Functional programming

(λ(λ(λ(λ(λ(λ(λ(λ(λ(λ(λ))))))))))))
A session with Scheme

```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
```

CSI3125, Functional Programming, page 92
( 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 ) )
  ) )
)

( 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 )
  ) )
)

(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

( append '( ab bc cd ) '( de ef fg gh ) )
'(ab bc cd de ef fg gh)
Functional programming languages are very simple.

Data structures are very simple:

- atoms
- cons
- car
- cdr
- list
- tail of a list
- head of a list

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

Fundamental control mechanisms:

- recursion
- conditional schema
- function composition
- function application
- function application

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)

Programs and data are manipulated as data.

Programs and data are distinguished by context:

- form: programs are written as lists, in a parenthesized prefix syntax
- function applications and conditional schemata

Programs and data are expressed in the same syntax.

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


There were many dialects, starting with Lisp 1.5.

In general, Lisp
• There are only two other essential operations:
  • evaluate an expression,
  • apply a function to (evaluated) arguments (plus several auxiliary operations to help handle
    arguments) •

There are only two other essential operations:

evaluate an expression,
apply a function to (evaluated) arguments (plus several auxiliary operations to help handle
arguments) •

Lisp is used interactively (as Prolog or Smalltalk): Lisp has several weaknesses:

Lisp 1.5 has several weaknesses:

Lisp is used interactively (as Prolog or Smalltalk):

The top level may be called (directly or indirectly) from the
algorithm:

The top level loop ("car") 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),

There is no main program,

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

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
Lisp.

Expressions are normally evaluated; you must
especially ask Lisp to leave something unevaluated.

Expressions are normally evaluated; you must
especially ask Lisp to leave something unevaluated.
Scheme is a small but well-designed subset/dialect of Lisp.

- Lexical scope rule.
- Correct treatment of functional arguments passed as arguments, returned as values.
  - Can be created, assigned to variables, 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.,
  ```scheme```
  ```
  (define pi 3.14159)
  ```
  A variable has a type implicitly, depending on its value. It can be assigned a new value:
  ```scheme```
  ```
  (set! pi 3.141592)
  ```
  (set! pi 'alpha)
  ```
  ```scheme```
  ```
  (define pt 3.14159)
  ```
  A variable: a name bound to a data object, e.g.,
- A symbol is a name that is used for its shape (it has no value other than itself). (Lisp called symbols "atoms").

(Thanks to lexical scoping):

- Subset/dialect of Lisp.

Scheme is a small but well-designed
Compound data structures

- A list: \((E_1 \ E_2 \ \ldots \ E_n)\) where \(E_i\) are S-expressions. Depending on context, a list is treated literally (as a piece of data), e.g., \((\text{William Shakespeare} \ \text{The Tempest})\) or as a function application with arguments passed by value, e.g., \((\text{append} \ x \ y)\).

- A "dotted" pair (seldom used in practice) underlies the structure of lists. A dotted pair is produced by \texttt{cons}:
  \[
  \text{cons}(\alpha \ \beta) \quad \text{return} \quad (\alpha . \beta)
  \]
  A list \((E_1 \ E_2 \ \ldots \ E_n)\) is actually represented as
  \[
  \text{cons} \ \text{cons}(E_1 \ \text{cons}(E_2 \ \text{cons}(E_n \ \text{cons}()) \ \ldots))
  \]
  that is, as
  \[
  (\ldots (\ldots (\ldots (\alpha . \beta) \ldots) \ldots) \ldots)
  \]

Evaluating a function

Given a list \((E_0 \ E_1 \ \ldots \ E_n)\), a function evaluation proceeds as follows:

1. \text{Evaluate } E_0 \text{ to get } V_0,
2. \text{Evaluate } E_1 \text{ to get } V_1,
   \quad \ldots,
3. \text{Evaluate } E_n \text{ to get } V_n.

Step (1)

Apply \(V_0\) to \(V_1, \ldots, V_n\):
\(V_0\) must be a function, \(V_1, \ldots, V_n\) are data objects.
Evaluate \(E_0\) to get \(V_0\).
Evaluate \(E_1\) to get \(V_1\).
Evaluate \(E_n\) to get \(V_n\).

Step (2)

Given a function \(\text{quote}\) of \(\alpha\) \& \(\beta\):
Evaluate a function application with arguments
\(\text{quote}\) one by one:
Evaluate \(E_1\) to get \(V_1\).
Evaluate \(E_0\) to get \(V_0\).

Step (3)

Evaluate \((E_1 \ (E_0))\) (\text{quote}\) \(\text{quote}\) by \texttt{quote}.

Examples:

\[
\text{quote } \pi
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]

\[
(\text{quote } \pi)
\]
List construction and access to elements

A list is defined recursively:

• an empty list is ()
• a non-empty list is (\(\text{cons}\) \(\alpha\) \(\xi\)) where \(\xi\) is a list.

The head and the tail of a list:

\((\text{car} (\text{cons} \alpha \xi))\) equals \(\alpha\)
\((\text{cdr} (\text{cons} \alpha \xi))\) equals \(\xi\)
\((\text{car} ())\) and \((\text{cdr} ())\) are incorrect

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

\((\text{caar} x)\) \(\equiv\) \((\text{car} (\text{car} x))\)
\((\text{cdadr} x)\) \(\equiv\) \((\text{cdr} (\text{car} (\text{cdr} x)))\)

For example, consider this 4-step evaluation:

\((\text{caadar} '((p ((q r) s) u) (v)))\)
\((\text{caadr} '(p ((q r) s) u))\)
\((\text{caar} '(((q r) s) u))\)
\((\text{car} '((q r) s))\)
\('[q r]\)

Another example:

the second element of list \(x\) —if it exists—is \((\text{cadr} x)\)
the third, fourth, ... elements—if they exist—are \((\text{caddr} x)\), \((\text{cadddr} x)\), etc.

There are three (out of five) primitive functions that ensure all the necessary access to a list, \text{car}, \text{cdr}, \text{cons} are three (out of five) primitive functions that return a special symbol \#f or \#t.

Two other primitives are predicates:

\(\text{(symbol? x)}\) if and only if \(x\) is a symbol,
\(\text{(number? x)}\) if and only if \(x\) is a number,
\(\text{(eq? x y)}\) if and only if \(x\) and \(y\) are identical.

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

\(\text{(equal? x y)}\) is true if the values of \(x\) and \(y\) are the same object, maybe not atomic.
\(\text{(null? x)}\) is true if the values of \(x\) and \(\xi\) are defined using the primitive \(\text{eq?}\).

There is a notional convention for accessing elements of a list:

\((\text{car} x)\) \(\equiv\) \((\text{cdr} (\text{cdr} (\text{cdr} (\text{cdr} (\text{cdr} (\text{cdr} x))))))\)
\((\text{caddr} x)\) \(\equiv\) \((\text{car} (\text{cdr} (\text{cdr} (\text{cdr} (\text{cdr} (\text{cdr} (\text{cdr} x))))))))\)

For example, consider this 4-step evaluation:

\((\text{caadar} '((p ((q r) s) u) (v)))\)
\((\text{caadr} '(p ((q r) s) u))\)
\((\text{caar} '(((q r) s) u))\)
\((\text{car} '((q r) s))\)
\('[q r]\)

Another example:

the second element of list \(x\) —if it exists—is \((\text{cadr} x)\)
the third, fourth, ... elements—if they exist—are \((\text{caddr} x)\), \((\text{cadddr} x)\), etc.

A list is defined recursively:

A list is non-empty if the length of the list is a non-zero number.
A list is empty if the length of the list is zero.

\((\text{null? x)}\) is true if \(x\) is the empty list.
Function expressions and definitions of functions

\[(\text{define } (\text{square } x) (* x x))\]
or
\[(\text{define } \text{square} \ (\lambda (x) (* x x)))\]

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:
\[(\text{cond } (C_1 \ E_1) (C_2 \ E_2) \cdots \ (C_n \ E_n))\]

The last part, \(E_{n+1}\), 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.

More examples of functions in Scheme:

\[(\text{define } (\text{same-neighbors? } l) \ (\text{cond } (\text{null? } l) \ #f \ (\text{null? } (\text{cdr } l)) \ #f \ (\text{equal? } (\text{car } l) \ (\text{cadr } l)) \ #t \ (\text{else } (\text{same-neighbors? } (\text{cdr } l))))))\]

The conditional schema, a special case:

\[(\text{cond } ((\text{same-neighbors? } l) \ (\text{else } \#f)))\]

can be abbreviated as

\[(\text{cond } (\text{null? } l) \ #f \ (\text{null? } (\text{cdr } l)) \ #f \ (\text{equal? } (\text{car } l) \ (\text{cadr } l)) \ #t \ (\text{else } \#f))\]

or

\[(\text{cond } ((\text{lambda } (x) (* x x)) \ (\text{else } \#f)))\]

Expression and definitions of functions

\[(\text{define } \text{square} \ (\lambda (x) (* x x)))\]
or

\[(\text{define } (\text{square } x) (* x x))\]
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))))

Stack operations in Scheme
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:

\[ (E_1 \ E_2 \ \ldots \ E_n) \rightarrow (f \ E_1 \ f \ E_2 \ \ldots \ f \ E_n) \]

\[
\text{define (map f l)}
\text{if (null? l) l}
\text{letrec (reduce f f0 l)}
\text{reduce (reduce f f0 (cdr l))}
\]

For example:

\[
\text{(map (lambda(x) (+ x 1)) '(1 2 3))}
\]

\[
\text{For example, this gives (2 3 4):}
\]

Reducers

Let \( f \) be a binary operation, that is, a two-argument function. Let \( f_0 \) be a constant. We want to express the following transformation:

\[ (E_1 \ E_2 \ \ldots \ E_n) \rightarrow (f \ E_1 \ (f \ E_2 \ (f \ \ldots \ (f \ E_n \ f_0) \ \ldots))) \]

This is better written with \( f \) as an infix operator:

\[ E_1 \ f \ E_2 \ f \ \ldots \ f \ E_n \ f_0 \]

This returning a list:

\[ (E_1 \ E_2 \ \ldots \ E_n) \rightarrow (f (f (f \ldots (f E_1 (E_2 (E_3 \ldots \ E_n))))) \]

\[
\text{define (reduce f f0 l)}
\text{reduce (reduce f f0 (cdr l))}
\]

For example:

\[
\text{(reduce + 0 '(1 2 3 4))}
\]

\[
\text{gives 10}
\]

\[
\text{(reduce * 1 '(1 2 3 4))}
\]

\[
\text{gives 24}
\]

\[
\text{Let \( g \) be a constant. The example of \text{map} \text{ is \text{map}} \text{.}}
\]

Higher-order means having functions as arguments.