The views, conclusions, or recommendations expressed in this document do not necessarily reflect the official views or policies of agencies of the United States Government.

The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004 with the System Development Corporation.

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406
Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM- 2710/230/00

AUTHOR
S. L. Kameny, SDC

TECHNICAL
E. Book
E. Book

RELEASE
J. Barnett
J. Barnett, SDC
L. Hawkinson, III
for J. I. Schwartz

DATE 12/12/66 PAGE 1 OF 22 PAGES

LISP 2 Source Language Syntax

Specifications for Syntax Translator

ABSTRACT

This document presents a set of syntax equations which define the syntax of LISP 2 Source Language (SL) and its transformation to Intermediate Language (IL).

It is intended to complement TM-2710/210/00, "Syntax of LISP 2 Tokens,"and TM-2710/220/00, "LISP 2 Intermediate Language."

These representations are the current state of the LISP 2 languages. However, an effort of design and and specification will continue through March 1967. Therefore, these equations are expected to be changed and updated throughout this period.

CONTENTS

INTRODUCTION

This document presents a set of syntax equations which define the syntax of LISP 2 Source Language (SL) and its transformation to Intermediate Language (IL).

An equation which defines such a transformation contains the sign SL:, a meta-linguistic variable, the sign =, the SL definition, the sign ||, the sign IL:, the IL definition, and, finally the sign ||. In such equations, the terms in the SL definition and the IL transformation are normally matched in an obvious way; otherwise, an explanation is given following the equation. Other equations apply to source language only, IL only, or are definitions applicable to both. They start with the signs (S), IL:, or SL or IL:, respectively.

The following terms, not defined in this document, are defined in TM-2710/210/00,"Syntax of LISP 2 Tokens,"and represent an interface between the syntax translator and a token-maker:

array
car:cdr:op
dotted:literal
gen:name
identifier
literal
number
operator
string
string:name
u:mark

1. CONSTANTS

```
SL or IL:  constant  =  simple:datum | quoted:expression  ||

SL or IL:  simple:datum  =  boolean | number | array | string |
                            functional:constant  ||

SL or IL:  boolean  =  TRUE | false  ||

SL or IL:  false  =  FALSE | NIL | ()  ||

SL or IL:  number  =  integer | octal | real  ||

SL or IL:  array  =  boolean:array | integer:array | octal:array |
                     real:array | symbol:array | functional:array  ||
```

Arrays and functional:constants have character representations beginning with a left bracket [character. On recognizing a left bracket token, the syntax translator can obtain the desired simple:datum by calling the function ARREAD ().

```
SL:  quoted:expression  =  ' s:expression | QUOTE (expression)  ||
     IL:  (QUOTE  s:expression) | (QUOTE  expression)  ||
```

On recognizing a prime ' token, the syntax translator can obtain the s:expression by calling READ (), and no translation of the s:expression occurs. The SL form with QUOTE causes translation to occur, because QUOTE is not recognized, and QUOTE (expression) is handled as a name:expression.

```
                                                            *
SL or IL#: s:expression  =  atom | (s:expression  s:expression
                             { . s:expression | empty})  ||

SL or IL:  atom  =  simple:datum | identifier  ||

SL or IL:  identifier  =  literal | dotted:literal | gen:name | string:name |
                          operator | u:mark  ||
```

The primitive terms used on this page are not further defined here, but are given in TM-2710/210/00, Syntax of LISP 2 Tokens, and represent an interface between the syntax translator and a token-maker.

# An asterisk * used as an exponent in a syntax equation means 0 or more of the preceding syntactic entity. The exponent *+1 means 1 or more.

2. VARIABLES, TYPES, AND MODES

The terms described here occur in too many places in the syntax equations to permit an orderly exposition.

```
SL or IL:  variable  =  tailed:variable |  untailed:variable  ||

SL:  tailed:variable  =  identifier $ {identifier | $}  ||
       IL:  (identifier . {identifier | LISP})  ||

SL:  untailed:variable  =  unreserved:name  ||
       IL:  identifier  ||

SL:  name  =  literal | dotted:literal | gen:name | string:name  ||
```

Unreserved:name is a name that is not a member of the set of reserved words given in Table 1.

```
SL or IL:  type  =  simple:type | array:type | functional:type  ||

SL or IL:  simple:type  =  BOOLEAN | INTEGER | OCTAL | REAL | SYMBOL  ||

SL:  array:type  =  f:type  ARRAY ||
       IL:  (ARRAY  f:type)  ||

SL or IL:  f:type  =  FUNCTIONAL | simple:type  ||

SL:  functional:type  =  value:type  FUNCTIONAL  ({indef:par:type |
                          parameter:type  {{ , parameter:type}* |
                          , indef:par:type} | empty})  ||
       IL:  (FUNCTIONAL  value:type  parameter:type*
            indef:par:type | empty})  ||

SL:  value:type  =  NOVALUE | f:type | empty ||
       IL:  NOVALUE | f:type | NIL  ||
```

Table 1. Reserved Words of Source Language

AND
ARRAY
ATOM
BLOCK
DECLARE
DEFAULT
DO
ELSE
END
FLUID
FOR
FREE
FUNARG
FUNCTION
FUNCTIONAL
GO
IF
IN
INSTRUCTIONS
LEXICAL
LOOP
MACRO
NOT
NULL
ON
OR
OWN
PROP
RESET
RETURN
ROUTINE
SECTION
STEP
THEN
UNLESS
UNTIL
WHILE

```
SL:  parameter:type  =  f:type  transmission:mode  ||
        IL:  (f:type  loc) | f:type  ||

SL:  transmission:mode  = ↞ |  empty  ||
        IL:  loc | empty  ||

IL#:  loc  =  LOC | ↞  ||


SL:  indef:par:type  =  f:type ()  transmission mode  ||
        IL:  ((f:type) loc) | (f:type)  ||

SL or IL:  type:option  =  type | empty  ||

SL or IL:  free:storage:mode  =  OWN | storage:mode  ||

SL or IL:  storage:mode  =  FLUID | FREE | empty  ||

SL or IL:  param:storage:mode  =  LEXICAL | storage:mode  ||
```

# The translation from ↞ to LOC is required for a LISP 2 system that has been derived by bootstrapping from LISP 1.5 (i.e., the Q-32 LISP 2).

3. TOP LEVEL OPERATIONS, AND DECLARATIVES

```
SL:  operative:file  =  END ; |
                        {file:name :  | empty}
                        {operation ,  | section:declaration |
                        default:declaration | free:declaration}*
                        {operation | empty} ;  ||

       IL:  STOP | (file:name  operation*)  ||
```

```
SL or IL:  file:name  =  identifier | number  ||
```

If no file:name is supplied in SL, the syntax translator calls GENID () to supply one.

```
SL or IL:  operation  =  declarative | expression  ||

SL or IL:  declarative  =  section:declaration | default:declaration |
                           free:declaration | function:definition |
                           dummy:function:declaration |
                           routine:definition | dummy:routine:declaration |
                           macro:definition | instructions:definition |
                           lap:definition  ||
```

3.1 SECTION AND DEFAULT DECLARATIONS

```
SL:  section:declaration  =  default:type  SECTION  section:list  ||
       IL:  (SECTION  section:list  default:type)  ||

SL:  default:type  =  f:type | empty  ||

SL:  section:list  =  section:name  { ,  section:name}*  ||
       IL:  section:name | (section:name*)  ||
```

For section:list in IL, (section:name) is equivalent to section:name.

```
SL or IL:  section:name  =  identifier  ||

SL:  default:declaration  =  DEFAULT  f:type :  ||
       IL:  (DEFAULT  f:type)  ||
```

3.2 FREE:DECLARATIONS

```
SL:  free:declaration  =  DECLARE (var:preset:list)  free:declaration list :  ||
        IL:  (DECLARE  free:variable:declaration*) ||

(S)  var:preset:list  =  var:preset  { ,var: preset}*  ||

(S)  var:preset  =  variable  { ← expression | ↞  name:expression | empty}  ||

(S)  free:declaration:list  =  free:declaration:fragment
                               { ; free:declaration:fragment}*  ||

(S)  free:declaration:fragment  =  {type  free:storage:mode |
                                   {OWN | FLUID | FREE}  type:option}
                                    variable  { , variable}*  ||

IL:  free:variable:declaration  =  variable | (variable  type:option
                                   free:storage:mode) |free:var:preset:decl |
                                   synonym:declaration  ||

IL#:  free:var:preset:declaration  =  (variable  type:option
                                      free:storage:mode
                                      {expression | LOC  full:locative}) |
                                      (variable  { ← expression |
                                      ↞ name:expression}  type:option
                                      free:storage:mode)  ||



SL:  synonym:declaration  =  variable  MEANS  {tailed:variable | name}  ||
        IL:  (variable  MEANS  {tailed:variable | name})  ||
```

# The first form given here agrees with IL for Q-32 LISP 2. The second form applies to all later versions of LISP 2.

3.3 DECLARATIONS OF FUNCTIONS, ROUTINE, AND MACROS

```
SL:  function:definition  =  dummy:function:declaration  expression  ||
       IL:  (function:heading  expression)  ||

SL:  dummy:function:declaration  =  value:type  FUNCTION  variable (param:list)
                                    param:decl:list :  ||
       IL:  (function:heading)  ||

IL:  function:heading  =  FUNCTION  {variable | (variable  value:type)}
                          parameter:list  ||

(S)  param:list  =  indef:param | param  { , param}*
                    { , indef:param | empty} | empty  ||

(S)  indef:param  =  variable  (variable)  transmission:mode  ||

(S)  param  =  variable  transmission:mode  ||

(S)  param:decl:list  =  param:decl:fragment
                         { ; param:decl:fragment}* | empty  ||

(S)  param:decl:fragment  =  {type  param:storage:mode |
                             { FLUID | FREE | LEXICAL | type:option}
                              variable  { , variable}*  ||

IL:  parameter:list  =  (parameter*  indef:parameter) ||

IL:  parameter  =  variable | (variable  type:option  param:stor:mode
                   transmission:mode)  ||

IL#:  indef:parameter  =  (variable  type:option  transmission:mode
                          INDEF  variable) |
                          ((variable variable)  type:option  transmission mode) ||
```

# The first IL syntax for indef:parameter is the one accepted by the first Q-32 LISP 2. Later versions of LISP 2 accept the second form of indef:parameter. Indef:parameters must use lexical variables. Specifying FLUID or FREE for an indef:parameter variable in SL will cause a syntax translator error.

The variables appear in the parameter:list in IL in the order of their occurrence in the param:list in SL.

The information in a param or an indef:param is obtained by merging any attributes of that variable found in the param:decl:list with those found in the param:list.

The syntax translator must specifically recognize the words FREE, FLUID, LEXICAL and the operator ↞ to prepare the parameter:list in the prescribed order.

```
SL:  routine:definition  =  dummy:routine:declaration  expression  ||

        IL:  (routine:heading  expression)  ||

SL:  dummy:routine:declaration  =  value:type  ROUTINE  variable  (param:list)
                                   param:decl:list  :  ||

        IL:  (routine:heading)  ||

IL:  routine:heading  =  ROUTINE  {variable | (variable  value:type)}
                         parameter:list  ||

SL:  macro:definition  =  MACRO  {tailed:variable | identifier}   (variable) :
                          expression  ||

        IL:  (MACRO  variable  (variable)  expression)  ||
```

Any variable may be used to name a macro, instructions, or lap-defined function in either SL or IL, and the usual restrictions on reserved names do not apply.

```
SL:  instructions:definition  =  INSTRUCTIONS  {tailed:variable | identifier}
                                 ()  expression  ||

        IL:  (INSTRUCTIONS  variable  ()  expression)  ||

SL:  lap:definition  =  LAP (d:list  listing  section:name)  ||

        IL:  (LAP  d:list  listing  section:name)  ||
```

The syntax translator does nothing with lap definitions except to place the word LAP inside of the untranslated list of arguments.

4. EXPRESSIONS

```
SL or IL:  expression  =  conditional:expression | unconditional:expression  ||

SL:  unconditional:expression  =  basic:expression | simple:expression |
                                  (expression)  ||
        IL:  basic:expression | simple:expression | expression  ||
```

4.1 SIMPLE EXPRESSIONS--LOGICAL AND ARITHMETIC INFIX OPERATORS

```
SL:  simple:expression  =  conjunction  {OR  conjunction}*  ||
       IL:  conjunction | (OR conjunction conjunction*)  ||

SL:  conjunction  =  negation  {AND negation}*  ||
       IL:  negation | (AND negation negation*) ||

SL:  negation  =  relation |  boolean:unary:op  negation   ||
       IL:  relation | (boolean:unary:op  negation)  ||

SL or IL:  boolean:unary:op  =  NOT | NULL |ATOM | ...  ||
```

The class of boolean unary operators is in principle open, and can admit any binary operator whose single argument is symbolic.

```
SL:  relation  =  construct  {rel:op  construct}*  ||
       IL:  construct | (RELATION construct  {rel:op  construct}*)  ||

SL or IL#:  rel:op  =  < | <= | > | >= | = | /= | ...  ||
```

The class of binary relational operators is in principle open, and can admit any binary relational operator whose arguments are symbolic.

```
SL:  construct  =  sum  { . sum}*  ||
       IL:  sum | (CONS  sum  sum*+1)  ||
```

# See Table 2.

Table 2. Operator Transformations for Q-32 LISP 2

For the first Q-32 LISP 2, the syntax translator must translate some of the SL infix operators into literals in IL, as given in the following table :

Operator	Literal	Operator	Literal
<	LS	*	TIMES
<=	LQ	/	RECIP
>	GR	\	REMAINDER
>=	GQ	//	IQUOTIENT
=	EQ	↑	EXPT
/=	NQ	←	SET
+	PLUS	↞	LOCSET (in locative assignment)
-	MINUS	↞	LOC (in declarations)

```
SL:  sum  =  term  {{+ | -}  term}*   ||
        IL#:  term | (+ term  {term | (- term)}*+1)  ||

SL:  term  =  / factor | factor  {{* | /}  factor}*  ||
        IL#:  (/ factor) | (* factor  {factor | (/ factor)}*+1)  ||

SL:  factor  =  part | factor  { \| //}  part  ||
        IL#:  part | ({ \ | //}  factor  part)  ||

SL:  part  =  {- | +} part | primary  { ↑ part | empty}  ||
        IL#:   (- part) | part | ( ↑ primary  part) | primary  ||

SL:  primary  =  basic:expression | conditional:expression | (expression)  ||
        IL:  basic:expression | conditional:expression | expression  ||
```

4.2 CONDITIONAL EXPRESSIONS

```
SL:  conditional:expression  =  closed:conditional:expression |
                                open:conditional:expression  ||
        IL:  closed:conditional:expression | open:conditional:expression)  ||

SL:  closed:conditional:expression  =  open:conditional:expression  ELSE
                                       unconditional:expression  ||
        IL:  open:conditional:expression  unconditional:expression )  ||

SL:  open:conditional:expression  =  if:clause  closed:expression
                                     {{ELSE | empty}  if:clause
                                     closed:expression}*  ||
        IL:  (IF  {if:clause  closed:expression}*+1  ||

SL:  if:clause  =  IF  expression  THEN  ||
        IL:  expression  ||

SL or IL:  closed:expression  =  closed:conditional:expression |
                                 unconditional:expression  ||
```

# See Table 2.

4.3 BASIC EXPRESSIONS

```
SL or IL:  basic:expression  =  block | compound | function:definition |
                                funarg | assignment:expression | locative |
                                constant ||

SL:  funarg  =  value:type  FUNARG  (param:list)  param:decl:list : expression ||
     IL:  (FUNARG  value:type  parameter:list  expression) ||

SL:  assignment:expression  =  locative ← expression ||
     IL#:  ( ← locative  expression) ||

SL or IL:  locative  =  symbolic:expression | locative:assignment ||

SL:  symbolic:expression  =  name:expression | unary:symbolic:op
                             symbolic:expression ||
     IL:  name:expression | (unary:symbolic:op  symbolic:expression) ||

SL:  locative:assignment  =  variable ↞ name:expression ||
     IL#:  ( ↞  variable  name:expression) ||

SL:  name:expression  =  variable {() | (expression  {, expression}*) |
                         empty} ||
     IL:  (variable expression*) | variable ||
```

Strictly speaking, locative:assignment requires full:locative instead of name:expression. However, to tell whether a name:expression is a full:locative in most instances requires semantics as well as syntax, so it is not particularly useful to make any check at the syntax translator level.

# See Table 2.

4.4 COMPOUNDS, BLOCKS, AND FUNARG

```
SL:  compound  =  DO  {label : | statement ;}*
                      {label : | statement | empty } END  ||
        IL:  (BLOCK () {label | statement}*)  ||

SL:  block  =  BLOCK  (var:preset:list)  declaration:list :
                 {label : | statement ;}*  {label : | statement | empty}  END  ||
        IL:  (BLOCK  variable:list  {label | statement}*)  ||

SL:  label  =  unreserved:name ||
        IL:  identifier  ||
```

The use of operators or reserved names as labels is permissible in IL, but interferes with the syntax of SL. To introduce an arbitrary identifier as a label can be done if LABEL (identifier, statement) is allowed as a statement, being converted to (LABEL identifier statement) which is correct in IL.

```
(S)  declaration:list  =  declaration:fragment  { ; declaration:fragment}*  ||

(S)  declaration:fragment  =  {type  param:storage:mode |
                               { FLUID | FREE | LEXICAL}  type:option}
                                variable  { , variable}*  ||

IL:  variable:list  =  (block:variable:declaration*)

IL:  block:variable:declaration  =  variable |
                                    (variable  type:option  param:storage:mode) |
                                    variable:preset:declaration    ||
IL#:  variable:preset:declaration  =  (variable type:option
                                      param:storage:mode
                                      {expression | LOC  name:expression}) |
                                      (variable  { ← expression |
                                      ↞ name:expression}  type:option
                                      param:storage:mode)    ||
```

The variables appear in the variable:list in IL in the same order as they are given in the var:preset:list in SL. The information in a block:variable: :declaration is obtained by merging any attributes of that variable found in the declaration:list with the preset information found in the var:preset:list.

The syntax translator must specifically recognize the words FREE, FLUID, LEXICAL and the operators ← and ↞ to prepare the variable:list in the desired order.

# The first form given here agrees with IL for Q-32 LISP 2. The second form applies to all later versions of LISP 2.

5. STATEMENTS

```
SL:  statement  =  conditional:statement | unconditional:statement  ||

SL:  unconditional:statement  =  compound | block:statement | go:statement |
                                 case:statement | return:statement |
                                 unconditional:expression | (statement)  ||

       IL:  compound | block:statement | go:statement | case:statement |
            return:statement | unconditional:expression | statement   ||
```

An unconditional:expression can be a statement only if it is not a constant or a variable. In IL, an identifier in statement context is a label.

5.1 GO, RETURN, AND CASE STATEMENT

```
SL:  go:statement  =  GO  label  ||
       IL:  (GO  identifier  ||
```

If LABEL (identifier, statement) is introduced into SL, then go:statement becomes GO identifier in SL.

```
SL:  return:statement  =  RETURN  expression  ||
       IL:  (RETURN  expression)  ||

SL:  case:statement  =  CASE (expression  { , labelled:statement}*)  ||
       IL:  (CASE  expression  labelled:statement*)  ||

SL:  labelled:statement  =  statement | label : labelled:statement ||
       IL:  statement | (LABEL  label  labelled:statement)  ||

SL:  code:statement  =  CODE (item*)  ||
       IL:  (CODE  item*)  ||
```

Except for the placing of the word CODE inside of the parentheses, no translation of code:statement is done by the syntax translator.

```
SL:  for:statement  =  FOR  locative  loop:control  while:phrase
                       unless:phrase :  simple:statement  ||

        IL:  (FOR  locative  loop:control  while:phrase  unless:phrase
             simple:statement)  ||
```

```
SL:  loop:control  =  {← expression | empty}  {RESET  expression |
                      STEP  expression  {rel:op  expression | empty}}  |
                      {IN | ON | LOOP}  expression | empty  ||

        IL:  (RESET   {expression | locative}  expression) |
             (STEP  {expression | locative}  expression
             {rel:op  expression | empty})
             ({IN | ON | LOOP}  expression) | ()   ||
```

The expression or locative that immediately precedes STEP or RESET in SL always follows the word STEP or RESET in IL; hence the empty initialization of a STEP or RESET for:element in SL causes the locative portion of a for:statement to appear in two places in IL.

```
SL:  while:phrase  =  WHILE  expression | empty  ||

        IL:  (WHILE  expression) | empty ||
```

```
SL:  unless:phrase  =  UNLESS  expression | empty ||

        IL:  (UNLESS  expression) | empty ||
```

5.2 CONDITIONAL STATEMENT

```
SL:  conditional:statement  =  closed:conditional:statement  |
                               open:conditional:statement    ||

        IL:  closed:conditional:statement  |
             open:conditional:statement )  ||
```

```
SL:  closed:conditional:statement  =  open:conditional:statement
                                      ELSE  labelled:unconditional:statement ||

        IL:  open:conditional:statement  labelled:unconditional:statement )  ||
SL:  labelled:unconditional:statement  =  unconditional:statement |
                                          label : labelled:unconditional:statement  ||

        IL:  unconditional:statement |
             (LABEL  label  labelled:unconditional:statement)  ||

SL:  open:conditional:statement  =  if:clause  simple:statement
                                    {{ELSE | empty}  labelled:if:clause
                                    simple:statement}*  ||

        IL:  (IF  if:clause  simple:statement  {labelled:if:clause
             simple:statement}*  ||

SL:  labelled:if:clause  =  if:clause | label : labelled:if:clause ||
        IL:  if:clause |  (LABEL  label  labelled:if:clause)  ||

SL:  simple:statement  =  unconditional:statement | closed:conditional:statement |
                          label : simple:statement  ||

        IL:  unconditional:statement | closed:conditional:statement |
             (LABEL  label  simple:statement)  ||
```

5.3 BLOCK STATEMENT

```
SL or IL:  block:statement  =  try:statement | code:statement | for:statement |
                               block
SL:  try:statement  =  TRY (locative  label  labelled:statement)  ||
        IL:  (TRY  locative  label  labelled:statement)  ||
```

INDEX

INDEX (contd.)