and conditional schemata are written as lists, in a parenthesized prefix form;
- program and data are distinguished by context.

This uniformity of data and programs gives functional programming languages their flexibility and expressive power:

programs can be manipulated as data.

A one-page interpreter of Lisp in Lisp was the basis of a first ever bootstrapping implementation of a programming language (a very powerful technique).

- Pure Lisp has only five primitive functions:

`cons`	—	build a list,
`car`	—	head of a list,
`cdr`	—	tail of a list,
`eq`	—	equality of atoms (Boolean),
`atom`	—	is this an atom (Boolean);

- There are only <u>two</u> other essential operations:
  - evaluate an expression,
  - apply a function to (evaluated) arguments

  (plus several auxiliary operations to help handle argument lists and conditional evaluation).

Lisp is used interactively (as Prolog or Smalltalk):

- there is no main program,
- the top level loop ("ear") evaluates an expression for its value or for its side-effects such as I/O (this expression may invoke a function that implements a large and complex algorithm),
- a Lisp program is a collection of functions that may be called (directly or indirectly) from the top level

Expressions are normally evaluated: you must specially ask Lisp to leave something unevaluated (quoted).

Atoms are treated literally, that is, they stand for themselves.

The name of an atom may mean something in the application domain, but that's not a concern for the programming language.

<u>Lisp 1.5 has several weaknesses:</u>

- awkward (though elegantly uniform) syntax,
- dynamic scope rule,
- inconsistent treatment of functions as arguments (because of dynamic scoping!).

Back to Scheme

Scheme is a small but well-designed subset/dialect of Lisp.

- Lexical scope rule.
- Correct treatment of functional arguments (thanks to lexical scoping):

  functions are first-class objects, that is, they can be created, assigned to variables, passed as arguments, returned as values.

Data structures in Scheme are simple, uniform and versatile. They are called S-expressions (like in Lisp).

Simple data structures

- A number: as usual (integer or float).
- A variable: a name bound to a data object, e.g.,

```
(define pi 3.14159)
```

A variable has a type implicitly, depending on its value. It can be assigned a new value:

```
(set! pi 3.141592)
(set! pi 'alpha)
```

- A symbol is a name that is used for its shape (it has no value other than itself).
  (Lisp called symbols "atoms".)

Compound data structures

- A list:

  (E_1 E_2 ...... E_n) where E_i are S-expressions.

  Depending on context, a list is treated literally (as a piece of data), e.g.,

  ```
  (William Shakespeare
       (The Tempest))
  ```

  or as a function application with arguments passed by value, e.g.

  ```
  (append x y)
  ```

- A "dotted" pair (seldom used in practice) underlies the structure of lists. A dotted pair is produced by `cons`:

  `cons(α β)` returns `(α . β)`

  A list (E_1 E_2 ...... E_n) is actually represented as

  (cons E_1 (cons E_2 ......
  (cons E_n ()) ......))

  that is, as

  (E_1 . (E_2 ... (E_n . ()) ...))

Evaluating a function

Given: a list (E0 E1 ... En)

Step (1)

Evaluate E_0 to get V_0,

Evaluate E_1 to get V_1,

......,

Evaluate E_n to get V_n.

V_0 must be a function, V_1, ..., V_n are data objects.

Step (2)

Apply V_0 to V_1, ..., V_n:

compute V0(V1, ..., Vn).

Evaluation may be suppressing by quoting

`(quote pi)` or, more conveniently,

`'pi`

Examples:

`(* 2.0 pi)`	gives `6.283184`
`(* 2.0 'pi)`	has a wrong argument
`('* 2.0 'pi)`	has a wrong function!
`(write 'pi)`	outputs the symbol `pi`
`(write pi)`	outputs `3.141592`

List construction and access to elements

A list is defined recursively:

- an empty list is `()`,
- a non-empty list is

  `(cons α ξ)`

  where ξ is a list.

The head and the tail of a list:

`(car (cons α ξ))` equals α

`(cdr (cons α ξ))` equals ξ

`(car ())` and `(cdr ())` are incorrect

There is a notational convention for accessing further elements of a list:

```
(caar x) ≡ (car (car x))
(cdadr x) ≡ (cdr (car (cdr x))))
```

For example, consider this 4-step evaluation:

```
(caadar '((p ((q r) s) u) (v)))
(caadr '(p ((q r) s) u))
(caar '(((q r) s) u))
(car '((q r) s))
'(q r)
```


Another example:
the second element of list `x`—if it exists—is
`(cadr x)`
the third, fourth, ... elements—if they exist—are
`(caddr x)`, `(cadddr x)`, etc.

`car`, `cdr`, `cons` are three (out of five) primitive functions that ensure all the necessary access to lists. Two other primitives are predicates: functions that return a special symbol `#t` or `#f`.

`(symbol? x)`
if and only if `x` is a symbol,

`(number? x)`
if and only if `x` is a number,

`(eq? x y)`
if and only if the values of `x` and `y` are symbols and are identical.

Other commonly used predicates (they can be defined using the primitive five):

`(equal? x y)` is true if the values of `x` and `y` are the same object, maybe not atomic.

`(null? x)` is true if `x` is `()`, i.e. the empty list.

Function expressions and definitions of functions

```
(define (square x)(* x x))
```

or

```
(define square
        (lambda (x)(* x x)))
```

Control in Scheme (as in Lisp) is very simple: function application, conditional schema, and—as a concession to the imperative programming habits—sequence (not discussed here).

The conditional schema:

(cond (C_1 E_1)
      (C_2 E_2) ......
      (C_n E_n)
      (else E_{n+1}))

The last part, (else E_{n+1}), is optional.
(C_i E_i) represents one condition-expression pair. Pairs are evaluated left-to-right. We stop when we find a true C_i (its value is `#t`). We return E_i as the value of the whole conditional schema.

The conditional schema, a special case:

(cond (C_1 E_1) (else E_2))

can be abbreviated as

(if C_1 E_1 E_2)

More examples of functions in Scheme:

```
(define (same_neighbours? l)
  (cond
    ((null? l) #f)
    ((null? (cdr l)) #f)
    ((equal? (car l)(cadr l)) #t)
    (else
       (same_neighbours? (cdr l)))
) )
```

Stack operations in Scheme

```
(define (empty? stack)
   (null? stack)
)
```

```
(define (push elem stack)
   (cons elem stack)
)
```

```
(define (pop stack)
   (if (empty? stack)
       stack
       (cdr stack)
)
```

```
(define (top stack)
   (if (empty? stack)
       ()
       (car stack)
)
```


Minimum of a list

```
(define (minl l)
  (if (null? l)
      l
      (minl-aux (car l)(cdr l))
) )
```

```
(define (minl-aux elt lst)
  (cond
   ((null? lst) elt)
   ((> elt (car lst))
    (minl-aux (car lst)(cdr lst)))
   (else (minl-aux elt (cdr lst)))
) )
```

A variant with local scope:

```
(define (minl-aux elt lst)
   (if (null? lst)
       elt
       (let
          ((carl (car lst))
           (cdrl (cdr lst)))
          (if
            (> elt carl)
            (minl-aux carl cdrl)
            (minl-aux elt cdrl)
)  )    )  )
```

Higher-order functions

"Higher-order" means having functions as arguments. The classic example is `map`, the operation of applying a function to a list and returning a list:

$$(E_1\ E_2\ \ldots\ E_n) \rightarrow ((f\ E_1)\ (f\ E_2)\ \ldots\ (f\ E_n))$$

```
(define (map f l)
  (if (null? l)
      l
      (cons (f (car l))
            (map f (cdr l)))
) )
```

For example, this gives `(2 3 4)`:

```
(map (lambda(x) (+ x 1)) '(1 2 3))
```

A version of map which does something for all elements, without creating a list of results:

```
(define (do-for-all f l)
  (if (null? l)
    l
    (let ((dummy (f (car l))))
         (do-for-all f (cdr l))
) ) )
```

For example:

```
(do-for-all write '(1 2 3))
```


Reducers

Let f be a binary operation, that is, a two-argument function. Let f0 be a constant. We want to express the following transformation:

$$(E_1\ E_2\ \ldots\ E_n) \rightarrow$$
$$(f\ E_1\ (f\ E_2\ (f\ \ldots\ (f\ E_n\ f0)\ \ldots)))$$

This is better written with f as an infix operator:

$$(E_1\ E_2\ \ldots\ E_n) \rightarrow E_1\ f\ E_2\ f\ \ldots\ f\ E_n\ f\ f0$$

Examples:

$$(E_1\ E_2\ \ldots\ E_n) \rightarrow E_1 + E_2 + \ldots + E_n + 0$$
$$(E_1\ E_2\ \ldots\ E_n) \rightarrow E_1 * E_2 * \ldots * E_n * 1$$

```
(define (reduce f f0 l)
  (if (null? l)
      f0
      (f (car l)
         (reduce f f0 (cdr l)))
) )
```

Examples:

`(reduce + 0 '(1 2 3 4))` gives `10`

`(reduce * 1 '(1 2 3 4))` gives `24`

What does this expression mean?

```
(reduce cons () '(1 2 3 4))
```

Scheme—summary