Programming Language Concepts

(Functional) Scheme

John McCarthy
Scheme48

- Solaris:
  ~ats/bin/scheme48

- Unix (source):
  http://s48.org/

- Windows (experimental):
  http://s48.org/1.3/windows.html
DrScheme

- Home:
  http://www.drscheme.org/

- Solaris:
  /usr/local/bin/drscheme

- requires the graphical environment.
Format

; this is a comment
(NoT #F)

- not case-sensitive
- whitespace separates words
- special characters can be (in) identifiers
- some special characters are significant:
  ( ) [ ] { } , ; : ' " \ \ | #
- minus, period, and digits cannot start identifiers
Literal

2
"string"
#t
#f
#\a
#\space

denotes “itself”
Variable

x
very-long-name
name2
+
null?

- denotes the value of it’s binding
- many standard bindings
Procedure Call

\[(\text{operator } \text{operand}_1 \text{ operand}_2 \ldots )\]

\[(* (+ 2 3) 4)\]
\[((\text{inc} 2) 3)\]

- terms evaluated prior to call
- unknown order
- higher-order: can return procedure
Definition

\[(\text{define} \ \text{variable expression})\]

- special form, controls evaluation
- binds value of expression to variable
- has no result value
- replacing a binding is allowed
Conditional Evaluation

\[
\text{(if test-exp then-exp else-exp)}
\]

\[
\text{(if (zero? 5) 1 (+ 1 2))}
\]

- evaluates test-exp
- depending on result evaluates one of the others
Data Type

- set of values
- procedures operating on the values
- representation: internally and externally

- dynamic type checking — Scheme
- static type checking — C
- a little bit of both — Java
Number

\[
\begin{align*}
2 & \ -5 \\
(+ & \ 1/3\ 4.5\ 5) \\
+ & \ -\ *\ / \\
number? \\
= \\
\leq & \ \geq \ < \ >
\end{align*}
\]

(some) arithmetic and comparison procedures take arbitrarily many arguments
Boolean

#t
#f
boolean?
eq?
not

(eq? (boolean? #f) (not #f))

- result of comparisons and predicates
- argument of conditionals
Symbol

(quote x)
'x
(define x 'x)
symbol?
(eq? x 'x)

- identifier as a value
- Note that literals (numbers, booleans, ...) are self-quoting.
List

'(1 2 3) ; list value
'() ; empty list
(+ 3 4) ; procedure call/special form

(list 'a 1 '()) ; '(a 1 ())
(cons 'a '()) ; '(a)
(cons '() '()) ; '(())
(append '(a b) '(c d)) ; '(a b c d)

ordered sequence of elements
List (2)

(car '(a b c)) ; 'a
(cdr '(a b c)) ; '(b c)
(cadr '(a b c)) ; 'b
(cddr '(a b c)) ; '(c)
(caddr '(a b c)) ; 'c

null?
eq?
equal?
Procedure

(procedure? car) ; #t
(procedure? 'car) ; #f
(procedure? (car (list cdr))) ; #t

((if (procedure? procedure?) cadr car)
  (list car cdr)) '(car in the car))

first class value — can be passed to procedures and stored in data structures.
apply (built-in)

(apply + '(1 2))
(define abc '(a b c))
(apply list (cdr abc))

(apply apply
  (list procedure? (list list apply)))

- executes a procedure value for a list of values as arguments
lambda (special form)

\[
((\text{lambda} \ (n) \ (+ \ n \ 2)) \ 4)
\]

\[
(\text{define} \ \text{add2} \ (\text{lambda} \ (n) \ (+ \ n \ 2)))
\]

\[
(\text{add2} \ 4)
\]

- creates an (anonymous) procedure value
- names in list are bound to argument values
- local to body of procedure
**map (built-in)**

```scheme
(define map
  (lambda (proc list)
    (if (null? list)
        '()
        (cons (proc (car list))
              (map proc (cdr list))))
  )
)

(map (lambda (n) (+ n 2)) '(1 2 3 4))

built-in can combine many list arguments
```
compose

(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))
  ))

(((compose car cdr) '(a b c d))

(((compose list (compose cdr cdr))
  '(a b c d))
Currying

(define sub
  (lambda (x1)
    (lambda (x2)
      (- x1 x2)
    )
  )
)

((sub 1) 2)

- sub: like – but takes one argument at a time
- any function of many variables can be represented as chain of functions on one variable
Some list functions

(append list..) concatenation of all contents of all arguments or empty list
(car list) first element from list
(cdr list) list without first element
(cons x list) constructs new list with x inserted before contents of list
(length list) number of elements
(list x..) list with x as elements or empty list
Some numerical functions

\( \text{max num ..} \)  maximum of the arguments
\( \text{min num ..} \)  minimum of the arguments
\( + \text{ num..} \)  sum of all arguments or \( 0 \)
\( - \text{ num ..} \)  cumulative difference of all arguments, need not accept more then two
\( * \text{ num..} \)  product of all arguments or \( 1 \)
\( / \text{ num ..} \)  cumulative quotient of all arguments, need not accept more then two
### Some predicates

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(boolean? x)</td>
<td><code>#t</code> if argument is <code>#t</code> or <code>#f</code>. No object satisfies more than one of this and the following predicates: <code>char?</code>, <code>number?</code>, <code>pair?</code>, <code>port?</code>, <code>procedure?</code>, <code>string?</code>, <code>symbol?</code>, and <code>vector?</code>.</td>
</tr>
<tr>
<td>(eq? x y)</td>
<td><code>#t</code> if equal atoms or identical lists</td>
</tr>
<tr>
<td>(equal? x y)</td>
<td><code>#t</code> if equal atoms or equal lists</td>
</tr>
<tr>
<td>(not x)</td>
<td><code>#t</code> only if <code>x</code> is <code>#f</code>; everything else acts as true in a condition</td>
</tr>
<tr>
<td>(null? x)</td>
<td><code>#t</code> only if <code>x</code> is the empty list</td>
</tr>
</tbody>
</table>
Some number predicates

\begin{itemize}
\item \texttt{(exact? num)} \quad \#t if argument is exact (integer or rational values)
\item \texttt{(even? num)} \quad \#t if argument is even
\item \texttt{(odd? num)} \quad \#t if argument is odd
\item \texttt{(= num ..)} \quad \#t if all arguments are equal
\item \texttt{(< num ..)} \quad \#t if all arguments form numerically ordered sequence
\item \texttt{(<= num ..)}
\item \texttt{(>) num ..)}
\item \texttt{(>= num ..)}
\end{itemize}
Special forms

(define name value) binds unbound name to new location and assigns value — which can be a lambda expression

(if test then) (if test then else) depending on test, then or else is evaluated and value (if any) is returned

(lambda (name..) body ..) defines lambda expression; when applied, arguments are bound to each name and then each body is evaluated; last value is returned

(lambda name body ..) same, but list of argument values is bound to name

(quote x) return x unevaluated

'x
Some macros

(and x..)
evaluates arguments in order until #f or last; returns #f or last or #t

(cond
 (test body ..)
 ..
 (else body ..)
)
evaluates each test until first #t; evaluates each body after that test; returns value of last body if any

(let
 ((name init) ..
 )
 body ..
)
simultaneously binds each name locally to the value of init; sequentially evaluates each body; returns last value.

(or x..)
dual to and
Euclid’s algorithm

(define euclid
 (lambda (x y)
   (cond
     ((> x y) (euclid (- x y) y))
     ((< x y) (euclid x (- y x)))
     (else    x)
   )
 ))
)
Factorial

(define factorial
  (lambda (n)
    (if (> n 1)
        (* n (factorial (- n 1)))
        1)
  )
)
(define factorial
  (letrec
    ((helper
       (lambda (result n)
         (if (> n 1)
             (helper (* result n) (- n 1))
             result
         )
       )
     )
   )
  )
)

Scheme supports arbitrary tail recursion.
Factorial — “loop”

(define factorial
  (lambda (n)
    (let loop
      ((result 1)
       (n n))
      (if (> n 1)
        (loop (* result n) (- n 1))
        result )))

let loop defines a local procedure.
Reverse a list

(define rev
  (lambda (l)
    (cond
      ((symbol? l) l)
      ((null? l) l)
      (else
       (append (rev (cdr l)) (list (car l))))
    ) ) )

- car cdr pattern: deal with first element explicitly and with the rest recursively.
Reverse a list — efficient

(define rev
  (lambda (list)
    (let loop
      ((list list)
       (result '()))
      (if (null? list)
        result
        (loop (cdr list) (cons (car list) result)))
    )
  )
)

cons is more efficient then append.
Prefix to infix

(define infix
  (lambda (expr)
    (cond
      ((symbol? expr) expr)
      ((= (length expr) 1) (car expr))
      ((= (length expr) 2) expr)
      (else
       (list (infix (cadr expr))
             (car expr)
             (infix (caddr expr)))
     ) ) ) )

recurse on the structure.
Quicksort

unsorted

\[
\begin{align*}
\text{low} & & \text{high} \\
\text{exchange} & & \\
< x & \geq x & \leq x & > x \\
\text{low} & & \text{high} \\
\text{high} & & \text{low}
\end{align*}
\]
Quicksort

(define sort ; returns sorted list of numbers
  (lambda (relation numbers)
    (letrec
      ( ;... helpers
        (quicksort ; recursive part
          (lambda (numbers)
            (let
              ((relation? ; checks x and first number
                  (lambda (x) (relation x (car numbers))))
              (if (null? numbers)
                '()
                (append
                  (quicksort (select relation? (cdr numbers)))
                  (list (car numbers))
                  (quicksort (select (not relation?) (cdr numbers)))
                  ))))))
    (quicksort numbers)
  )))
Quicksort — helpers

(define sort ; returns sorted list of numbers
  (lambda (relation numbers)
    (letrec
      ((not ; returns complemented relation
         (lambda (relation)
           (lambda list (if (apply relation list) #f #t))
         ))
      (select ; returns list of x in elements
        (lambda (filter? elements) ; with (filter? x) true
          (cond
            ((null? elements) '())
            ((filter? (car elements))
              (cons (car elements) (select filter? (cdr elements)))
            )
            (else (select filter? (cdr elements)))
          )))
    )))
Permutations

A permutation is any arrangement of the numbers from 1 to \( n \) where each number appears exactly once.

(define perms ; returns list of permutations of 1..n
  (lambda (n)
    (if (> n 1)
      (insert-into-perms n (perms (- n 1)))
      '((1))
    )
  ))
Permutations

(define insert-into-perms  ; returns list of lists from perms
  (lambda (n perms)  ; with n inserted anywhere
    (let loop ((perms perms)) ; foreach perm in perms
      (if (not (null? perms))
        (append
          (insert-into-perm n (car perms))
          (loop (cdr perms))
        )
        '()
      )
    ) ) ) )

(define insert-into-perm  ; returns list of lists
  (lambda (n perm)  ; with n inserted anywhere in perm
    (let loop ((head '()) (tail perm)) ; foreach way to split perm
      (if (not (null? tail))  ; into head and tail
        (cons
          (append head (cons n tail))
          (loop
            (append head (list (car tail)))
            (cdr tail)
          )
        )
        (list (append perm (list n)))
      )
    ) ) ) )
Space-filling curves

Around 1890 Peano and Hilbert discovered curves that converge towards a function mapping the unit interval to the unit square.
Space-filling curves

- There are four shapes: A, B, C, D

- A lower-level shape is inserted in each vertex of each shape:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>←</td>
<td>A</td>
<td>↓</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>↓</td>
<td>B</td>
<td>←</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>↑</td>
<td>C</td>
<td>→</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>→</td>
<td>D</td>
<td>↑</td>
<td>D</td>
</tr>
</tbody>
</table>
Space-filling curves

(define hilbert
  (lambda (n)
    (letrec
      ((_a (lambda (m) (append (B m) '(W) (A m) '(S) (A m) '(E) (C m)))))
      ((_b (lambda (m) (append (A m) '(S) (B m) '(W) (B m) '(N) (D m)))))
      ((_c (lambda (m) (append (D m) '(N) (C m) '(E) (C m) '(S) (A m)))))
      ((_d (lambda (m) (append (C m) '(E) (D m) '(N) (D m) '(W) (B m)))))
      (A (lambda (n) (if (zero? n) '() (_a (- n 1))))))
      (B (lambda (n) (if (zero? n) '() (_b (- n 1))))))
      (C (lambda (n) (if (zero? n) '() (_c (- n 1))))))
      (D (lambda (n) (if (zero? n) '() (_d (- n 1))))))
    (A n)
  ))

Note that case is not distinguished
Turtle graphics

Andreas Borchert explains that these curves can be generated using Lindenmayer’s string rewriting technique and turtle graphics:

\[
\text{(define hilbert}
\begin{array}{l}
\text{(lambda (n)}
\begin{array}{l}
\text{(letrec}
\begin{array}{l}
\text{((l (lambda (m) (append ; "L" -> "+RF-LFL-FR+"}
\text{'(+) (R m) '(F) '(-) (L m) '(F) (L m) '(-) '(F) (R m) '(+)))})
\text{(_r (lambda (m) (append ; "R" -> "-LF+RFR+FL-"
\text{'(-) (L m) '(F) '(+) (R m) '(F) (R m) '(+) '(F) (L m) '(-)))})
\text{(L (lambda (n) (if (zero? n) '() (_l (- n 1)))))}
\text{(R (lambda (n) (if (zero? n) '() (_r (- n 1)))))}
\end{array}
\end{array}
\end{array}
) )
\end{array}
) )
\]
code/plot/plotter.jar can be used to display either kind of output.
8 Queens — backtracking

- Problem is to position $n$ queens on an $n$ by $n$ board so that they cannot strike each other.

- Occupied rows and diagonals can be tracked as lists of numbers.

(define queen
  (lambda (n row col rows ups downs) ...)
(queen 8 1 1 '() '() '())

- starts at top left and tries to fill the board.
8 Queens — backtracking

(define queen
  (lambda (n col row rows ups downs)
    (if (> col n)
      rows
      (let
        ((result
          (if (or (member row rows) (member (+ col row) ups)
                (member (- col row) downs))
            #f
            (queen n (+ col 1) 1
              (cons row rows)
              (cons (+ col row) ups)
              (cons (- col row) downs))
          ))
        )
      (cond
        ((list? result) result)
        (else #f)
      )
    )
  )
)
8 Queens — all solutions

(define all-queen
  (lambda (n col row rows ups downs all)
    (if (> col n)
        (cons rows all)
        (let
          ((result
              (if (or (member row rows) (member (+ col row) ups)
                      (member (- col row) downs))
               all
               (all-queen n (+ col 1) 1
               (cons row rows)
               (cons (+ col row) ups)
               (cons (- col row) downs) all)
          )))
          (cond
            (((< row n)
              (all-queen n col (+ row 1) rows ups downs result))
            (else result)
          )
        )
      )
    )
  )
)
Knight’s journey

- Problem is to start a knight at the top left of an $n$ by $n$ board and visit each cell exactly once.

- The journey and therefore visited cells can be tracked as list of ordered pairs with row and column.

- The possible moves can be stored as deltas in two vectors.
Knight’s journey

(define knight ; knight's journey on n x n board
  (lambda (n)
    (let*
      ((d-row ' #(2 1 -1 -2 -2 -1 1 2)) ; row-increment
       (d-col ' #(1 2 2 1 -1 -2 -2 -1)) ; col-increment
       (last-choice (- (vector-length d-row) 1)) ; max index
       (journey
        (let loop
          ((move 2) ; try to make move'th move:
            (pos '(0 . 0)) ; from pos
            (choice 0) ; take choice'th choice
            (board ' ((0 . 0))) ; on partially filled board
          )
          ; ... produces #f or completed board
        )))
      (if journey (reverse journey) #f)
    )
  )
)
Knight’s journey

(if (> move (* n n))
  board ; enough moves -- success
  (let*
    ((next-pos ; pos after choice or #f
      (let
        ((row (+ (car pos) (vector-ref d-row choice)))
         (col (+ (cdr pos) (vector-ref d-col choice))))
        (if (or (< row 0) (>= row n) (< col 0) (>= col n)
                (member (cons row col) board))
          #f
          (cons row col))))
     (result ; completed journey or #f
      (if next-pos
        (loop (+ move 1) next-pos 0 (cons next-pos board))
        #f)))))
  (cond
    (result result)
    ((< choice last-choice)
      (loop move pos (+ choice 1) board))
    (else #f)))))))