Symbolic Programming

Artificial Intelligence
Chapter 2

Symbolic Programming

Create symbols and symbolic expressions:
- fred, block1, block2, (height fred), (on block1 block2)

Pass them around as arguments to procedures.

Manipulate them to infer new expressions:
- (on block1 block2), (on block2 block3)
  \[ \Rightarrow \text{(stack block1 block2 block3)} \]

Symbolic Programs for Playing Chess

- Symbols: chess pieces
- Expressions: legal moves

eg.: black-night1, black-night2
  (sequence (forward x 2) (right x 1))

AI in Practice: Playing Chess
2.1 Rule-Based Reactive System
Example

- Symbols: sensors, controls
- Sensors names:
  forward, right, left, jright, jleft, rear
  ⇒ {near, away, far}
- Control parameters: speed, direction
  speed ⇒ {zero, slow, fast}
  turn ⇒ {left, straight, right}

2.2 Introduction to Lisp

- Programs and data are represented as lists,
  i.e. ( expressions )
- An expression is a Lisp object or a list of zero or
  more expressions.
- Examples: (), (1), (1 (2))
- Symbols: sym14, sym-one, sym_two, foo, FOO, Foo, t, nil
- A Lisp program consists of a sequence of Lisp
  expressions.
  e.g.: (defun square (x) (* x x ))

2.3 Interacting with Lisp

- Lisp interpreter: read-eval-print loop
- eval and apply
  > "string"
  "string"
  > (+ 3 4 5)
  12
  > (square 2)
  4
2.4 Functions in Lisp

- `defun`: function definition
  ```lisp
  (defun hypotenuse (a b)
    (sqrt (+ (square a) (square b))))
  (defun square (x)
    (* x x))
  > (square 3)
  9
  > (hypotenuse 3 4)
  5
  ```

Eval and Apply

- Eval takes the list and extracts the first element of the list and the list of arguments.
- Eval passes the function associated with the symbol and the list of argument to apply.
- Apply uses eval to evaluate the list of arguments.
- Apply extracts the function definition and check the number of arguments.
- Apply sends the evaluated arguments to eval to be evaluated.

Conditional Statements

- `(if test conditional alternative)`
  ```lisp
  (defun next-odd-number (n)
    (if (evenp n)
      (+ n 1)
      (+ n 2)))
  > (next-odd-number 1)
  3
  ```

Conditional Statements: `cond`

- `(cond clauses). clauses := (test body)*`
  ```lisp
  (defun primep (n)
    (cond ((>= n 49) (princ "Can’t count that height.")) nil)
      ((or (= n 2) (= n 3) (= n 5)) t)
      ((or (= (mod n 2) 0)
        (= (mod n 3) 0)
        (= (mod n 5) 0)) nil)
      (t t))
  ```
Using Defined Functions

> (primep 32)
NIL
> (primep 37)
T
> (primep 76)
Can't count that high.
NIL

Recursive Functions

- Reduce the original problem to simpler problems, and then apply itself to solve the simpler problems.
- Break the problem down again, and again, until the pieces of the problem can be solved easily.
- Problem reduction
- Divide-and-conquer

Recursive Functions: An Example

- Lisp Example
  (defun raise (x n)
    (if (= n 0)
        1
        (* x (raise x (- n 1)))))

> (raise 3 3)
27

- Evaluating Functions in Files
  > (load "my_program.lisp")

2.5 Environments, Symbols, and Scope

- Assigning values to symbols
  > (setq sym 2)
  2
  > sym
  2
  > (setq sym 3)
  3
  > sym
  3
  > (setq new sym)
  3
  > new
  3
# Eval

1. If \( x \) is a number or string, return \( x \).
2. If \( x \) is a symbol, then look up its value in the environment.
3. If \( x \) a special form, handle it accordingly.
4. If \( x \) a list, send the function (first item) and arguments (the rest of items) to apply.

---

# Apply

1. Look up the definition of the function.
2. Use `eval` to evaluate each argument in the environment.
3. Create a new environment.
4. Use `eval` to evaluate the definition in the new environment.

---

# Structured Environments

- An environment allocates storage for symbolic values.
- Global environments as a large table.
- A new environment points to the existing parent environment.
- To determine the value of a symbol, `eval` first looks in the table pointed by the environment. If no entry exists, then `eval` looks in the parent environment.

---

# Scope

```scheme
> (setq x 2)
2
> (defun local (x)
  (setq x (+ x 1))
  (+ x x))
LOCAL
> (local (+ x 1))
16
> x
2
> (let ((x 1)
        (let ((x 3))
          (printc x))
       (princ x))
321
1
```
Let: Introducing Local Variables

(defun guess ()
  (princ "Guess an integer from 0 to 9: ")
  (let ((resp (read)) (num (random 10)))
    (cond ((> resp num) (princ "Too high!"))
          ((< resp num) (princ "Too low!"))
          (t (princ "Lucky guess!")))))

GUESS
> (guess)
Guess an integer from 0 to 9: 3
Too low!

2.6 More on Functions

- Functions with Local State

  > (let (x (y 1))
        (defun squarelast (z)
          (setq x y) (setq y z) (* x x)))

  SQUARELAST
  > (squarelast 2)
  1
  > (squarelast 3)
  4

- Lambda Functions

  - defun: define named functions.
    > (defun square (x) (* x x))
  - lambda: creates unnamed functions.
    > (lambda (x) (* x x))
  - function: evaluates lambda functions or a symbol defined as a function.
    > (function (lambda arguments body))
    > (function #lambda arguments body))
    > (function square)
Functions as Arguments and `funcall`

```lisp
> (funcall #\'((lambda (x) (* x x)) 3))
9
> (defun decreasingp (x y f)
  (f (> (funcall f x) (funcall f y)) t nil))
DECREASINGP
> (decreasingp 1 2 #\'((lambda (x) (* x x))))
NIL
> (setq reciprocal #\'((lambda (x) (/ 1 x)))
#\<interpreted--Function (LAMBDA (X) (/ 1 X)) 104BB75>
> (decreasingp 1 2 reciprocal)
T
```

2.7 List Processing

```lisp
> (quote expression)
SYM
> (quote sym)
SYM
> 'sym
'(first second third)
(FIRST SECOND THIRD)
```

Building and Accessing Elements in Lists

```lisp
> (setq four (list 1 2 3 4))
(1 2 3 4)
> (list (first four)
(second four) (nth 2
four))
(1 2 3)
> (rest four)
(2 3 4)
> (setq first 1)
1
> (setq rest (list 2 3 4))
(2 3 4)
> (cons first rest)
(1 2 3 4)
```

Lists in Memory

```lisp
> (car x)
1
> (cadr y)
2
> (cadr (cadr y))
2
```
Lists in Memory: Illustrations

Modifying Existing List Structures

Comparing List Structures: eq and equal

Built-In List Manipulation Functions

<table>
<thead>
<tr>
<th>(setq x (cons 1 2))</th>
<th>(setq y (cons 1 (cons 2 ())))</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>1</td>
<td>1 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setq x (list 1 (list 1)))</th>
<th>(setf (first (rest x)) x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setq x (list 1 (list 2)))</th>
<th>(1 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (setq x (list 1 (list 2)))</td>
<td></td>
</tr>
<tr>
<td>(1 2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setf (first (rest x)) x)</th>
<th>(1 (1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (setf (first (rest x)) x) 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>&gt; x</td>
<td></td>
</tr>
<tr>
<td>(1 (1))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setq y (cons 1 (cons 2 ())))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setq x (list 1 (list 1)))</th>
<th>(setf (first (rest x)) x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(setq x (list 1 (list 1)))</td>
<td>(setf (first (rest x)) x)</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setq sym 'foo)</th>
<th>(FOO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (setq sym 'foo) FOO</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(eq sym 'foo)</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (eq sym 'foo) T</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setq sym (list 'foo))</th>
<th>(FOO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (setq sym (list 'foo)) (FOO)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(eq sym (list 'foo))</th>
<th>NIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (eq sym (list 'foo)) NIL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(setq new sym)</th>
<th>(FOO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (setq new sym) FOO</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(eq new sym)</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (eq new sym) T</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(equal sym (list 'foo))</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (equal sym (list 'foo)) T</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(reverse x)</th>
<th>(4 3 2 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (reverse x) (4 3 2 1)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(member 5 (append '(1 2 3) '(4 5) '(6 7 8)))</th>
<th>(5 6 7 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (member 5 (append '(1 2 3) '(4 5) '(6 7 8))) (5 6 7 8)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(conc x y)</th>
<th>; destructive</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (conc x y) ; destructive</td>
<td></td>
</tr>
<tr>
<td>(1 2 3 4)</td>
<td></td>
</tr>
<tr>
<td>&gt; x</td>
<td></td>
</tr>
<tr>
<td>(1 2 3 4)</td>
<td></td>
</tr>
</tbody>
</table>
Optional Arguments

> (member '(2) '((1) (2) (3))) ;; eq is used
NIL
> (member '(2) '((1) (2) (3))
:test #'equal)
((2) (3))
> (member '(1 2) '((0 4) (1 3))
:test #'lambda (x y) (eq (first x)
(first y))))
((1 3))

List-Processing Examples: Homeworks

• insert (p. 49)
• search (p. 50)
• nearest-pairs (p. 50)
• remember and estimate (p. 51)
• interpolate (p. 40)
• Data Abstraction:
  TREE-sub (p. 52-53)

2.8 Iterative Constructs: mapcar

• mapcar, mapc, mapcan, reduce

> (mapcar #'+ '(1 2 3) '(4 5 6))
(5 7 9)
> (mapcar #'(lambda (x y)
(if (> x y) x y))
'(2 7 5) '(1 9 4))
(2 9 5)
> (reduce #'+ '(1 2 3))
6

do

• (do index-var-specs (end-test result) body)
• index-var-specs := (step-var init-val step-val)

> (do ((i 1 (+ i 1)) ; index-var-specs
   (j (* j i)))
  ((= i 10) j) ; (end-test result)
  (princ i)) ; body
123456789
362880
dolist, dotimes

- (dolist (var expr result) body) ; var is bound to elements of list expr
- (dotimes (var expr result) body) ; var is an integer resulting from expr

> (dolist (x '(a b c)) (princ x))
  ABC
  NIL
> (dotimes (i 10 i) (princ i))
  0123456789
  10

2.9 Monitoring and Debugging Programs

- Tracing the programs
  > (trace raise square)
  (RAISE SQUARE)
  > (square (raise 2 2))
  1 Enter RAISE 2 2
  | 2 Enter RAISE 2 1
  | ...
  | 2 Exit RAISE 2
  1 Exit RAISE 4
  1 Enter SQUARE 4
  1 Exit SQUARE 16
  > (untrace raise)
  (RAISE)

Formatted Output

> (format nil "~D integer: ~A symbol" 17 'foo)
  "17 integer: FOO symbol"
> (format nil "~4,2F real" 1.2345)
  "1.23 real"
(format nil "Here~%is a line break.")
  "Here
  is a line break."

2.10 Rule-Based Reactive System Revisited

- Homework
  > (assoc 'left far '(((left far) 1)))
  NIL
  > (assoc 'left far '(((left far) 1)) :test #'equal)
  ((left far) 1)
  > (assoc 'left '(((left far) 1)) :key #'first)
  ((left far) 1)
Exercises

- (switch (a b)) ⇒ (b a)
- (switch '(a b c d) 2 4) ⇒ (c b a d)
- (factorial 3) ⇒ 6
- (distance '(2 5) '(3 6)) ⇒ 1.4
- (remove 'b '(a b c b d)) ⇒ (a c d)
- (intersect '(a b b c d) '(c a b b)) ⇒ (a b c)

Q: Explain in detail what eval does when it encounters (cubic y) in the following Lisp session.

```lisp
(defun cubic (x) (* x x x))
(splot y 3)
(cubic y)
```