MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Project MAC

Artificial Intelligence Project
Memo 64--

Memorandum MAC-M-134
January 24, 1964

LISP Exercises

by Timothy P. Hart and Michael I. Levin

The following exercises are carefully graded to mesh with the sections in Chapter I, "The LISP Language," in the LISP 1.5 Programmer's Manual. Each exercise should be worked immediately after reading the manual section indicated.

Instructions to "Read Section 1.1" refer to this manual. The exercises are more finely sub-divided than the manual sections. If you don't understand an entire manual section try the exercises--they may teach you enough to allow you to proceed on through the manual section.

A. S-expressions. Read Section 1.1.

Which of the following are s-expressions?

1. (X . Y)
2. ((Z)
3. CHI
4. (BOSTON . NEW . YORK)
5. ((SPINACH . BUTTER) . STEAK)

B. Elementary Functions. Read Section 1.2.

Write the s-expression which is the value of each of the following expressions. (Some of them are undefined!)

6. car[(A . B)]

7. cdr[(A . B)]

8. car[((FOO . CROCK) . GLITCH)]

9. cdr[((FOO . CROCK) . GLITCH)]

10. car[X]

11. cons[POOT;TOOP]

12. cons[X;(Y . Z)]

13. eq[X;X]

14. eq[Y;Z]

15. eq[(Q . R);(Q . R)]

16. atom[X]

17. atom[(TOOT . TOOT)]

18. atom[ISTHISATRIVIALEXERCISE]

C. Function Composition

The notation of function composition is extremely useful: car[cdr[(A . (B . C))]] means that s-expression which is the car part of the cdr part of (A . (B . C)), namely B. The general rule for evaluating composed functions is to evaluate the innermost ones first, then the next outer layer and so on until the entire expression has been evaluated.

Example

cons[cdr[((A . B) . C)];car[((A . B) . C)]] = cons[C;(A . B)] = (C . (A . B))

Evaluate the following:

19. car[cons[A;B]]

20. cons[car[(A . B)];cdr[(A . B)]]

21. car[cdr[(A . (B . C))]]

22. eq[car[(A . B)];cdr[(C . B)]]

23. eq[cdr[(A . B)];cdr[(C . B)]]

24. eq[A;car[cons[car[(A . B)];C]]]

25. atom[cons[A;B]]

26. atom[cdr[car[((A . C) . B)]]]

27. car[cdr[car[((A . (B . C)) . D)]]]

D. Variables.

The notion of a variable is very important: a small letter appearing in an expression in places where we expect to find an s-expression is to stand for the same (unspecified) s-expression in every place where it occurs in that s-expression. Using this notation we can give the following definitions:

car[(x . y)] = x

cdr[(x . y)] = y

cons[x;y] = (x . y)

(Notice that these definitions don't allow us to find an s-expression equivalent to car[ATOM]; this is why we say that such an expression is undefined.)

Which of the following identities are always true?

28. car[cons[x;y]] = x

29. cons[car[cons[x;y]];y] = (x . y)

30. atom[cons[u;v]] = T

31. cons[A;cons[x;Y]] = (A . (x . Y))

32. car[car[cdr[(A . ((B . C) . D))]]] = C

33. atom[cdr[cons[x;Y]]] = T

34. eq[X;car[cons[X;cons[u;v]]]] = T

E. List Notation. Read Section 1.3.

We will use the list notation almost exclusively from now on, so do not try to avoid learning it. The second set of examples on manual page 4 are key prototypes for remembering what the basic functions do to s-expressions in list notation:

car gets the first element of a list.

cdr eliminates the first element of a list by moving the leading left paren past the initial s-expression.

cons inserts its first argument at the head of the list which is its second argument.

cdr of a list with only one element is the atomic symbol NIL.

Which one of the following s-expressions cannot be expressed in list notation? Translate the rest into list notation.

35. ((A . NIL) . ((B . NIL) . NIL))

36. (A . (B . (C . NIL)))

37. (NIL . NIL)

38. ((A . (B . NIL)) . ((C . NIL) . NIL))

39. ((X . NIL) . ((NIL . Y) . NIL))

Translate the following s-expressions into dot notation.

40. A

41. ((PLOOP) FLOP TOP)

42. ((X GLITCH) (Y CROCK))

43. (((X)))

44. (SNAP (CRACKLE (POP)))

Write the s-expression which is the value of each expression. Use list notation for your answer whenever possible. Some of them may be undefined.

45. car[(A B C)]

46. cdr[(A B C)]

47. car[(AB DC)]

48. car[(A)]

49. cdr[(A)]

50. car[A]

51. cdr[NIL]

52. cons[A;(B)]

53. cons[A;(B C)]

54. cons[A;B]

55. cons[A;NIL]

56. cons[NIL;A]

57. cons[(A);(B C)]

58. cons[(A B);(C D)]

59. cons[(A B);C]

60. cons[(A . B);((C . D)(E . F))]

61. cons[(A B);NIL]

62. car[(((A)))]

63. cons[eq[A;A];(F T F)]

64. atom[NIL]

F. Multiple car-cdr Functions

Write multiple car-cdr LISP functions which will find the "A" in each of the following (denote the s-expressions by "x"):

65. (A)

66. (C A T)

67. (B . A)

68. (S T A Y)

69. (1 2 3 A)

70. ((A . B) (C . D))

71. ((B . A) (C . D))

72. (((C)) ((A)))

73. ((X . Y) (A . B))

74. ((((A))))

G. The Functions list and null.

We will use the predicate null to test for the atomic symbol NIL. null[NIL] = T, null[x] = F , if x is any other s-expression than NIL. The expression "( )" is identical to NIL, that is, ( ) = NIL, so null[( )] = T and atom[( )] = T, since null[NIL] = T and atom [NIL] = T. Notice this consistency: cdr[(A)] = ( ) = NIL.

The function list is used to create lists. It is a shorthand notation as the following identities illustrate:

list[ ] = ( ) = NIL

list[x] = (x) = cons[x;NIL]

list[x_1;x_2] = (x_1 x_2) = cons[x_1;cons[x_2;NIL]]

.
.
.

list[x_1;x_2;...;x_n] = (x_1 x_2 ...x_n) = cons[x_1;cons[x_2;...cons[x_n;NIL]...]]

Examples

list[A] = (A)

list[A;B] = (A B)

list[X;(Y Z)] = (X (Y Z))

Exercises

75. null[A]

76. null[NIL]

77. null[()]

78. null[car[(A)]]

79. null[cdr[(A)]]

80. null[cdr[((A B C))]]

81. list[A;B;C]

82. list[(A);(B);(C)]

83. null[cdr[list[A]]]

84. null[cdr[cons[A;B]]]

Write expressions involving only cons, list, and atoms which will generate the following expressions. Example: (A B) = list[A;B]

85. (A . B)

86. (X Y)

87. ((A B) (C D))

88. ((A . B) (C . D))

89. (((A)))

90. ((B . C))

91. (( ) ( ))

92. (A . (B C D))

Evaluate:

93. eq[car[(A)];car[(B)]]

94. eq[cdr[(A)];cdr[(B)]]

95. atom[cdr[(X Y)]]

96. atom[cdr[cdr[(X Y)]]]

97. car[(A B C)]

98. cadr[(A B C)]

99. caddr[(A B C)]

100. cddr[(A B C)]

101. cadr[A (B C) D)]

102. caadr[(A (B C) D)]

103. cadadr[(A (B C) D)]

104. cddadr[(A (B C) D)]

105. cddr[(A)]

106. eq[car[(A)];cadr[(B A)]]

107. cadr[list[A;B;C]]

108. caddr[list[A;B;C]]

109. atom[cadr[(A (B) C)]]

110. atom[cadr[(A B C)]]

111. cdr[cons[A;(B . C)]]

112. atom[car[cons[A;(B C)]]]

113. atom[cdr[cons[A;NIL]]]

114. null[list[A]]

115. eq[(A . B);cons[A;B]]

116. car[cons[car[(A)];NIL]]

117. eq[A;car[(A)]]

118. eq[(A B);list[A;B]]

H. <u>The LISP Meta-Language</u>. Read Section 1.4. (Skip Section 1.5.)

119. [T→ A;T→ B]

120. [F →A;T→ B]

121. [F→A;F →B]

122. [eq[A;A] → car[(A)];T → cdr[(B)]]

123. [null[X] → Y;null[( )] → NIL;T → atom[A]]

124. [atom[x] → atom[x];T → eq[X;X]]

In exercises 125-133 the variables f, m, n, t, x, y, and z are bound by the following table. All other variables are unbound and therefore undefined.

f = F	t = T	z = AB
m = AB	x = ((AB))	
n = (AB . C)	y = ((AB . C))	

125. [eq[m;z] → n;T → x]

126. [f → A;T → B;T → c]

127. [eq[AB;m] → A;T → B]

128. [atom[m] → A;T → B]

129. [atom[n] → A;T → B]

130. [eq[m;caar[x]] → y;T → w]

131. [[T → F;F → T] → F;T → [F → T;T → T]]

132. [[eq[cdr[n];cdar[y]] → F;T → T] → y;T → w]

133. [[eq[m;z] → eq[f;t] → null[cdr[x]]] → B;T → C]

I. λ Notation and λ Conversion.

The λ notation is necessary for specifying the order in which a function expects its arguments. Understand the difference between a function and a form.

Examples

form:	cons[x;y]
functions:	λ[[x;y];cons[x;y]]

The process of pairing the elements of a list of variables following a λ with their respective arguments to form a table of bindings such as in the recent exercise, and then evaluating the form inside the λ expression is called λ-conversion.

<u>Example</u>: λ-conversion

form:	table
λ[[x;y]cons[y;x]][A;B] =	x = A
cons[y;x] =	Y = B
cons[B;A] =	
(B . A)	

Evaluate:

134. λ[[x];x][A]
135. λ[[y];y][(A)]
136. λ[[y];A][B]
137. λ[[x];car[x]][(A)]
138. λ[[x];car[(A)]][(B)]
139. λ[[u;v];u][A;B]
140. λ[[u;v];u][B;A]
141. λ[[u;v];v][A;B]
142. λ[[x;y];cons[car[y];cdr[x]]][(A . B);(C . D)]
143. λ[[x];λ[[y];car[y]][x]][(A)]
144. λ[[x;y];[atom[x]→y;T→A]][R;S]
145. cons[A;[null[Q]→B;T→C]]
146. cons[A;λ[[x];car[x]][(B)]]
147. cons[A;[λ[[x;y];y][T;F]→B;T→C]]

J. Function Definition.

Unse the following table to evaluate the forms below:

	u = (A B)
	v = X
	w = (T U V)
	pred = λ[[x;y];eq[x;car[y]]]
	test = λ[[x];[atom[x]→T;atom[car[x]]→F; T→F]]
<u>Example</u>	
pred[A;(A B)] =	
λ[[x;y];eq[x;car[y]]][A;(A B)]=	
	x = A
	y = (A B)
eq[x;car[y]] =	
eq[A;car[(A B)]] =	
eq[A;A] =	
T	
148. test[A]	
149. test[(u B C)]	
150. test[(v . u)]	
151. pred[X;(A B)]	
152. pred[F;(F T F)]	
153. pred[test[v];w]	

K. Recursive Functions

Use the definition of jj given below to evaluate the following forms. Write the argument for jj at each recursive step in the evaluations.

```
jj = λ[[x];[atom[x]→x;T→jj[cdr[x]]]]
```

154. λ

155. (A . B)

156. ((X . Y) . (X . Z))

157. (A B C)

158. (A (C . E))

Use this definition to evaluate the folowing forms.

```
match =λ[[kk;m];[null[kk]→NO;null[m]→NO;eq[car[kk];car[m]]→car[kk];
                 T→match[cdr[kk];cdr[m]]]
```

159. match[(X);(X)]

160. match[(A B E);(J O E)]

161. match[(X A Y);(S V E)]

Use the following definition to evaluate these forms.

```
twist =λ[[s];[atom[s]→s;T→cons[twist[cdr[s]];twist[car[s]]]]]
```

162. twist[A]

163. twist[(A . B)]

164. twist[((A . B) . C)]

165. twist[(A B C)]

166. twist[((A . B))]

L. List vs. s-expression recursions

Recursive LISP functions are generally terminated in wither of two ways. In operations which deal with data which are lists (that is can be expressed in list-notation) the termination is by <u>null</u> when a datum is exhausted. In functions dealing with dot notation data (that is with data in which atoms other than NIL may occur in the <u>cdr</u> part) the corresponding terminating condition is <u>atom</u>.

<u>Examples</u>

<u>null</u>-termination is used in a recursion down a list in the function <u>among</u> which decides whether or not an atom is among those on a list:

```
among = λ[[a;kk];null[kk]  F;eq[a;car[kk]]  T;T  among[a;cdr[kk]]]]
```

<u>atom</u>-termination is used in the recursion in <u>inside</u> which decides whether or not an atom appears anywhere in an s-expression:

```
inside =λ[[a;s];[atom[s]   eq[a;s];
                 inside[a;car[s]]  T;
                 T   inside[a;cdr[s]]]]
```

<u>Exercises</u> (167-175 are <u>list</u> type; 176-178 are s-expression type.)

Write LISP functions for the following purposes:

167. To determine whether an atom is a member of a list

```
e.g.  member[B;(A B C)] =T
      member[X;(A B C)] =F
      member[A;(B (A B) C)] = F
```

168. to produce a table (list of dotted pairs) given two lists, one of the references, the other of values.

```
e.g. pair[ONE TWO THREE);(1 2 3)]=((ONE . 1)(TWO . 2)(THREE . 3))
     pair[(PLANE SUB);(B47 THRESHER)]=((PLANE . B47)(SUB . THRESHER))
```

169. to append one list onto another.

```
e.g. append[(A B C);(D E F)] = (A B C D E F)

     append[((A B) C(D(E)));((A))] = ((A B) C(D(E)) (A))
```

170. to delete an element from a list.

```
    e.g. delete[Y;(X Y Z)] =(X Z)
         delete[X;((U V) X Y)] =((U V) Y)
```

171. to reverse a list. (hint: use <u>append</u>.)

```
    e.g. reverse[(A B C)] = (C B A)
         reverse[(A (B C) D)] =(D (B C) A)
```

172. to rpoduce a list of all the atoms which are in either of two lists.

```
    e.g. union[(U V W);(W X Y)] =(U V W X Y)
         union[(A B C);(B C D)] = (A B C D)
         union[(A B C);(A B C)] = (A B C)
```

173. to produce a list of all the atoms in common to two lists.

```
    e.g. intersection[(A B C);(B C D)] =(B C)
         intersection[(A B C);(A B C)] =(A B C)
         intersection[(A B C);(D E F)] = NIL
```

174. to find the last element on a list.

```
    e.g. last[(A B C)] = C
         last[((A B)(C))[ = (C)
```

175. to reverse all levels of a list.

```
    e.g. superreverse[(A B(C D))] =((D C) B A)
         superreverse[((U V)(X Z) Y))]=((Y (Z X)) (V U))
```

176. to determine whether a given atomic symbol is some part of an s-expression.

```
    e.g. part[A;A] = T
         part[A;(X . (Y . A))] = T
         part[A;(U V(W . X) Z)] = F
```

177. to substitute one atomic symbol for another in an s-expression.

```
e.g.    subst[X;Y;(U V W X Y Z)] = (U V W X X Z)
        subst[X;Y;((U Y) X((Y) Z))] =((U X) X ((X) Z))
```

178. to produce a list of all the atoms in s-expression.

```
e.g.    listofatoms[(A (B C) D)] = (A B C D)
        listofatoms[((B (C D)) E)] = (B C D E)
        listofatoms[(A . (B . C))] = (A B C)
```

M. A Differentiation Program.

A simple differentiation program can easily be written in LISP and it will serve two purposes for us: it will be an example of an application of LISP; and it illustrates the usefulness of prefix notation in representing algebraic expressions.

The differentiation rules which we will be implementing are:

$$\frac{dx}{dx} = 1 \qquad \text{(rule 1)}$$

$$\frac{dy}{dx} = 0 \ (y \neq x) \qquad \text{(rule 2)}$$

$$\frac{d(u + v)}{dx} = \frac{du}{dx} + \frac{dv}{dx} \qquad \text{(rule 3)}$$

$$\frac{d(u \cdot v)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx} \qquad \text{(rule 4)}$$

The LISP function diff[e;x] is to differentiate the algebraic expressions e with respect to the variable x. Clearly some way of representing algebraic expressions as s-expressions must be invented, since the arguments of diff must be s-expressions.

There are many possible ways to represent an algebraic expression as an s-expression, e.g., y +zw might become (Y PLUS Z TIMES W) or (Y SUM(ZPRDCT W)) or (PLUS Y (TIMES Z W)).

We shall choose this last representation (called Polish prefix notation) for reasons of convenience, and will specify it as follows:

1. algebraic constants and variables shall become atoms, e.g., y→Y, ρ→RHO, 37→THIRTYSEVEN

(The programmed versions of LISP all handle numbers more conveniently than this)

2. the operators + and . shall be limited to two operands by associative grouping, e.g.

x + y + z→x + (y + z)

u . v . w→u . (v . w)

3. We will use the notation convention that an expression followed by * means the translation of that expression.

x + y shall become (PLUS x* y*) and x . y shall become (TIMES x* y*).

The following examples should clarify these rules.

Algebraic Expression	S-expression Representation
a + b	(PLUS A B)
a . b	(TIMES A B)
a . (b + c)	(TIMES A (PLUS B C))
α + β γx	(PLUS ALPHA (PLUS BETA (TIMES GAMMA X)))
2πr	(TIMES TWO (TIMES PI R))

The program for diff using this representation for algebraic expressions is straightforward:

```
diff = λ[[e;x]:
[atom[e]→[eq[e;x]→ONE;                                        (rule 1)
          T ⟶ ZERO];                                          (rule 2)

eq[car[e];PLUS]→list[PLUS;diff[cadr[e];x];diff[caddr[e];x]]    (rule 3)
eq[car[e];TIMES]→list[PLUS;list[TIMES;caddr[e];diff[cadr[e];x]]

     list[TIMES;cadr[e];diff[caddr[e];x]]]]]                   (rule 4)
```

Exercies. Evaluate these forms:

179. diff[(PLUS A X);X]
180. diff[(TIMES A X);X]
181. diff[(TIMES THREE X);Y]
182. diff[(PLUS Y Y);Y]
183. diff[(TIMES TWO (TIMES Z Z));Z]

Adding the following translation rules for converting algebraic expressions to s-expressions allows us to add new rules to increase the power of diff.

algebra	s-expression
-x	(MINUS x*)
$\frac{1}{x}$	(RECIP x*)
sin[x]	(SIN x*)
cos[x]	(COS x*)

Additional clauses can be added to the main conditional expression of diff which implement these additional rules.

Example:

$\frac{d(-u)}{dx} = -(\frac{du}{dx})$ is implemented by adding this clause to diff:
eq[car[e];MINUS]→list[MINUS;diff[cadr[e];x]]

Exercise.

Implement the following differentiation rules by writing a clause for diff to handle each:

184. $$\frac{d(\frac{1}{u})}{dx} = \frac{\frac{du}{dx}}{u \cdot u}$$

185. $$\frac{d(\sin u)}{dx} = (\cos u) \cdot \frac{du}{dx}$$

186. $$\frac{d(\cos u)}{dx} = (\sin u) \cdot (\frac{du}{dx})$$

N. s-expression Representation of LISP Expressions.

Exercises. Translate these m-expressions into s-expressions.

187. a

188. x

189. A

190. T

191. NIL

192. ((A B))

193. QUOTE

194. (QUOTE A)

195. car

196. car[x]

197. car[A]

198. atom[x]

199. ff[x]

200. ff[car[x]]

201. [atom[x]→x;T→ff[car[x]]]

202. λ[[x];x]

203. λ[[x];car[x]]

204. λ[[x];[atom[x]→x;T→ff[car[x]]]

205. label[ff;λ[[x];[atom[x]→x:T→ff[car[x]]]

206. λ[[x];λ[[y];car[y]][x]]

O. Table Building and Searching.

Evaluate the following forms. Using the definitions of pairlis and assoc given in Section 1.6, changing the occurrence of equal in assoc to eq.

207. pairlis[NIL;NIL;((Y . B))]

208. pairlis[(X);(A);NIL]

209. pairlis[(X Y);(A B),((Z . C))]

210. assoc[y;((X . A) (Y . B) (Z . C))]

211. assoc[X;pairlis[(X);(A);((Z . C))]]

P. A LISP Interpreter

Using the Section 1.6 definitions of apply, eval, pairlis, assoc, evcon and evlis and with a = ((X. M) (Y . T) (Z . (M N)) (T . T)) evaluate the following:

212. assoc[Z;a]

213. assoc[Y;a]

214. eval[(QUOTE A);a]

215. eval[T;a]

216. eval[X;a]

217. evlis[(X);a]

218. evlis[(X Y Z);a]

219. evcon[((T X));a]

220. eval[(COND (T X));a]

221. apply[CAR;((A));a]

222. apply[CONS;(A B);a]

223. apply[CONS;((A) (B));( )]

224. apply[CAR;(A);( )]

225. eval[(ATOM X);a]

226. evcon[(((ATOM X) X) (T (FF (CAR X)))) ;a]

227. eval[(COND ((ATOM X) X) (T (FF (CAR X))));a]

228. apply[(LAMBDA (X) (CAR X));((A));( )]

229. apply[(LABEL CARP (LAMBDA (X) (CAR X)));((A));( )]

230. apply[FIRST;((A));((FIRST . CAR))]

231. apply[(LAMBDA (X) (COND ((ATOM X)X) (T (FF (CAR X)))));(A);( )]

232. apply[(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X))))));(A);( )]

233. apply[(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X))))));
(((Q . R) (S . T)));( )]

Q. Fifteen Minute Problems

Write the LISP functions described below. You need not redefine any functions you use which appeared in an earlier exercise. Assume that integers are atomic symbols.

234. infix[polish;table] is to convert an expression from Polish to infix notation, using a table which gives the correspondence between prefix and infix operators.

e.g. infix[(TIMES 3 A B);((PLUS . +) (TIMES . X))] =(3 X A X B)

infix[(DIVIDE (TIMES π α β) (PLUS 3 (TIMES α 2) A B));((PLUS . +) (TIMES . X) (DIVIDE ./))] = ((π X α X β)/(3 + (α X 2) + A + B))

235. polish[infix;table] is to do the corresponding reverse transformation.

236. permut[string] is to produce a list of all the permutations of a given string. Assume all the elements of the original string are unique.

e.g. permut[(A B C)] = ((A B C) (A C B) (B A C) (B C A) (C A B) (C B A))

237. permutations[string] is to produce a list of all the permutations of a given string.[it is difficult to do this in an efficient way.]

e.g. permutations[(A A B)] = ((A A B) (A B A) (B A A))

238. Invent the necessary s-expression representation and write clauses

for diff to implement leibnitz' rule:

$$\frac{d}{d\alpha}\int_{a(\alpha)}^{d(\alpha)} f(x,\alpha)\, dx = \int_{a}^{b} \frac{df}{d\alpha}\, dx + f(b,\alpha)\frac{db}{d\alpha} - f(a,\alpha)\frac{da}{d\alpha}$$

239. Given the predicate greater[x;y] which orders atomic symbols, i.e., if greater[x;y] = T then greater[y;x] = F where x and y are any two different atoms, write the predicate larger[x;y] which orders s-expressions, i.e., if larger[x;y] = T, then larger[y;x] = F.

240. One s-expression is a <u>factor</u> of another s-expression if the second includes the first as a sub-expression. An s-expression equivalent to a greatest common divisor is a common sub-expression which is not a factor of any other common sub-expression. (It is not necessarily unique.) Write functions to find a greatest common divisor of two s-expressions.

e.g. gcd[(A . ((B . C) . D));(E . (B . C))] = (B . C)

Write functions to find all the g.c.d.'s of two s-expressions.

e.g. allgcds[(X . (Y . (A . B)));(B . (X . (A . B)))] = ((A . B) X)

ANSWERS

1. yes
2. no
3. yes
4. no
5. yes
6. A
7. B
8. (FOO . CROCK)
9. GLITCH
10. undefined
11. (POOT . TOOP)
12. (X . (Y . Z))
13. T
14. F
15. undefined
16. T
17. F
18. T
19. A
20. (A . B)
21. B
22. F
23. T
24. T
25. F
26. T
27. B
28. true
29. true
30. false
31. true
32. false
33. true
34. true
35. ((A) (B))
36. (A B C)
37. (NIL)
38. ((A B) (C))
39. not list
40. A
41. ((PLOOP . NIL) . (FLOP . (TOP . NIL)))
42. ((X . (GLITCH . NIL)) . ((Y . (CROCK . NIL)) NIL))
43. (((X . NIL) . NIL) . NIL)
44. (SNAP . ((CRACKLE . ((POP . NIL) . NIL)) . NIL))
45. A
46. (B C)
47. AB
48. A
49. NIL
50. undefined
51. undefined
52. (A B)

53. (A B C)

54. (A . B)

55. (A)

56. (NIL . A)

57. ((A) B C)

58. ((A B) C D)

59. ((A B) . C)

60. ((A . B) (C . D) (E . F))

61. ((A B))

62. ((A))

63. (T F T F)

64. T

65. car[x]

66. cadr[x]

67. cdr[x]

68. caddr[x]

69. cadddr[x]

70. caar[x]

71. cdar[x]

72. caaadr[x]

73. caadr[x]

74. caaaar[x]

75. F

76. T

77. T

78. F

79. T

80. T

81. (A B C)

82. ((A) (B) (C))

83. T

84. F

85. cons[A;B]

86. list[X;Y]

87. list[list[A;B];list[C;D]]

88. list[cons[A;B];cons[C;D]]

89. list[list[list[A]]]

90. list[cons[B;C]]

91. list[list[ ];list[ ]]

92. list[A;B;C;D]

93. F

94. T

95. F

96. T

97. A

98. B

99. C

100. (C)

101. (B C)

102. B

103. C

104. NIL

105. undefined

106. T

107. B

108. C

109. F

110. T

111. (B . C)

112. T

113. T

114. F

115. undefined

116. A

117. T

118. undefined

119. A

120. B

121. undefined

122. A

123. NIL

124. T

125. (AB . C)

126. B

127. A

128. A

129. B

130. ((AB . C))

131. T

132. undefined

133. C

134. A

135. (A)

136. A

137. A

138. A

139. A

140. B

141. B

142. (C . B)

143. A

144. S

145. (A . C)

146. (A . B)

147. (A . C)

148. T

149. F

150. F

151. F

152. T

153. T

154. jj[A] = A

155. jj[(A . B)] = jj[B] = B

156. jj[((X . Y) . (X . Z))] =jj[(X . Z)] = jj[Z} =

157. jj[(A B C)] = jj[(B C)] =JJ[(C)] = jj[NIL] = NI

158. jj[(A (C . E))]= jj[((C . E))] = jj[NIL] = NIL

159. X

160. E

161. NO

162. A

```
163.  (B . A)
164.  (C . (B . A))
165.  (((NIL . C) .B) . A)
166.  (NIL . (B . A))
167.  member = λ[[x;m];[null[m]→ F;eq[x;car[m]]→T;T → member[x;cdr[m]]]]
168.  pair = λ[[m;n];[null[m]→ NIL,T→cons[cons[car[m];car[n]];pair[cdr[m];

              cdr[n]]]]]
169.  append = λ[[m;n];[null[m]→ n;T→cons[car[m];append[cdr[m];n]]]]
170.  delete = λ[[x;m];[null[m]→NIL;eq[x;car[m]]→cdr[m];
              T→cons[car[m];delete[x;cdr[m]]]]]
171.  reverse =  [[m];[null[m]→NIL;T→ append[reverse[cdr[m]];list[car[m]]]]]
172.  union = λ[[m;n];[null[m]→n;member[car[m];n]→ union[cdr[m];n];
             T→cons[car[m];union[cdr[m];n]]]]
173.  intersection = λ[[m;n];[null[m]→ NIL;member[car[m];n] →
                      cons[car[m];intersection[cdr[m];n]];T→intersection[cdr[m];
                      n]]]
174.  last = λ[[m];[null[cdr[m]]→car[m];T→last[cdr[m]]]]
175.  superreverse = λ[[m];[null[m]→NIL;T→append[superreverse[cdr[m]];
                      list[superreverse[car[m]]]]]]
176.  part =λ[[x;s];[atom[s]→eq[x;s];part[x;car[s]]→T;T→ part[x;cdr[s]]]]
177.  subst =λ[[x;y;s];[atom[s]→[eq[y;s]→x;T→s];T→cons[subst[x;y;
              car[a]];subst[x;y;cdr[s]]

178.  listofatoms =λ[[s];[atom[s]→list[s];T→union[listofatoms[car[s]];
                    listofatoms[cdr[s]]]]]


179.  (PLUS ZERO ONE)
180.  (PLUS (TIMES X ZERO) (TIMES A ONE))
181.  (PLUS (TIMES X ZERO) (TIMES THREE ZERO))
182.  (PLUS ONE ONE)
```

```
183.  (PLUS (TIMES (TIMES Z Z) ZERO) (TIMES TWO (PLUS (TIMES Z ONE)
               (TIMES Z ONE))))

184.  eq[car[e];RECIP]→list[TIMES;list[RECIP;list[TIMES;cadr[e];cadr[e]]];
         diff[cadr[e];x]]

185.  eq[car[e];SIN]→list[TIMES;list[COS;cadr[e]];diff[cadr[e];x]]

186.  eq[car[e];COS]→list[MINUS;list[TIMES;list[SIN;cadr[e]];diff[cadr[e];x]]]

187.  A

188.  X

189.  (QUOTE A)

190.  (QUOTE T)

191.  (QUOTE NIL)

192.  (QUOTE ((A B )))

193.  (QUOTE QUOTE)

194.  (QUOTE (QUOTE A))

195.  CAR

196.  (CAR X)

197.  (CAR (QUOTE A))

198.  (ATOM X)

199.  (FF X)

200.  (FF (CAR X))

201.  (COND ((ATOM X) X) (T (FF (CAR X))))

202.  (LAMBDA (X) X)

203.  (LAMBDA (X) (CAR X))

204.  (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X)))))

205.  (LABEL FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X))))))

206.  (LAMBDA (X) ((LAMBDA (Y) (CAR Y)) X))
```

207. ((Y . B))
208. ((X . A))
209. ((X . A) (Y . B) (Z . C))
210. (Y . B)
211. (X . A)
212. (Z . (M N))
213. (Y . T)
214. A
215. T
216. M
217. (M)
218. (M T (M N))
219. M
220. M
221. A
222. (A . B)
223. ((A) . (B)) =((A) B)
224. undefined
225. T
226. M
227. M
228. A
229. A
230. A
231. A
232. A
233. Q

CS-TR Scanning Project
Document Control Form

Date : 11 / 30 / 95

Report # AIM - 64

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other: ______

Document Information Number of pages: 27 (31-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:	Intended to be printed as :
☒ Single-sided or	☒ Single-sided or
☐ Double-sided	☐ Double-sided

Print type:

☐ Typewriter	☐ Offset Press	☐ Laser Print
☐ InkJet Printer	☐ Unknown	☒ Other: COPY OF MEMEOGRAPH

Check each if included with document:

☐ DOD Form	☐ Funding Agent Form	☐ Cover Page
☐ Spine	☐ Printers Notes	☐ Photo negatives
☐ Other: ______		

Page Data:

Blank Pages(by page number): ______

Photographs/Tonal Material (by page number): ______

Other (note description/page number):

Description :	Page Number:
IMAGE MAP! (1-27) UN#'ED TITLE PAGE, 2-27	
(28-31) SCANCONTROL, TRGT'S (3)	

Scanning Agent Signoff:
Date Received: 11 / 30 / 95 Date Scanned: 12 / 6 / 95 Date Returned: 12 / 7 / 95

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd