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 specification will continue through March 1967. Therefore, these equations are expected to be changed and updated throughout this period.

Copyright System Development Corporation and Information International, Incorporated 1966
CONTENTS

INTRODUCTION ............................................. 3

Chapter 1.  
CONSTANTS .................................................. 4

2.  VARIABLES, TYPES, AND MODES ...................... 5

3.  TOP LEVEL OPERATIONS, AND DECLARATIVES ........ 8
   3.1 SECTION AND DEFAULT DECLARATIONS ............... 8
   3.2 FREE:DECLARATIONS .................................. 9
   3.3 DECLARATIONS OF FUNCTIONS, ROUTINES, AND MACROS 10

4.  EXPRESSIONS ............................................ 12
   4.1 SIMPLE EXPRESSIONS--LOGICAL AND ARITHMETIC  
       INFIX OPERATORS .................................. 12
   4.2 CONDITIONAL EXPRESSIONS ......................... 14
   4.3 BASIC EXPRESSIONS .................................. 15
   4.4 COMPOUNDS, BLOCKS, AND FUNARG ................. 16

5.  STATEMENTS ............................................. 18
   5.1 GO, RETURN, AND CASE STATEMENT ................. 18
   5.2 CONDITIONAL STATEMENT ......................... 19
   5.3 BLOCK STATEMENT .................................. 20

INDEX ..................................................... 21

Table 1.  Reserved Words of Source Language ........... 6
Table 2.  Operator Transformations for Q-32 LISP 2 .... 13
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  identifier  string
(car:cdr:op)  literal  string:name
(dotted:literal)  number  u:mark
(gen:name)  operator
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 ||


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 the function 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.1
   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

<table>
<thead>
<tr>
<th>词</th>
<th>定义</th>
<th>语法</th>
<th>语义</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>FUNARG</td>
<td>ON</td>
<td></td>
</tr>
<tr>
<td>ARRAY</td>
<td>FUNCTION</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>ATOM</td>
<td>FUNCTIONAL</td>
<td>OWN</td>
<td></td>
</tr>
<tr>
<td>BLOCK</td>
<td>GO</td>
<td>PROP</td>
<td></td>
</tr>
<tr>
<td>DECLARE</td>
<td>IF</td>
<td>RESET</td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td>IN</td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>DO</td>
<td>INSTRUCTIONS</td>
<td>ROUTINE</td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>LEXICAL</td>
<td>SECTION</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>LOOP</td>
<td>STEP</td>
<td></td>
</tr>
<tr>
<td>FLUID</td>
<td>MACRO</td>
<td>THEN</td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td>NOT</td>
<td>UNLESS</td>
<td></td>
</tr>
<tr>
<td>FREE</td>
<td>NULL</td>
<td>UNTIL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WHILE</td>
<td></td>
</tr>
</tbody>
</table>
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:var:pres:declaration = (variable type:option
    free:storage:mode) | free:var:pres:decl |
    synonym:declaration ||

IL#: free:var:pres: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 = valu::type FUNCTION variable (param:list)
    param:decl:list:
    IL: (function:heading) ||

IL: function:heading = FUNCTION (variable | (variable valu::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:storage: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 $\leftrightarrow$ 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) ||
    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.
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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Literal</th>
<th>Operator</th>
<th>Literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>LS</td>
<td>*</td>
<td>TIMES</td>
</tr>
<tr>
<td>&lt;=</td>
<td>LQ</td>
<td>/</td>
<td>RECIP</td>
</tr>
<tr>
<td>&gt;</td>
<td>GR</td>
<td>\</td>
<td>REMAINDER</td>
</tr>
<tr>
<td>&gt;=</td>
<td>GQ</td>
<td>//</td>
<td>IQUOTIENT</td>
</tr>
<tr>
<td>=</td>
<td>EQ</td>
<td>+</td>
<td>EXPT</td>
</tr>
<tr>
<td>/=</td>
<td>NQ</td>
<td>+</td>
<td>SET</td>
</tr>
<tr>
<td>+</td>
<td>PLUS</td>
<td>↔</td>
<td>LOCSET (in locative assignment)</td>
</tr>
<tr>
<td>-</td>
<td>MINUS</td>
<td>↔</td>
<td>LOC (in declarations)</td>
</tr>
</tbody>
</table>
SL: \[ \text{sum} = \text{term} \{ (+ | - ) \text{ term} \}^* \]
    \[ \quad \text{IL#}: \quad \text{term} \mid ( + \text{ term} \{ \text{term} \mid ( - \text{ term}) \}^+)^+ \]

SL: \[ \text{term} = / \text{ factor} \mid \text{ factor} \{ (\ast | /) \text{ factor} \}^+ \]
    \[ \quad \text{IL#}: \quad ( / \text{ factor} ) \mid ( \ast \text{ factor} \{ \text{factor} \mid ( / \text{ factor}) \}^+) \]

SL: \[ \text{factor} = \text{ part} \mid \text{ factor} \{ \backslash | // \} \text{ part} \]
    \[ \quad \text{IL#}: \quad \text{part} \mid ( ( \backslash | // ) \text{ factor part} ) \]

SL: \[ \text{part} = (- | + ) \text{ part} \mid \text{ primary} \{ + \text{ part} \mid \text{ empty} \} \]
    \[ \quad \text{IL#}: \quad ( - \text{ part} ) \mid \text{ part} \mid ( + \text{ primary part} ) \mid \text{ primary} \]

SL: \[ \text{primary} = \text{ basic: expression} \mid \text{ conditional: expression} \mid ( \text{ expression} ) \]
    \[ \quad \text{IL}: \quad \text{basic: expression} \mid \text{ conditional: expression} \mid \text{ expression} \]

4.2 \quad \text{CONDITIONAL EXPRESSIONS}

SL: \[ \text{conditional: expression} = \text{ closed: conditional: expression} \mid \text{ open: conditional: expression} \]
    \[ \quad \text{IL}: \quad \text{closed: conditional: expression} \mid \text{ open: conditional: expression} \]

SL: \[ \text{closed: conditional: expression} = \text{ open: conditional: expression} \text{ ELSE} \text{ unconditional: expression} \]
    \[ \quad \text{IL}: \quad \text{open: conditional: expression} \text{ unconditional: expression} ) \]

SL: \[ \text{open: conditional: expression} = \text{ if: clause closed: expression} \]
    \[ \quad \{ (\text{ELSE} | \text{ empty}) \text{ if: clause} \text{ closed: expression} \}^+ \]
    \[ \quad \text{IL}: \quad \text{(IF} \{ \text{if: clause closed: expression}\}^+ \text{) ELSE} \text{empty} \]

SL: \[ \text{if: clause} = \text{ IF} \text{ expression} \text{ THEN} \]
    \[ \quad \text{IL}: \quad \text{expression} \]

SL or IL: \[ \text{closed: expression} = \text{ closed: conditional: expression} \mid \text{ unconditional: expression} \]

# \quad \text{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:
description 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. \underline{STATEMENTS}

SL: statement = conditional:statement | unconditional: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 \underline{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.
    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 ) ||

  ELSE labelled:unconditional:statement ||
SL: labelled:unconditional:statement = unconditional:statement |
     label : labelled:unconditional:statement |

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

     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

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>4</td>
</tr>
<tr>
<td>array:type</td>
<td>4</td>
</tr>
<tr>
<td>assignment:expression</td>
<td>15</td>
</tr>
<tr>
<td>atom</td>
<td>4</td>
</tr>
<tr>
<td>basic:expression</td>
<td>15</td>
</tr>
<tr>
<td>block</td>
<td>16</td>
</tr>
<tr>
<td>block:statement</td>
<td>20</td>
</tr>
<tr>
<td>block:variable:declaration</td>
<td>16</td>
</tr>
<tr>
<td>boolean</td>
<td>4</td>
</tr>
<tr>
<td>boolean:unary:op</td>
<td>12</td>
</tr>
<tr>
<td>case:statement</td>
<td>18</td>
</tr>
<tr>
<td>closed:conditional:expression</td>
<td>14</td>
</tr>
<tr>
<td>closed:conditional:statement</td>
<td>19</td>
</tr>
<tr>
<td>closed:expression</td>
<td>14</td>
</tr>
<tr>
<td>code:statement</td>
<td>18</td>
</tr>
<tr>
<td>compound</td>
<td>16</td>
</tr>
<tr>
<td>conditional:expression</td>
<td>14</td>
</tr>
<tr>
<td>conditional:statement</td>
<td>19</td>
</tr>
<tr>
<td>conjunction</td>
<td>12</td>
</tr>
<tr>
<td>constant</td>
<td>4</td>
</tr>
<tr>
<td>construct</td>
<td>12</td>
</tr>
<tr>
<td>declaration:fragment</td>
<td>16</td>
</tr>
<tr>
<td>declaration:list</td>
<td>16</td>
</tr>
<tr>
<td>declarative</td>
<td>8</td>
</tr>
<tr>
<td>default:declaration</td>
<td>8</td>
</tr>
<tr>
<td>default:type</td>
<td>8</td>
</tr>
<tr>
<td>dummy:function:declaration</td>
<td>10</td>
</tr>
<tr>
<td>dummy:routine:declaration</td>
<td>11</td>
</tr>
<tr>
<td>expression</td>
<td>12</td>
</tr>
<tr>
<td>f:type</td>
<td>5</td>
</tr>
<tr>
<td>factor</td>
<td>14</td>
</tr>
<tr>
<td>false</td>
<td>4</td>
</tr>
<tr>
<td>file:name</td>
<td>8</td>
</tr>
<tr>
<td>for:statement</td>
<td>19</td>
</tr>
<tr>
<td>free:declaration</td>
<td>9</td>
</tr>
<tr>
<td>free:declaration:fragment</td>
<td>9</td>
</tr>
<tr>
<td>free:declaration:list</td>
<td>9</td>
</tr>
<tr>
<td>free:storage:mode</td>
<td>7</td>
</tr>
<tr>
<td>free:variable:declaration</td>
<td>9</td>
</tr>
<tr>
<td>funarg</td>
<td>15</td>
</tr>
<tr>
<td>function:definition</td>
<td>10</td>
</tr>
<tr>
<td>function:heading</td>
<td>10</td>
</tr>
<tr>
<td>functional:type</td>
<td>5</td>
</tr>
<tr>
<td>go:statement</td>
<td>18</td>
</tr>
<tr>
<td>identifier</td>
<td>4</td>
</tr>
<tr>
<td>if:clause</td>
<td>14</td>
</tr>
<tr>
<td>indef:par:type</td>
<td>7</td>
</tr>
<tr>
<td>indef:param</td>
<td>10</td>
</tr>
<tr>
<td>indef:parameter</td>
<td>10</td>
</tr>
<tr>
<td>instructions:definition</td>
<td>11</td>
</tr>
<tr>
<td>label</td>
<td>16</td>
</tr>
<tr>
<td>labelled:if:clause</td>
<td>20</td>
</tr>
<tr>
<td>labelled:statement</td>
<td>18</td>
</tr>
<tr>
<td>labelled:unconditional:statement</td>
<td>20</td>
</tr>
<tr>
<td>lap:definition</td>
<td>11</td>
</tr>
<tr>
<td>loc</td>
<td>7</td>
</tr>
<tr>
<td>locative</td>
<td>15</td>
</tr>
<tr>
<td>locative:assignment</td>
<td>15</td>
</tr>
<tr>
<td>loop:control</td>
<td>19</td>
</tr>
<tr>
<td>Index Entry</td>
<td>Page(s)</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>macro: definition</td>
<td>11</td>
</tr>
<tr>
<td>s: expression</td>
<td>4</td>
</tr>
<tr>
<td>section: declaration</td>
<td>8</td>
</tr>
<tr>
<td>section: list</td>
<td>8</td>
</tr>
<tr>
<td>name</td>
<td>5</td>
</tr>
<tr>
<td>section: name</td>
<td>8</td>
</tr>
<tr>
<td>name: expression</td>
<td>15</td>
</tr>
<tr>
<td>simple: datum</td>
<td>4</td>
</tr>
<tr>
<td>negation</td>
<td>12</td>
</tr>
<tr>
<td>simple: expression</td>
<td>12</td>
</tr>
<tr>
<td>simple: type</td>
<td>5</td>
</tr>
<tr>
<td>simple: statement</td>
<td>20</td>
</tr>
<tr>
<td>statement</td>
<td>18</td>
</tr>
<tr>
<td>open: conditional: expression</td>
<td>14</td>
</tr>
<tr>
<td>storage: mode</td>
<td>7</td>
</tr>
<tr>
<td>open: conditional: statement</td>
<td>20</td>
</tr>
<tr>
<td>sum</td>
<td>14</td>
</tr>
<tr>
<td>operation</td>
<td>8</td>
</tr>
<tr>
<td>symbolic: expression</td>
<td>15</td>
</tr>
<tr>
<td>operative: file</td>
<td>8</td>
</tr>
<tr>
<td>synonym: declaration</td>
<td>9</td>
</tr>
<tr>
<td>param</td>
<td>10</td>
</tr>
<tr>
<td>tailed: variable</td>
<td>5</td>
</tr>
<tr>
<td>param: decl: fragment</td>
<td>10</td>
</tr>
<tr>
<td>term</td>
<td>14</td>
</tr>
<tr>
<td>param: decl: list</td>
<td>10</td>
</tr>
<tr>
<td>transmission: mode</td>
<td>7</td>
</tr>
<tr>
<td>param: list</td>
<td>10</td>
</tr>
<tr>
<td>try: statement</td>
<td>20</td>
</tr>
<tr>
<td>param: storage: mode</td>
<td>7</td>
</tr>
<tr>
<td>type</td>
<td>5</td>
</tr>
<tr>
<td>parameter</td>
<td>10</td>
</tr>
<tr>
<td>type: option</td>
<td>7</td>
</tr>
<tr>
<td>parameter: list</td>
<td>10</td>
</tr>
<tr>
<td>unconditional: expression</td>
<td>12</td>
</tr>
<tr>
<td>parameter: type</td>
<td>7</td>
</tr>
<tr>
<td>unconditional: statement</td>
<td>18</td>
</tr>
<tr>
<td>part</td>
<td>14</td>
</tr>
<tr>
<td>unless: phrase</td>
<td>19</td>
</tr>
<tr>
<td>primary</td>
<td>14</td>
</tr>
<tr>
<td>un tiled: variable</td>
<td>5</td>
</tr>
<tr>
<td>quoted: expression</td>
<td>4</td>
</tr>
<tr>
<td>value: type</td>
<td>5</td>
</tr>
<tr>
<td>rel: op</td>
<td>12</td>
</tr>
<tr>
<td>var: preset</td>
<td>9</td>
</tr>
<tr>
<td>relation</td>
<td>12</td>
</tr>
<tr>
<td>var: preset: list</td>
<td>9</td>
</tr>
<tr>
<td>return: statement</td>
<td>18</td>
</tr>
<tr>
<td>variable</td>
<td>5</td>
</tr>
<tr>
<td>routine: definition</td>
<td>11</td>
</tr>
<tr>
<td>variable: list</td>
<td>16</td>
</tr>
<tr>
<td>routine: heading</td>
<td>11</td>
</tr>
<tr>
<td>variable: preset: declaration</td>
<td>17</td>
</tr>
<tr>
<td>while: phrase</td>
<td>19</td>
</tr>
</tbody>
</table>