This document was produced by SDC and III in performance of contract AF 19(628)-5166 with the Electronic Systems Division, Air Force Systems Command, in performance of ARPA Order 773 for the Advanced Research Projects Agency Information Processing Techniques Office, and Subcontract 65-107.

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- 2710/210/00

AUTHOR Donna Firth
Donna Firth
S. L. Kameny

TECHNICAL S. L. Kameny

RELEASE C. Weissman
for J. I. Schwartz

DATE 8/25/66 PAGE 1 OF 11 PAGES

Syntax of LISP 2 Tokens

ABSTRACT

defines the syntax of LISP 2 at the token level. Tokens are parsed by a finite state machine and then used to construct source language or S-expressions.

FOREWORD

LISP 2 is a joint development of SDC and III. The idea for LISP 2 as a language combining the properties of an algebraic language like ALGOL and the list-processing language LISP was conceived by M. Levin of MIT. Development of the concepts of LISP 2 was carried forth in a series of conferences held at MIT and Stanford University. Contributions in concepts and detail were made by Prof. John McCarthy of Stanford University, Prof. Marvin Minsky of MIT, and the LISP 2 project team consisting of M. Levin, L. Hawkinson, R. Saunders and P. Abrahams of III, and S. Kameny, C. Weissman, E. Book, Donna Firth, J. Barnett and V. Schorre of SDC.

For the implementation of LISP 2, it was decided to define a standard, computer-independent, LISP-like intermediate language and to define the LISP 2 source language in terms of its translation into the intermediate language.

This document describes the syntax of tokens.

CONTENTS

1. INTRODUCTION

Tokens are the units from which S-expressions and source language are constructed. A token has no internal structure as do atoms or lists; it exists momentarily when the finite state machine stops; further existence depends upon the use made of it by the S-expression reader, the syntax translator, or the token reader.

2. CHARACTERS

The LISP 2 character set is the 128 characters of the revised ASCII standard.[1] Character mapping (Section 5) is also available so that non-graphic characters may be entered, or the limitations of certain input devices circumvented. All characters shown in token syntax are assumed to be the result of character mapping which takes precedence when used.

2.1 CHARACTER CLASSES

Tokens are formed by combining character classes rather than individual characters. All 128 characters belong to the class named *character* with sub-classes as in Section 6.1. A basic alphabet of 58 graphic characters, a space (b̸), and 6 non-graphic characters is used in this document. The class assignment of the remaining 63 characters is implementation-dependent. Class membership can be dynamically changed within the LISP 2 system when a user so desires. The finite state machine which parses tokens is also changeable when languages other than source language or S-expressions are being read, although a change may not be required.

If the lower case letters are assigned to the class *letter*, the question of the equivalence of such sequences as 'BEGIN' and 'begin' arises. The answer to this question will depend on implementation.

3. NOTATION

The symbols | { } * , the use of italics, and the form of syntax equations conform to the usage in TM-2710/220/01, LISP 2 Intermediate Language. The characters shown in the basic alphabet stand for themselves (in the ASCII scheme) except for ↑ ← \ which are assigned for each implementation, and the non-graphic characters which are written as b̸ for space, <u>CR</u> for carriage return, <u>NUL</u> for null, etc.

---

[1]E. Lohse, ed., "Proposed Revised American Standard Code for Information Interchange," <u>Comm. ACM</u>, Vol. 8, No. 4, April 1965, pp. 207-214.

Other symbols used are the superscript ^{o}, which means that the entity so designated is not a part of the token which is formed, the negation sign $\sim$, and superscripts referring to footnotes which are not part of the syntax equations themselves. In all *token* syntax equations in Sections 4 through 6, *space* is explicitly indicated in the equations. In Section 7, *spaces* are implicit in the definition of *s:expressions* as *token* sequences.

4. SPECIAL CHARACTERS

The class *escape:character* has one member which will usually be % although this is changeable as are all other class assignments. The use of *escape:character* has the highest precedence in token parsing. At present it is used for creation of unusual identifiers, character mapping, remarks, and hyphenators. The *hyphenator* is actually a special case of character mapping. The syntax of *hyphenator* is given with the basic alphabet because it maps onto the single character NUL.

NUL and *hyphenator* constitute the *null:class* which is a character class completely invisible in token parsing except when preceded by a prime in string context. Outside of this special context the following is always true:

$$\textit{character} \;\; \textit{null:class}^{*} \;\; \textit{character} = \textit{character} \;\; \textit{character}$$

The occurrence of *null:class* is not shown in token syntax; it may occur at any point in a *character* sequence with an effect as above.

5. CHARACTER MAPPING

The meaning of *escape:character* is always governed by the following character. When the *escape:character* itself is intended it is followed by an I. This is the identity mapping and is the only way that the *escape:character* can mean itself. The characters I, b̸, R, G, #, ;, CR, US, RS, and C have special meaning in token syntax when they follow the *escape:character*. The use of *escape:character* C is a general form of character mapping, as follows:

cardinal = *unsigned:integer* | *unsigned:octal* (See Section 6.2)

character = *escape:character* C *cardinal* .

The *character* resulting from this mapping is the one whose numeric code is the same as the *cardinal*. For example, %C101Q. means A. Character mapping is recursive, consequently %%C103Q.101Q. also means A. In this latter example the *character* following the % is C and not another % because of the precedence of character mapping previously mentioned.

The use of b̸, R, G, #, ;, CR, US, and RS following the *escape:character* is given in Sections 6.1 and 6.2; the meaning of any *character* other than these mentioned will depend on implementation.

The use of the characters DEL (delete) and BS (backspace) may cause a form of character mapping, but these also are implementation dependent.

6. TOKENS

All tokens but one are explicitly defined below. The one exception is *unrecognizable*. This is defined by default to mean any *character* sequence which does not satisfy one of the syntax equations for the other tokens. Examples of *unrecognizable* are:

#Ab̸FS 1.E10A 5E6. #('FS;1ZEM

6.1 BASIC ALPHABET

ec = E

gc = G

qc = Q

rc = R

letter = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

octal:digit = 0|1|2|3|4|5|6|7

digit = *octal:digit*|8|9

mark = *|:|/|\|<|>|=|←|↑

p:mark = (|)|[|]|,|$|%|+|-|.|b̸

ordinary = *letter*|*digit*|*mark*|*p:mark*

prime = '

fence = #

semi:colon = ;

period = .

space = b̸

plmn = +|-

u:mark = ,|$|%|'|;

lpar = (

rpar =)

lbrac = [

rbrac =]

boundary = CR|US|RS

data:separator = FS|EM

termin = *semi:colon*|*boundary*

hyphenator = *escape:character termin*|
escape:character space{*ordinary*|*prime*|*fence*}* *termin*

null:class = NUL|*hyphenator*

6.2 SYNTAX OF TOKENS

remark = *escape:character rc* {*ordinary*|*prime*|*fence*}* *termin*

string:spelling = *fence* {*ordinary*|*prime character*[1]|*semi:colon*|*boundary*^{o}}* *fence*

alpha = *letter*|*digit*|*period*

literal = *letter alpha** {~*alpha*}^{o}

dotted:literal = *period*{*letter*|*period*} *alpha** {~*alpha*}^{o}

operator = *mark mark** {~ *mark*}^{o}|*plmn* {~{*period*|*digit*}}^{o}

string:name = *escape:character string:spelling*

gen:spelling = *escape:character gc* {*literal*|*string:name*|*operator*|*dotted:literal*}

n:delimiter = ~{*letter*|*period*}

decimal = *digit digit**

---

[1]The effect of *prime character* is to enter the *character* in the token and to discard the *prime*. This *character* may be any *character* at all, including *boundary*, *data:separator*, and *null:class*. This is the only place in which *null:class* is meaningful.

unsigned:integer = {*decimal ec decimal*|*decimal*}*n:delimiter*^{o}

octal:spelling = *octal:digit octal:digit* qc*

unsigned:octal = {*octal:spelling decimal*|*octal:spelling*}*n:delimiter*^{o}

exponent = *ec*{*plmn decimal*|*decimal*}

mantissa = *decimal period decimal*|*decimal period*|*period decimal*

unsigned:real = {*mantissa exponent*|*mantissa*}*n:delimiter*^{o}

sign = *plmn*{*period*|*digit*}^{o}

spacer = *space space**|*boundary*

dot = *period* {~*alpha*}^{o}

token = *remark*[1]|*string:spelling*|*literal*|*dotted:literal*[2]|*operator*|*string:name*|
gen:spelling|*unsigned:integer*|*unsigned:octal*|*unsigned:real*|*spacer*|
dot|*lpar*|*rpar*|*lbrac*|*rbrac*|*sign*|*u:mark*|*data:separator*|*unrecognizable*

---

[1] A *remark* may occur in SL or IL wherever a *spacer* may be used. In IL commas are not optional. See Section 7.

[2] The definitions of *dot* and *dotted:literal* prevent the character *period* from being a *dotted:literal*.

7. S-EXPRESSIONS AT THE TOKEN LEVEL

Occurrences of words such as FUNCTION, REAL, etc. denote the *token* that was a *literal* with the same character representation.

octal = *sign unsigned:octal*|*unsigned:octal*

integer = *sign unsigned:integer*|*unsigned:integer*

real = *sign unsigned:real*|*unsigned:real*

number = *octal*|*integer*|*real*

spaces = *spacer*|*remark*

empty[1] = *spaces**|

false = FALSE|NIL|*lpar* *rpar*

boolean = TRUE|*false*

string = *string:spelling*

identifier = *literal*[2]|*dotted:literal*|*string:name*|*operator*|*gen:spelling*|*u:mark*

f:name = *lpar* *identifier* *dot* *identifier* *rpar*

value:type = *literal*

f:type = *literal*

a:type = *f:type*|*lpar* *f:type* {LOC|VALUE} *rpar*

i:type = *lpar* *f:type* {LOC|VALUE|*empty*} INDEF *rpar*

functional:constant = *lbrac* FUNCTION *f:name* *lpar* *value:type* *a:type** {*i:type*|*empty*} *rpar* *rbrac*

numeric:row = *lbrac* {*number* *number**| *numeric:row* *numeric:row** } *rbrac*

real:array = *lbrac* REAL {*number** |*numeric:row** } *rbrac*

---

1 *empty* means either a sequence of *spaces* or nothing. It has no semantic effect on the *s:expression*. By the definition of *sign* in Section 6.2, no *spacer* or *remark* can occur between a *sign* and an unsigned number; in all other *s:expressions* which are *token* sequences *empty* may occur between tokens.

2 The character representation is not TRUE, FALSE or NIL.

integer:array = *lbrac* INTEGER {*number** |*numeric:row** } *rbrac*

octal:array = *lbrac* OCTAL {*number** |*numeric:row** } *rbrac*

numeric:array = *real:array*|*integer:array*|*octal:array*

boolean:exp = *s:expression*

boolean:row = *lbrac* {*boolean:exp* *boolean:exp** |*boolean:row* *boolean:row** } *rbrac*

boolean:array = *lbrac* BOOLEAN {*boolean:exp** |*boolean:row** } *rbrac*

symbol:element = *boolean*|*number*|*string*|*identifier*|*list*| *dot*{*array*|*functional:constant*}[1]

symbol:row = *lbrac* {*symbol:element* *symbol:element** |*symbol:row* *symbol:row** } *rbrac*

symbol:array = *lbrac* SYMBOL {*symbol:element** |*symbol:row** } *rbrac*

functional:row = *lbrac* {*functional:constant* *functional:constant** | *functional:row* *functional:row** } *rbrac*

functional:array = *lbrac* FUNCTIONAL {*functional:constant** | *functional:row** } *rbrac*

array[2] = *boolean:array*|*numeric:array*|*symbol:array*|*functional:array*

simple:datum = *boolean*|*number*|*string*|*array*|*functional:constant*

atom = *simple:datum*|*identifier*

list = *lpar* *s:expression* *s:expression** {*dot* *s:expression*|*empty*} *rpar*

s:expression = *atom*|*list*

---

[1] This notation is used for arrays or functional constants that are elements of the *symbol:array* and not for sub-elements. For example:

```
[SYMBOL A (B C) .[REAL 1.0 2.0]]
[SYMBOL [A (B C) .[INTEGER 1 2]] [(x) #(# (E . F)]]
```

require the notation but

```
[SYMBOL A (B [REAL 1.0 2.0])]
```

does not. In the last example use of *dot* would make the element into a dotted pair.

[2] The rows of a multi-dimensional array must have the same number of elements.

Distribution

R. Barancik	2105
J. Barnett	2025
R. Berman	4317
E. Book	2332
R. Bosak	2013
J. Burger	9919
D. Drukey	2105
Marsha Drapkin(2)	9723
G. Feingold	9525
D. Firth	2310
M. Howard	2042
H. Howell	9912
A. Irvine	9627
E. Jacobs	2344
B. Jones	2231
S. Kameny (50)	2009
R. Long	9913
R. Martin	2228
E. Myer	2227
M. Perstein	2344
D. Perry	2042
V. Schorre	2330
J. Schwartz	2123
E. Simmons	9439
E. Stefferud	9734
A. Vorhaus	2213
C. Weissman(10)	2214
R. Wolfson	2368
L. Hawkinson(III)	**9720**
D. Crandel (III)	**9721**
D Anschultz (III)	**9721**
B. Saunders (III)	**9721**
P. Stygar (III)	**9721**