stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 98/78 SEPTEMBER

H.B.M. JONKERS

A FINITE STATE LEXICAL ANALYZER FOR THE STANDARD HARDWARE REPRESENTATION OF ALGOL 68

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A15

ACM-Computing Reviews-categories: 4.12

A finite state lexical analyzer for the standard hardware representation of ALGOL 68*)

by

H.B.M. Jonkers

ABSTRACT

A finite state lexical analyzer for ALGOL 68 programs written in the standard hardware representation is described. The analyzer is written in a very simple language, allowing semi-mechanical translation to an arbitrary language. The whole language, including format-texts, is dealt with.

KEY WORDS & PHRASES: ALGOL 68, lexical analysis, finite state machine, semi-mechanical translation.

*) This report will be submitted for publication elsewhere

1. INTRODUCTION

For two reasons the lexical analysis of ALGOL 68 programs is not as trivial as might be expected. First of all at some places (e.g., TAO-symbols) the lexical structure of ALGOL 68 is rather awkward. Secondly ALGOL 68 programs can be represented in different stropping regimes [1]. A lexical analyzer for ALGOL 68 featuring all three stropping regimes has already been published [2]. Apart from the deviations from [1] mentioned in the next paragraph, the lexical analyzer described here differs from [2] in the following points:

(1) It basically is a finite state machine. This allows a wide range of implementation methods to be applied and adds to efficiency.
(2) It is described in a very simple language, allowing semi-mechanical translation to an arbitrary language (e.g., machine language). The lexical analyzer was in fact tested by translating it into an ALEPH program using a text editor.
(3) All parts of programs are dealt with, including format-texts.
(4) The description is hopefully more accessible and more readable than [2].

The lexical analyzer takes as its input program texts representing ALGOL 68 particular-programs in the standard hardware representation [1], allowing the following deviations from [1]:

(1) Besides worthy characters all characters occurring in section 9.4.1. of [3] are allowed; for a list of all characters accepted by the lexical analyzer see appendix 1. If only worthy characters are to be accepted, this can be achieved by adding a preprocessor to the lexical analyzer accepting worthy characters only.
(2) Besides the three stropping regimes defined in [1], a fourth regime is provided, the STROP regime. In the STROP regime, tags and bolds are represented as they are in POINT stropping, with the addition of the following rule:
    - A bold word may be written as a strop ("'"), followed, in order, by the worthy letters or digits corresponding to the bold-faced letters or digits in the word, followed by a strop. If the bold word is followed by a disjunctor other than a strop, the last strop may be omitted.
(3) In the RES regime the point may be omitted from a bold word if it is preceded by a digit from an integral-, real- or bits-denotation (cf. [4]).

The output of the lexical analyzer consists of "tokens", which we shall call "words" (as in [2]) to prevent confusion, since there already is an ALGOL 68 paranotion "token". The exact definition of a "word" is given in section 3. Roughly speaking a "word" corresponds to an ALGOL 68 denotation, comment or NOTION-symbol. Each time the lexical analyzer is activated, it delivers a word. By repeated activation of the lexical analyzer the program text will be transformed into a stream of words. If the program text corresponds to an ALGOL 68 particular program in the standard hardware representation (augmented as above), the stream of words will correspond to this particular program in a way more fully described in section 2. If the program text does not satisfy the specifications of the standard hardware representation, the lexical analyzer will generate one or more error

messages. Otherwise the program text, and consequently the stream of words, does not correspond to an ALGOL 68 program. If the lexical analyzer is part of a compiler, this will lead to an error message at a higher level in the compiler.

The lexical analyzer itself consists of four separate lexical analyzers, one for each stropping regime. The first advantage of this is an increase of efficiency: it is no longer necessary to inspect the environment continually during lexical analysis to determine which stropping regime we are in. Second, if we don't want to allow all of the stropping regimes, we can simply omit the lexical analyzers for one or more of the stropping regimes. In this way, we are not burdened with the details of stropping regimes which are not allowed anyway, as would be the case with a lexical analyzer in which all stropping regimes are integrated. A disadvantage seems to be the size of such a lexical analyzer when allowing more than one stropping regime. However, since the lexical analyzers for the different stropping regimes differ from each other at only a limited number of places, large parts of them can be combined. This combination of the separate lexical analyzers is not difficult and is left to the implementer (see also note 1 in section 7). The coordination of the separate lexical analyzers during lexical analysis must be taken care of by the global routine using them (e.g., a parser). We shall call this routine the "supervisor".

As the lexical analyzer is composed of four lexical analyzers, one for every stropping regime, so is in turn each lexical analyzer made up of two analyzers: the "unit level lexical analyzer" and the "format level lexical analyzer". The unit level lexical analyzer is designed to analyze program text at the unit and pragmat level, assuming that the interior of pragmats has a somewhat ALGOL 68-like structure. Comments are automatically skipped by the unit level lexical analyzer. The format level lexical analyzer is designed to analyze program text at the format-text level, comments also being skipped automatically. A considerable part of the unit and format level lexical analyzer coincides, so they can partially be combined. The supervisor must coordinate the unit and format level lexical analyzer. We shall often use "the lexical analyzer" to mean one of the separate (unit or format level) lexical analyzers.

For reasons of efficiency, the model of a finite transducer has been chosen for the lexical analyzer, i.e., the lexical analyzer can be viewed as a program for a finite state machine. The description of this machine is found in section 3. The machine is completely described in ALGOL 68 by a number of data structures and a number of operations on these data structures, which we shall call "instructions". Moreover, a number of predicates on these data structures is given, which we shall call "conditions". These "conditions" are used to enable conditional state transitions. We point out here beforehand, that this method of description has only been chosen for the sake of clarity and is not the best way to implement the machine (see section 7). To describe the program which is to run on this machine, we use a mini language called ALEX, defined in sections 4 and 5. Programs in ALEX are closely related to right-linear (transduction) grammars. The entire lexical analyzer was in fact constructed by transforming context-free grammars for the different words into right-linear grammars and subsequently combining these into an ALEX program. The lexical analyzer program itself is listed in section 6.

2. WORDS

A word is a value with a structure (a "mode") described by the following ALGOL 68 declaration:

```
mode word = struct (int mark, string info);
```

The words generated by the lexical analyzer are described below. For each value of the mark field the corresponding paranotion(s) is (are) given. For each value of the info field the corresponding representation of the paranotion in the reference language is given, omitting typographical display features (the Greek letter "ξ" is used to indicate a character). Values of the mark field are indicated by names in upper case letters. Values of the info field are indicated by strings (without embracing quotes), "ε" indicating the empty string.

Remarks:

(1) It is not always possible for a finite state machine to determine whether an "=" at the end of a TAO-symbol belongs to this TAO-symbol or not (see also [2]). In case of doubt the TAO-symbol and the "=" are packed together into one word with mark = SHORTOP EQUALSETY (in contrast with the algorithm in [2]). Words with mark = SHORTOP EQUALSETY are the only words that may correspond to a sequence of more than one symbols (see the second column in the table below).

(2) For some applications the filling of the info field of some words might have to be changed. For example, if comments should not be discarded, the info field of a word with mark = COMMENT could be filled with the comment text. In general, no fundamental changes in the lexical analyzer are needed for this. In most cases the insertion and/or deletion of a few "instructions" in the lexical analyzer program will suffice.

(3) For the value "EOF" of the mark field no corresponding paranotion is given since there is none. A word with mark = EOF is used to indicate the end of the word stream.

mark	paranotion	info	representation
TAG	TAG-symbol.	$\xi_1 \cdots \xi_n$	$\xi_1 \cdots \xi_n$
BOLD	bold-TAG-symbol. except: bold-comment-symbol; style-i-comment-symbol; bold-pragmat-symbol; style-i-pragmat-symbol.	$\xi_1 \cdots \xi_n$	$\xi_1 \cdots \xi_n$
INT	integral-denotation.	$\xi_1 \cdots \xi_n$	$\xi_1 \cdots \xi_n$
REAL	real-denotation.	$\xi_1 \cdots \xi_n$	$\xi_1 \cdots \xi_n$
BITS	bits-denotation.	$\xi_1 \cdots \xi_n$	$\xi_1 \cdots \xi_n$
SHORTOP	DOP-BECOMESETY-symbol. except: equals-symbol; tilde-symbol.	$\xi_1 \cdots \xi_n$	$\xi_1 \cdots \xi_n$
SHORTOP EQUALSETY	DYAD-cum-equals-symbol; DYAD-symbol, is-defined-as-symbol; DYAD-cum-equals-cum- becomes-symbol; DYAD-cum-assigns-to-symbol, is-defined-as-symbol.	$\xi_1 \cdots \xi_n =$	$\xi_1 \cdots \xi_n =$
STRING	string-denotation.	$\xi_1 \cdots \xi_n$	"$\xi_1 \cdots \xi_n$"
CHAR	character-denotation.	ξ	"ξ"
BECOMES	becomes-symbol.	ε	:=
IS	is-symbol.	ε	:=:
ISNOT	is-not-symbol.	ε	:≠:
		ε	:/=:
STICKCOLON	brief-else-if-symbol; brief-ouse-symbol.	ε	|:
EQUALS	equals-symbol; is-defined-as-symbol.	ε	=
TILDE	tilde-symbol; skip-symbol.	ε	~
STICK	brief-then-symbol; brief-else-symbol; brief-in-symbol; brief-out-symbol.	ε	|
COLON	label-symbol; colon-symbol; up-to-symbol; routine-symbol.	ε	:
COMMA	and-also-symbol.	ε	,
SEMICOLON	go-on-symbol.	ε	;
OPEN	brief-begin-symbol; brief-if-symbol; brief-case-symbol; style-i-sub-symbol.	ε	(
CLOSE	brief-end-symbol; brief-fi-symbol; brief-esac-symbol; style-i-bus-symbol.	ε	)
SUB	brief-sub-symbol.	ε	[
BUS	brief-bus-symbol.	ε	]

AT	at-symbol.	ε	@
NIL	nil-symbol.	ε	∘
DOLLAR	formatter-symbol.	ε	$
COMMENT	comment.	¢	¢$\xi_1 \ldots \xi_n$¢
		comment	comment$\xi_1 \ldots \xi_n$comment
		co	co$\xi_1 \ldots \xi_n$co
		#	#$\xi_1 \ldots \xi_n$#
PRAGSYM	bold-pragmat-symbol; style-i-pragmat-symbol.	pragmat	pragmat
		pr	pr
EOF		ε	

The following words can be generated by the format level lexical analyzer exclusively:

CHARROW	string-denotation; character-denotation.	$\xi_1 \ldots \xi_n$	"$\xi_1 \ldots \xi_n$"
FIXNUM	fixed-point-numeral.	$\xi_1 \ldots \xi_n$	$\xi_1 \ldots \xi_n$
ASYM	letter-a-symbol.	ε	a
BSYM	letter-b-symbol.	ε	b
CSYM	letter-c-symbol.	ε	c
DSYM	letter-d-symbol.	ε	d
ESYM	letter-e-symbol.	ε	e
FSYM	letter-f-symbol.	ε	f
GSYM	letter-g-symbol.	ε	g
ISYM	letter-i-symbol.	ε	i
KSYM	letter-k-symbol.	ε	k
LSYM	letter-l-symbol.	ε	l
NSYM	letter-n-symbol.	ε	n
PSYM	letter-p-symbol.	ε	p
QSYM	letter-q-symbol.	ε	q
RSYM	letter-r-symbol.	ε	r
SSYM	letter-s-symbol.	ε	s
XSYM	letter-x-symbol.	ε	x
YSYM	letter-y-symbol.	ε	y
ZSYM	letter-z-symbol.	ε	z
POINT	point-symbol.	ε	.
PLUS	plus-symbol.	ε	+
MINUS	minus-symbol.	ε	-

3. MACHINE

The lexical analyzer programs are described in a language called ALEX (see sections 4 and 5). ALEX programs describe a series of actions of a "machine". This machine is described below by a set of ALGOL 68 declarations. The machine consists of a number of data structures, a number of actions on the data structures, called "instructions", and a number of predicates on the data structures, called "conditions". The "instructions" are used in ALEX programs to denote primitive actions of the machine. The "conditions" are used to make decisions dependent upon the value of the machine data structures.

```
# 1. Data structures. #

struct (int state, string buffer) status;

# The variable "status" represents the status of the machine.
  The "state" field holds the current state of the machine.
  The "buffer" field is used to cope with lookahead. #

string input;
char head;

# The variable "input" represents the input file.
  The variable "head" is used to temporarily save the first character of
  "input". #

struct (int mark, string info) word;

# The variable "word" is used to pass information on the token which has
  been read to the outside world. #

int match index;
bool match possible;

# The variables "match index" and "match possible" are used for pattern
  matching purposes inside comments, thus allowing an efficient skipping of
  comments. #

# 2. Auxiliary definitions. #

char eof = ...;

# "eof" is used as an end of file marker and must be some character that
  cannot occur in the input. #

op norm = (char ch) char:
   if   ch = "A" then "a"
   elif ch = "B" then "b"
   .
   .
   .
   elif ch = "Z" then "z"
```

```
   else ch
   fi;

# We need the operator "norm" because of the fact that with a few
  exceptions the two cases of a letter are equivalent. #

proc write = (string s) void: info of word +:= s;

op head = (string s) char: s[1];
op tail = (string s) string: s[2 : upb s];

proc reserved = (string s) bool:
   (s = "at" or s = "begin" or ... or s = "while");
proc comment = (string s) bool:
   (s = "co" or s = "comment");
proc pragmat = (string s) bool:
   (s = "pr" or s = "pragmat");

# 3. Instructions. #

proc put = void: write(norm head);
proc putitem = void: write(head);
proc save = void: buffer of status +:= norm head;
proc clear = void: buffer of status := "";
proc append = void: begin write(buffer of status); clear end;
proc reread = void: begin buffer of status +=: input; clear end;
proc read = void: head := if input = "" then eof else head input fi;
proc next = void: input := tail input;

proc point = void: write(".");
proc zero = void: write("0");
proc quote = void: write("""");
proc strop = void: write("´´");
proc equals = void: write("=");
proc tilde = void: write("~");
proc colon = void: write(":");
proc differs = void: write("≠");
proc divided = void: write("/");

proc reset = void: begin match index := 0; match possible := true end;
proc match = void:
   if match possible
   then if match index < upb info of word
        then match index +:= 1;
             match possible := norm head = info[match index]
        else match possible := false
        fi
   fi;

proc error = (int n) void: ...;

# What should be done when an error occurs is left to the implementer.
  For error diagnostics, see appendix 2. #

# 4. Conditions. #
```

```
proc reservedinfo = bool: reserved(info of word);
proc reservedbuffer = bool: reserved(buffer of status);
proc commentinfo = bool: comment(info of word);
proc commentbuffer = bool: comment(buffer of status);
proc pragmatinfo = bool: pragmat(info of word);
proc pragmatbuffer = bool: pragmat(buffer of status);
proc two = bool: info of word = "2";
proc four = bool: info of word = "4";
proc eight = bool: info of word = "8";
proc sixteen = bool: info of word = "16";
proc sizeone = bool: upb info of word = 1;
proc sizetwo = bool: upb info of word = 2;
proc sizethree = bool: upb info of word = 3;

proc matching = bool:
   (match possible and match index = upb info of word);
```

4. SYNTAX OF ALEX

ALEX programs syntactically resemble right-linear grammars. The only difference is that to every production rule a (possibly empty) "action", and to every "single production" rule a (possibly empty) "condition" is associated. If we omit the "actions" and "conditions", what remains is a pure right-linear grammar. In the case of the lexical analyzer described here, this grammar generates an (infinite) stream of ALGOL 68 symbols in the standard hardware representation. The syntax of ALEX is given by a van Wijngaarden grammar. The van Wijngaarden grammar is used here only in its most simple form, viz. as an abbreviation mechanism for a context free grammar. The syntax introduces a terminology, which is used in the next section to define the semantics of ALEX.

```
PRODUCTIVITY::
     productive;
     nonproductive.
program:
     transduction rule sequence.
transduction rule:
     PRODUCTIVITY transduction rule.
PRODUCTIVITY transduction rule:
     defined state, colon symbol, PRODUCTIVITY transduction rule body.
defined state:
     state.
PRODUCTIVITY transduction rule body:
     PRODUCTIVITY alternative sequence option, out alternative.
PRODUCTIVITY alternative:
     PRODUCTIVITY condition, transduction, go on symbol.
productive condition:
     charset.
nonproductive condition:
     sub symbol, condition, bus symbol.
transduction:
     curly open symbol, action, curly close symbol, applied state.
action:
     empty;
     mark;
     instruction list;
     instruction list, and also symbol, mark.
applied state:
     state.
out alternative:
     transduction.
```

Some notions are not defined in the syntax; we define them informally below.

```
state       : a state of the machine.
charset     : a set of characters.
condition   : a predicate on the machine data structures.
instruction: an operation on the machine data structures.
mark        : a value of the mark field of a word.
```

In addition, an ALEX program must satisfy the following conditions:

(1) All charsets in a productive transduction rule are disjoint.
(2) All conditions in a nonproductive transduction rule are mutually exclusive.
(3) All defined states are different.
(4) All applied states occur as a defined state.

Remarks:

(1) A termination condition for ALEX programs could be added without great difficulty. However, since we only use ALEX for the description of the lexical analyzer, we shall omit this. Termination of the constituent programs of the lexical analyzer (see section 6) can be verified rather easily.
(2) A transduction rule with a body consisting of an out alternative only can be parsed as a productive as well as a nonproductive transduction rule. Since in this case both kinds of transduction rules are semantically equivalent, the ambiguity causes no harm.

5. SEMANTICS OF ALEX

We shall define the semantics of an ALEX program by translating it into a pseudo ALGOL 68 procedure operating on the machine described in section 3.

TRANSLATION OF A PROGRAM:

Let P be an ALEX program.
P = "R1 ... Rn",
where R1, ... , Rn are transduction rules.
The translation TRANS(P) of P is defined as:

```
TRANS(P) = "proc p = void:
            begin word := (skip, "");
               goto state of status;
               TRANS(R1);
               .
               .
               .
               TRANS(Rn);
               exit:
            end"
```

TRANSLATION OF A TRANSDUCTION RULE:

Let R be a transduction rule.

(1) R is a productive transduction rule.
R = "S: C1 T1; ... ; Cn Tn; T0.",
where S is a state,
C1, ... , Cn are charsets,
T0, ... , Tn are transductions.
The translation TRANS(R) of R is defined as:

If n = 0:

TRANS(R) = "S: TRANS(T0)"

If n > 0:

```
TRANS(R) = "S: read;
               if   head in C1 then next; TRANS(T1)
               elif head in C2 then next; TRANS(T2)
               .
               .
               .
               elif head in Cn then next; TRANS(Tn)
               else TRANS(T0)
               fi"
```

N.B.
The instruction "read" does not remove a character from the string

"input" ("next" does). It merely assigns the head of "input" to "head".

(2) R is a nonproductive transduction rule.
R = "S: [B1] T1; ... ; [Bn] Tn; T0.",
where S is a state,
B1, ... , Bn are conditions,
T0, ... , Tn are transductions.
The translation TRANS(R) of R is defined as:

If n = 0:

TRANS(R) = "S: TRANS(T0)"

If n > 0:

```
TRANS(R) = "S: if   B1 then TRANS(T1)
               elif B2 then TRANS(T2)
               .
               .
               .
               elif Bn then TRANS(Tn)
               else TRANS(T0)
               fi"
```

TRANSLATION OF A TRANSDUCTION:

Let T be a transduction.

(1) T = "{I1, ... , In} S",
where I1, ... , In are instructions,
S is a state.
The translation TRANS(T) of T is defined as:

```
TRANS(T) = "I1; ... ; In;
            state of status := S;
            goto S"
```

(2) T = "{I1, ... , In, M} S",
where I1, ... , In are instructions,
M is a mark,
S is a state.
The translation TRANS(T) of T is defined as:

```
TRANS(T) = "I1; ... ; In;
            mark of word := M;
            state of status := S;
            goto exit"
```

6. PROGRAMS

There are eight ALEX programs constituting the lexical analyzer, one for each pair (level, regime), where level is UNIT or FORMAT and regime is POINT, UPPER, RES or STROP. Large parts of these programs are textually equal. Rather than listing them all in their full length, we shall combine them in a single listing and use two variables "level" and "regime" inside the text to indicate what part of the text belongs to what program. So the program for level = l and regime = r can be constructed by simply erasing all text with level ≠ l or regime ≠ r.

Remarks:

(1) The names of the states have been chosen so as to indicate the string of characters that has been read so far.
(2) All charsets occurring in the transduction rules are listed in appendix l, except for the charset "other". The latter is not a fixed charset but, if it occurs in a transduction rule T, it is equal to the set of all characters (except "eof") that are not element of a charset of T (other than "other").
(3) The state "STRINGESCAPE" has been provided to enable the use of the strop character as an escape character inside character and string denotations. If the strop character is to be used this way, the transduction rule for this state must be modified.
(4) Before the first activation of a program the machine must be initialized properly. This initialization should be done by the supervisor and should read:

```
status := (EMPTY, "");
```

LISTING OF THE PROGRAMS

```
level = UNIT
|
| regime = POINT
| |
| | EMPTY:
| |     letter {put} TAG;
| |     point {} POINT;
| |     digit {put} FIX;
| |     quote {} QUOTE STRING;
| |     equals {} EQUALS;
| |     tilde {} TILDE;
| |     dyad {put} DYAD;
| |     stick {} STICK;
| |     colon {} COLON;
| |     comma {COMMA} EMPTY;
| |     semicolon {SEMICOLON} EMPTY;
| |     open {OPEN} EMPTY;
| |     close {CLOSE} EMPTY;
| |     sub {SUB} EMPTY;
| |     bus {BUS} EMPTY;
| |     at {AT} EMPTY;
| |     nil {NIL} EMPTY;
| |     dollar {DOLLAR} EMPTY;
| |     cent {put} BRIEFCOMMENT;
| |     cross {put} STYLEIICOMMENT;
| |     typo {} EMPTY;
| |     other {error(1)} EMPTY;
| |     {EOF} EMPTY.
|
| regime = UPPER
| |
| | EMPTY:
| |     lowerletter {put} TAG;
| |     upperletter {put} POINTETY UPPERBOLD;
| |     point {} POINT;
| |     digit {put} FIX;
| |     quote {} QUOTE STRING;
| |     equals {} EQUALS;
| |     tilde {} TILDE;
| |     dyad {put} DYAD;
| |     stick {} STICK;
| |     colon {} COLON;
| |     comma {COMMA} EMPTY;
| |     semicolon {SEMICOLON} EMPTY;
| |     open {OPEN} EMPTY;
| |     close {CLOSE} EMPTY;
| |     sub {SUB} EMPTY;
| |     bus {BUS} EMPTY;
| |     at {AT} EMPTY;
| |     nil {NIL} EMPTY;
| |     dollar {DOLLAR} EMPTY;
| |     cent {put} BRIEFCOMMENT;
| |     cross {put} STYLEIICOMMENT;
| |     typo {} EMPTY;
| |     other {error(1)} EMPTY;
| |     {EOF} EMPTY.
```

```
|
| regime = RES
| |
| | EMPTY:
| |    letter {put} TAGBOLD;
| |    point {} POINT;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    equals {} EQUALS;
| |    tilde {} TILDE;
| |    dyad {put} DYAD;
| |    stick {} STICK;
| |    colon {} COLON;
| |    comma {COMMA} EMPTY;
| |    semicolon {SEMICOLON} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    sub {SUB} EMPTY;
| |    bus {BUS} EMPTY;
| |    at {AT} EMPTY;
| |    nil {NIL} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(1)} EMPTY;
| |    {EOF} EMPTY.
|
| regime = STROP
| |
| | EMPTY:
| |    letter {put} TAG;
| |    point {} POINT;
| |    strop {} STROP;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    equals {} EQUALS;
| |    tilde {} TILDE;
| |    dyad {put} DYAD;
| |    stick {} STICK;
| |    colon {} COLON;
| |    comma {COMMA} EMPTY;
| |    semicolon {SEMICOLON} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    sub {SUB} EMPTY;
| |    bus {BUS} EMPTY;
| |    at {AT} EMPTY;
| |    nil {NIL} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(1)} EMPTY;
| |    {EOF} EMPTY.
|
```

```
| regime = POINT, STROP
| |
| | TAG:
| |    letgit {put} TAG;
| |    typoscore {} TAG TYPOSCORE;
| |    {TAG} EMPTY.
| |
| | TAG TYPOSCORE:
| |    letgit {put} TAG;
| |    typo {} TAG TYPOSCORE;
| |    {TAG} EMPTY.
|
| regime = UPPER
| |
| | TAG:
| |    lowerletgit {put} TAG;
| |    underscore {} TAG UNDERSCORE;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG UNDERSCORE:
| |    lowerletgit {put} TAG;
| |    upperletter {save, error(5), TAG} POINTETY UPPERLETTER;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG TYPO:
| |    lowerletgit {put} TAG;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
|
| regime = RES
| |
| | TAGBOLD:
| |    letgit {put} TAGBOLD;
| |    underscore {} TAG UNDERSCORE;
| |    typo {} TAGBOLD TYPO;
| |    {} TAGBOLD END.
| |
| | TAGBOLD TYPO:
| |    [reservedinfo] {} BOLD;
| |    {} TAG TYPO.
| |
| | TAGBOLD END:
| |    [reservedinfo] {} BOLD;
| |    {TAG} EMPTY.
| |
| | TAG:
| |    letgit {put} TAG;
| |    underscore {} TAG UNDERSCORE;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
```

```
| | TAG UNDERSCORE:
| |    letgit {put} TAG;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG TYPO:
| |    letter {save} TAG BOLDETY;
| |    digit {put} TAG;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG BOLDETY:
| |    letgit {save} TAG BOLDETY;
| |    underscore {append} TAG UNDERSCORE;
| |    typo {} TAG BOLDETY TYPO;
| |    {} TAG BOLDETY END.
| |
| | TAG BOLDETY TYPO:
| |    [reservedbuffer] {TAG} SAVEDBOLD;
| |    {append} TAG TYPO.
| |
| | TAG BOLDETY END:
| |    [reservedbuffer] {TAG} SAVEDBOLD;
| |    {append, TAG} EMPTY.
| |
| | SAVEDBOLD:
| |    {append} BOLD.
|
| regime = POINT, RES, STROP
| |
| | POINT:
| |    letter {put} POINT BOLD;
| |    digit {point, put} VAR;
| |    typo {} POINT TYPO;
| |    {error(3)} EMPTY.
| |
| | POINT TYPO:
| |    digit {point, put} VAR;
| |    typo {} POINT TYPO;
| |    {error(3)} EMPTY.
| |
| | POINT BOLD:
| |    letgit {put} POINT BOLD;
| |    underscore {error(6)} BOLD;
| |    {} BOLD.
|
| regime = UPPER
| |
| | POINT:
| |    lowerletter {put} POINT LOWERBOLD;
| |    upperletter {put} POINTETY UPPERBOLD;
| |    digit {point, put} VAR;
| |    typo {} POINT TYPO;
| |    {error(3)} EMPTY.
| |
```

```
| | POINT TYPO:
| |     digit {point, put} VAR;
| |     typo {} POINT TYPO;
| |     {error(3)} EMPTY.
| |
| | POINT LOWERBOLD:
| |     lowerletgit {put} POINT LOWERBOLD;
| |     underscore {error(6)} BOLD;
| |     {} BOLD.
| |
| | POINTETY UPPERBOLD:
| |     upperletgit {put} POINTETY UPPERBOLD;
| |     underscore {error(6)} BOLD;
| |     {} BOLD.
|
| regime = STROP
| |
| | STROP:
| |     letter {put} STROP BOLD;
| |     {error(4)} EMPTY.
| |
| | STROP BOLD:
| |     letgit {put} STROP BOLD;
| |     strop {} BOLD;
| |     underscore {error(6)} BOLD;
| |     {} BOLD.
|
| regime = POINT, UPPER, RES, STROP
| |
| | BOLD:
| |     [commentinfo] {} BOLDCOMMENT;
| |     [pragmatinfo] {PRAGSYM} EMPTY;
| |     {BOLD} EMPTY.
|
| regime = POINT, RES, STROP
| |
| | FIX:
| |     digit {put} FIX;
| |     point {} FIX POINT;
| |     ten {put} STAG POWER;
| |     letter e {save} FIX E;
| |     letter r {save} FIX R;
| |     typo {} FIX;
| |     {INT} EMPTY.
| |
| | FIX POINT:
| |     digit {point, put} VAR;
| |     letter {save, INT} POINT LETTER;
| |     typo {point} FIX POINT TYPO;
| |     {point, zero, error(8)} VAR.
|
```

```
regime = UPPER
|
| FIX:
|    digit {put} FIX;
|    point {} FIX POINT;
|    ten {put} STAG POWER;
|    lowerletter e {save} FIX E;
|    lowerletter r {save} FIX R;
|    typo {} FIX;
|    {INT} EMPTY.
|
| FIX POINT:
|    digit {point, put} VAR;
|    lowerletter {save, INT} POINT LOWERLETTER;
|    upperletter {save, INT} POINTETY UPPERLETTER;
|    typo {point} FIX POINT TYPO;
|    {point, zero, error(8)} VAR.

regime = POINT, UPPER, RES, STROP
|
| FIX POINT TYPO:
|    digit {put} VAR;
|    typo {} FIX POINT TYPO;
|    {zero, error(8)} VAR.
|
| FIX E:
|    digit {append, put} FLO;
|    sign {append, put} STAG POWER SIGN;
|    typo {append} STAG POWER;
|    {INT} LEGGLE.

regime = POINT, UPPER, STROP
|
| FIX R:
|    [two] {} RADIX R(1);
|    [four] {} RADIX R(2);
|    [eight] {} RADIX R(3);
|    [sixteen] {} RADIX R(4);
|    {INT} LEGGLE.

regime = RES
|
| FIX R:
|    [two] {} RADIX R(1);
|    [four] {} RADIX R(2);
|    [eight] {} RADIX R(3);
|    [sixteen] {} HEXBITS LEGGLE;
|    {INT} LEGGLE.
|
```

```
| regime = POINT, RES, STROP
| |
| | VAR:
| |     digit {put} VAR;
| |     ten {put} STAG POWER;
| |     letter e {save} VAR E;
| |     typo {} VAR;
| |     {REAL} EMPTY.
|
| regime = UPPER
| |
| | VAR:
| |     digit {put} VAR;
| |     ten {put} STAG POWER;
| |     lowerletter e {save} VAR E;
| |     typo {} VAR;
| |     {REAL} EMPTY.
|
| regime = POINT, UPPER, RES, STROP
| |
| | VAR E:
| |     digit {append, put} FLO;
| |     sign {append, put} STAG POWER SIGN;
| |     typo {append} STAG POWER;
| |     {REAL} LEGGLE.
| |
| | STAG POWER:
| |     digit {put} FLO;
| |     sign {put} STAG POWER SIGN;
| |     typo {} STAG POWER;
| |     {zero, error(9), REAL} EMPTY.
| |
| | STAG POWER SIGN:
| |     digit {put} FLO;
| |     typo {} STAG POWER SIGN;
| |     {zero, error(9), REAL} EMPTY.
| |
| | FLO:
| |     digit {put} FLO;
| |     typo {} FLO;
| |     {REAL} EMPTY.
|
| regime = POINT, RES, STROP
| |
| | RADIX R(n):
| |     radigit(n) {append, put} BITS(n);
| |     noradletgit(n) {save, INT} LEGGLE;
| |     typo {append} RADIX R TYPO(n);
| |     {append, zero, error(10), BITS} EMPTY.
| |
| | RADIX R TYPO(n):
| |     radigit(n) {put} BITS(n);
| |     typo {} RADIX R TYPO(n);
| |     {zero, error(10), BITS} EMPTY.
| |
```

```
| | BITS(n):
| |    radigit(n) {put} BITS(n);
| |    typo {} BITS(n);
| |    {BITS} EMPTY.
|
| regime = UPPER
| |
| | RADIX R(n):
| |    lowerradigit(n) {append, put} BITS(n);
| |    lowernoradletgit(n) {save, INT} LEGGLE;
| |    typo {append} RADIX R TYPO(n);
| |    {append, zero, error(10), BITS} EMPTY.
| |
| | RADIX R TYPO(n):
| |    lowerradigit(n) {put} BITS(n);
| |    typo {} RADIX R TYPO(n);
| |    {zero, error(10), BITS} EMPTY.
| |
| | BITS(n):
| |    lowerradigit(n) {put} BITS(n);
| |    typo {} BITS(n);
| |    {BITS} EMPTY.
|
| regime = RES
| |
| | HEXBITS:
| |    digit {put} HEXBITS;
| |    hexletter {save} HEXBITS LEGGLE;
| |    nohexletter {save} HEXBITS LEGGLE END;
| |    typo {} HEXBITS;
| |    {} HEXBITS END.
| |
| | HEXBITS END:
| |    [sizethree] {zero, error(10), BITS} EMPTY;
| |    {BITS} EMPTY.
| |
| | HEXBITS LEGGLE:
| |    digit {append, put} HEXBITS;
| |    hexletter {save} HEXBITS LEGGLE;
| |    nohexletter {save} HEXBITS LEGGLE END;
| |    typo {append} HEXBITS;
| |    {append} HEXBITS END.
| |
| | HEXBITS LEGGLE END:
| |    [sizetwo] {INT} LEGGLE;
| |    [sizethree] {zero, error(10), BITS} LEGGLE;
| |    {BITS} LEGGLE.
|
| regime = POINT, UPPER, STROP
| |
| | LEGGLE:
| |    {append} TAG.
|
```

```
| regime = RES
| |
| | LEGGLE:
| |     {append} TAGBOLD.
|
| regime = POINT, RES, STROP
| |
| | POINT LETTER:
| |     {append} POINT BOLD.
|
| regime = UPPER
| |
| | POINT LOWERLETTER:
| |     {append} POINT LOWERBOLD.
| |
| | POINTETY UPPERLETTER:
| |     {append} POINTETY UPPERBOLD.
|
| regime = POINT, UPPER, RES, STROP
| |
| | STRINGRETURN:
| |     [sizeone] {CHAR} EMPTY;
| |     {STRING} EMPTY.
| |
| | EQUALS:
| |     equals {equals, equals} DYAD EQUALS;
| |     nomad {equals, put} DYAD NOMAD;
| |     colon { } EQUALS COLON;
| |     {EQUALS} EMPTY.
| |
| | EQUALS COLON:
| |     equals {equals, colon, equals, SHORTOP} EMPTY;
| |     {EQUALS} COLON.
| |
| | TILDE:
| |     equals {tilde, equals} DYAD EQUALS;
| |     nomad {tilde, put} DYAD NOMAD;
| |     colon {tilde} DYAD NOMADETY COLON;
| |     {TILDE} EMPTY.
| |
| | DYAD:
| |     equals {equals} DYAD EQUALS;
| |     nomad {put} DYAD NOMAD;
| |     colon { } DYAD NOMADETY COLON;
| |     {SHORTOP} EMPTY.
| |
| | DYAD EQUALS:
| |     equals { } DYAD NOMAD EQUALS;
| |     colon {colon} DYAD EQUALS COLON;
| |     typo { } SHORTOP EQUALSETY TYPOSETY;
| |     {SHORTOP EQUALSETY} EMPTY.
| |
| | DYAD EQUALS COLON:
| |     equals {equals} SHORTOP EQUALSETY TYPOSETY;
| |     {SHORTOP} EMPTY.
| |
```

```
| | DYAD NOMAD:
| |    equals {} DYAD NOMAD EQUALS;
| |    colon {} DYAD NOMADETY COLON;
| |    {SHORTOP} EMPTY.
| |
| | DYAD NOMAD EQUALS:
| |    colon {equals, colon, SHORTOP} EMPTY;
| |    {SHORTOP} EQUALS.
| |
| | DYAD NOMADETY COLON:
| |    equals {colon, equals, SHORTOP} EMPTY;
| |    {SHORTOP} COLON.
| |
| | SHORTOP EQUALSETY TYPOSETY:
| |    equals {SHORTOP} EQUALS;
| |    typo {} SHORTOP EQUALSETY TYPOSETY;
| |    {SHORTOP EQUALSETY} EMPTY.
| |
| | STICK:
| |    colon {STICKCOLON} EMPTY;
| |    {STICK} EMPTY.
| |
| | COLON:
| |    equals {} COLON EQUALS;
| |    differs {} COLON DIFFERS;
| |    divided {} COLON DIVIDED;
| |    {COLON} EMPTY.
| |
| | COLON EQUALS:
| |    colon {IS} EMPTY;
| |    {BECOMES} EMPTY.
| |
| | COLON DIFFERS:
| |    colon {} COLON DIFFERS COLON;
| |    {COLON} DIFFERS.
| |
| | COLON DIFFERS COLON:
| |    equals {COLON} DIFFERS COLON EQUALS;
| |    {ISNOT} EMPTY.
| |
| | COLON DIVIDED:
| |    equals {} COLON DIVIDED EQUALS;
| |    {COLON} DIVIDED.
| |
| | COLON DIVIDED EQUALS:
| |    colon {ISNOT} EMPTY;
| |    {COLON} DIVIDED EQUALS.
| |
| | DIFFERS:
| |    {differs} DYAD.
| |
| | DIFFERS COLON EQUALS:
| |    {differs, colon, equals, SHORTOP} EMPTY.
| |
```

```
| | DIVIDED:
| |     {divided} DYAD.
| |
| | DIVIDED EQUALS:
| |     {divided, equals} DYAD EQUALS.

level = FORMAT
|
| regime = POINT
| |
| | EMPTY:
| |     letter {save, reread} LETGITS;
| |     point {} POINT;
| |     digit {put} FIX;
| |     quote {} QUOTE STRING;
| |     plus {PLUS} EMPTY;
| |     minus {MINUS} EMPTY;
| |     comma {COMMA} EMPTY;
| |     open {OPEN} EMPTY;
| |     close {CLOSE} EMPTY;
| |     dollar {DOLLAR} EMPTY;
| |     cent {put} BRIEFCOMMENT;
| |     cross {put} STYLEIICOMMENT;
| |     typo {} EMPTY;
| |     other {error(2)} EMPTY;
| |     {EOF} EMPTY.
|
| regime = UPPER
| |
| | EMPTY:
| |     lowerletter {save, reread} LETGITS;
| |     upperletter {save} POINTETY UPPERTAGGLE;
| |     point {} POINT;
| |     digit {put} FIX;
| |     quote {} QUOTE STRING;
| |     plus {PLUS} EMPTY;
| |     minus {MINUS} EMPTY;
| |     comma {COMMA} EMPTY;
| |     open {OPEN} EMPTY;
| |     close {CLOSE} EMPTY;
| |     dollar {DOLLAR} EMPTY;
| |     cent {put} BRIEFCOMMENT;
| |     cross {put} STYLEIICOMMENT;
| |     typo {} EMPTY;
| |     other {error(2)} EMPTY;
| |     {EOF} EMPTY.
|
```

```
regime = RES
|
| EMPTY:
|     letter {save} TAGGLE;
|     point {} POINT;
|     digit {put} FIX;
|     quote {} QUOTE STRING;
|     plus {PLUS} EMPTY;
|     minus {MINUS} EMPTY;
|     comma {COMMA} EMPTY;
|     open {OPEN} EMPTY;
|     close {CLOSE} EMPTY;
|     dollar {DOLLAR} EMPTY;
|     cent {put} BRIEFCOMMENT;
|     cross {put} STYLEIICOMMENT;
|     typo {} EMPTY;
|     other {error(2)} EMPTY;
|     {EOF} EMPTY.
|
regime = STROP
|
| EMPTY:
|     letter {save, reread} LETGITS;
|     point {} POINT;
|     strop {} STROP;
|     digit {put} FIX;
|     quote {} QUOTE STRING;
|     plus {PLUS} EMPTY;
|     minus {MINUS} EMPTY;
|     comma {COMMA} EMPTY;
|     open {OPEN} EMPTY;
|     close {CLOSE} EMPTY;
|     dollar {DOLLAR} EMPTY;
|     cent {put} BRIEFCOMMENT;
|     cross {put} STYLEIICOMMENT;
|     typo {} EMPTY;
|     other {error(2)} EMPTY;
|     {EOF} EMPTY.
|
regime = RES
|
| TAGGLE:
|     letgit {save} TAGGLE;
|     {} TAGGLE END.
|
| TAGGLE END:
|     [commentbuffer] {append} POINTETY COMMENT;
|     [pragmatbuffer] {append} POINTETY PRAGMAT;
|     {reread} LETGITS.
|
```

```
| regime = POINT, RES, STROP
| |
| | LETGITS:
| |     letter a {ASYM} LETGITS;
| |     letter b {BSYM} LETGITS;
| |     letter c {CSYM} LETGITS;
| |     letter d {DSYM} LETGITS;
| |     letter e {ESYM} LETGITS;
| |     letter f {FSYM} LETGITS;
| |     letter g {GSYM} LETGITS;
| |     letter i {ISYM} LETGITS;
| |     letter k {KSYM} LETGITS;
| |     letter l {LSYM} LETGITS;
| |     letter n {NSYM} LETGITS;
| |     letter p {PSYM} LETGITS;
| |     letter q {QSYM} LETGITS;
| |     letter r {RSYM} LETGITS;
| |     letter s {SSYM} LETGITS;
| |     letter x {XSYM} LETGITS;
| |     letter y {YSYM} LETGITS;
| |     letter z {ZSYM} LETGITS;
| |     hjmotuvw {error(2)} LETGITS;
| |     digit {put} FIX;
| |     {} EMPTY.
|
| regime = UPPER
| |
| | LETGITS:
| |     lowerletter a {ASYM} LETGITS;
| |     lowerletter b {BSYM} LETGITS;
| |     lowerletter c {CSYM} LETGITS;
| |     lowerletter d {DSYM} LETGITS;
| |     lowerletter e {ESYM} LETGITS;
| |     lowerletter f {FSYM} LETGITS;
| |     lowerletter g {GSYM} LETGITS;
| |     lowerletter i {ISYM} LETGITS;
| |     lowerletter k {KSYM} LETGITS;
| |     lowerletter l {LSYM} LETGITS;
| |     lowerletter n {NSYM} LETGITS;
| |     lowerletter p {PSYM} LETGITS;
| |     lowerletter q {QSYM} LETGITS;
| |     lowerletter r {RSYM} LETGITS;
| |     lowerletter s {SSYM} LETGITS;
| |     lowerletter x {XSYM} LETGITS;
| |     lowerletter y {YSYM} LETGITS;
| |     lowerletter z {ZSYM} LETGITS;
| |     lowerhjmotuvw {error(2)} LETGITS;
| |     digit {put} FIX;
| |     {} EMPTY.
|
| regime = POINT, RES, STROP
| |
| | POINT:
| |     letter {save} POINT TAGGLE;
| |     {POINT} EMPTY.
| |
```

```
| | POINT TAGGLE:
| |    letgit {save} POINT TAGGLE;
| |    {} POINT TAGGLE END.
| |
| | POINT TAGGLE END:
| |    [commentbuffer] {append} POINTETY COMMENT;
| |    [pragmatbuffer] {append} POINTETY PRAGMAT;
| |    {reread, POINT} LETGITS.
|
| regime = UPPER
| | |
| | POINT:
| |    lowerletter {save} POINT LOWERTAGGLE;
| |    upperletter {save} POINTETY UPPERTAGGLE;
| |    {POINT} EMPTY.
| |
| | POINT LOWERTAGGLE:
| |    lowerletgit {save} POINT LOWERTAGGLE;
| |    {} POINT LOWERTAGGLE END.
| |
| | POINT LOWERTAGGLE END:
| |    [commentbuffer] {append} POINTETY COMMENT;
| |    [pragmatbuffer] {append} POINTETY PRAGMAT;
| |    {reread, POINT} LETGITS.
| |
| | POINTETY UPPERTAGGLE:
| |    upperletgit {save} POINTETY UPPERTAGGLE;
| |    {} POINTETY UPPERTAGGLE END.
| |
| | POINTETY UPPERTAGGLE END:
| |    [commentbuffer] {append} POINTETY COMMENT;
| |    [pragmatbuffer] {append} POINTETY PRAGMAT;
| |    {clear, error(7)} EMPTY.
|
| regime = POINT, UPPER, RES, STROP
| | |
| | POINTETY COMMENT:
| |    underscore {error(6)} BOLDCOMMENT;
| |    {} BOLDCOMMENT.
| |
| | POINTETY PRAGMAT:
| |    underscore {error(6), PRAGSYM} EMPTY;
| |    {PRAGSYM} EMPTY.
|
| regime = STROP
| | |
| | STROP:
| |    letter {save} STROP TAGGLE;
| |    {error(4)} EMPTY.
| |
| | STROP TAGGLE:
| |    letgit {save} STROP TAGGLE;
| |    {} STROP TAGGLE END.
| |
```

```
| | STROP TAGGLE END:
| |     [commentbuffer] {append} STROP COMMENT;
| |     [pragmatbuffer] {append} STROP PRAGMAT;
| |     {reread, error(4)} LETGITS.
| |
| | STROP COMMENT:
| |     strop {} BOLDCOMMENT;
| |     underscore {error(6)} BOLDCOMMENT;
| |     {} BOLDCOMMENT.
| |
| | STROP PRAGMAT:
| |     strop {PRAGSYM} EMPTY;
| |     underscore {error(6), PRAGSYM} EMPTY;
| |     {PRAGSYM} EMPTY.
|
| regime = POINT, UPPER, RES, STROP
| |
| | FIX:
| |     digit {put} FIX;
| |     typo {} FIX TYPO;
| |     {FIXNUM} LETGITS.
| |
| | FIX TYPO:
| |     digit {put} FIX;
| |     typo {} FIX TYPO;
| |     {FIXNUM} EMPTY.
| |
| | STRINGRETURN:
| |     {CHARROW} EMPTY.

level = UNIT, FORMAT
|
| regime = POINT, UPPER, RES, STROP
| |
| | QUOTE STRING:
| |     quote {} QUOTE STRING QUOTE;
| |     strop {} QUOTE STRING STROP;
| |     item {putitem} QUOTE STRING;
| |     control {} QUOTE STRING;
| |     other {error(11)} QUOTE STRING;
| |     {error(13)} STRINGRETURN.
| |
| | QUOTE STRING QUOTE:
| |     quote {quote} QUOTE STRING;
| |     typo {} QUOTE STRING QUOTE TYPO;
| |     {} STRINGRETURN.
| |
| | QUOTE STRING QUOTE TYPO:
| |     quote {} QUOTE STRING;
| |     typo {} QUOTE STRING QUOTE TYPO;
| |     {} STRINGRETURN.
| |
| | QUOTE STRING STROP:
| |     strop {strop} QUOTE STRING;
| |     {} STRINGESCAPE.
| |
```

```
| | STRINGESCAPE:
| |    {strop, error(12)} QUOTE STRING.
| |
| | BRIEFCOMMENT:
| |    cent {COMMENT} EMPTY;
| |    other {} BRIEFCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | STYLEIICOMMENT:
| |    cross {COMMENT} EMPTY;
| |    other {} STYLEIICOMMENT;
| |    {error(14), COMMENT} EMPTY.
|
| regime = POINT
| |
| | BOLDCOMMENT:
| |    point {} BOLDCOMMENT POINT;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT:
| |    letter {reset, match} BOLDCOMMENT POINT TAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT TAGGLE:
| |    letgit {match} BOLDCOMMENT POINT TAGGLE;
| |    underscore {} BOLDCOMMENT;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
|
| regime = UPPER
| |
| | BOLDCOMMENT:
| |    upperletter {reset, match} BOLDCOMMENT POINTETY UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT:
| |    lowerletter {reset, match} BOLDCOMMENT POINT LOWERTAGGLE;
| |    upperletter {reset, match} BOLDCOMMENT POINTETY UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
```

```
| | BOLDCOMMENT POINT LOWERTAGGLE:
| |    lowerletgit {match} BOLDCOMMENT POINT LOWERTAGGLE;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT POINTETY UPPERTAGGLE:
| |    upperletgit {match} BOLDCOMMENT POINTETY UPPERTAGGLE;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT UNDERSCORE:
| |    upperletter {} BOLDCOMMENT UNDERSCORE UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT UNDERSCORE UPPERTAGGLE:
| |    upperletgit {} BOLDCOMMENT UNDERSCORE UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
|
| regime = RES
| |
| | BOLDCOMMENT:
| |    letter {reset, match} BOLDCOMMENT TAGGLE;
| |    digiscore {} BOLDCOMMENT LETGITSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT TAGGLE:
| |    letgit {match} BOLDCOMMENT TAGGLE;
| |    underscore {} BOLDCOMMENT LETGITSCORE;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT LETGITSCORE:
| |    letgitscore {} BOLDCOMMENT LETGITSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
|
```

```
| regime = STROP
| |
| | BOLDCOMMENT:
| |    point {} BOLDCOMMENT POINT;
| |    strop {} BOLDCOMMENT STROP;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT:
| |    letter {reset, match} BOLDCOMMENT POINT TAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    strop {} BOLDCOMMENT STROP;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT TAGGLE:
| |    letgit {match} BOLDCOMMENT POINT TAGGLE;
| |    underscore {} BOLDCOMMENT;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT STROP:
| |    letter {reset, match} BOLDCOMMENT STROP TAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    strop {} BOLDCOMMENT STROP;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT STROP TAGGLE:
| |    letgit {match} BOLDCOMMENT STROP TAGGLE;
| |    strop {} BOLDCOMMENT ENDTEST;
| |    underscore {} BOLDCOMMENT;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
```

7. IMPLEMENTATION NOTES

Essentially the lexical analyzer described here is a finite state machine. Implementation techniques for finite state machines are well known, so we shall not discuss them here. Nevertheless there are some details, largely pertaining to the method of description of the lexical analyzer, that should get some attention. We discuss them below.

(1) The lexical analyzer consists of eight separate programs, one for each pair (level, regime). If more than one such program is needed, one might wish to combine coinciding parts of these programs. An obvious way to do this, is to turn common sets of states representing a submachine of the finite state machine into procedures or subroutines. Such sets of states are, for instance, the sets of states for the reading of short operators, strings and comments. The degree of interweaving can even be increased by combining "similar" states, such as the "EMPTY" states, into a single state. Pushing this interweaving too far, however, can easily lead to a loss of efficiency, because it requires a frequent inspection of the current regime and/or level.

(2) The "append" instruction can be implemented by copying the "buffer" to the "info" and subsequently clearing the buffer (as described in section 3). However, it can be seen that if the buffer is not empty, the only instructions executed on info and buffer are "save", "append" and "clear". So the concatenation of info and buffer behaves like a stack. Therefore we can implement them as:

```
string infobuff;
int sep;
```

where

```
infobuff[1 : sep]
```

represents the info, and

```
infobuff[sep+1 : upb infobuff]
```

represents the buffer. An "append" instruction now boils down to:

```
sep := upb infobuff;
```

(3) In the description of the machine the input is represented as a string, while in fact it most likely is a file. This can give some problems implementing the "reread" instruction. The "reread" instruction appends the contents of the buffer to the head of the input and clears the buffer. This instruction is only used in the format level programs (so if we only need the unit level programs, the problem does not exist). It can be implemented by copying the buffer to a special lookahead buffer and (after clearing the buffer) start reading from this lookahead buffer instead of the input file. It can easily be seen that as long as the lookahead buffer is not empty, no characters are "saved", i.e. put in the buffer. So one might be tempted not to copy the buffer at all and

use the buffer itself as the lookahead buffer. By doing so, however, the stack behavior of the concatenation of info and buffer will get lost, because it is possible that a "put" instruction must be executed with a nonempty buffer (it is possible to restore the stack behavior though, but this is rather tricky). So if the info and buffer are concatenated as described in (2), one should not use the buffer as the lookahead buffer.

REFERENCES

[1] HANSEN, W.J. and H. BOOM,
Report on the Standard Hardware Representation for ALGOL 68,
Algol Bulletin 40 (1976) 24-43.

[2] BELL, R.,
A Token Recognizer for the Standard Hardware Representation of ALGOL 68,
Algol Bulletin 41 (1977) 47-70.

[3] WIJNGAARDEN, A. VAN, et al. (eds.),
Revised Report on the Algorithmic Language ALGOL 68,
Acta Informatica 5 (1975) 1-236.

[4] HANSEN, W.J.,
Trouble Spots in the Standard Hardware Representation for ALGOL 68,
Algol Bulletin 42 (1978) 11-13.

APPENDIX 1: CHARSETS

All worthy characters (including both upper and lower case letters) plus all characters of the reference language (including some control characters) may occur in the input, i.e. the following characters are allowed:

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
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
0 1 2 3 4 5 6 7 8 9 . 10 " space ⍎ ∨ ∧ & ≠ < ≤ ≥ > /
÷ % ⌷ ⌊ ⌈ ⊥ ⌝ ~ ↓ ↑ + - = × * , ; ( ) | : [ ] @ ∘ $
¢ # ' _ newline newpage

The charsets applied in section 6 are defined below. A set of characters is denoted by a list of its elements surrounded by curly brackets, each element separated by a blank. Furthermore we use "+" for set union and "-" for set difference. The charset "item" is not defined; it must be equal to the set of all characters that are allowed as a string item.

```
at                = {@}
bus               = {]}
cent              = {¢}
close             = {)}
colon             = {:}
comma             = {,}
control           = {newline newpage}
cross             = {#}
differs           = {≠}
digiscore         = digit + underscore
digit             = {0 1 2 3 4 5 6 7 8 9}
divided           = {/}
dollar            = {$}
dyad              = nomad + {∨ ∧ & ≠ ≤ ≥ ÷ % ⌷ ⌊ ⌈ ⊥ ⌝ ↓ ↑ + -}
equals            = {=}
hexletter         = {a b c d e f A B C D E F}
hjmotuvw          = {h j m o t u v w H J M O T U V W}
letgit            = letter + digit
letgitscore       = letgit + underscore
letter            = lowerletter + upperletter
letter a          = {a A}
letter b          = {b B}
letter c          = {c C}
letter d          = {d D}
letter e          = {e E}
letter f          = {f F}
letter g          = {g G}
letter i          = {i I}
letter k          = {k K}
letter l          = {l L}
letter n          = {n N}
letter p          = {p P}
letter q          = {q Q}
letter r          = {r R}
```

```
letter s             = {s S}
letter x             = {x X}
letter y             = {y Y}
letter z             = {z Z}
lowerhjmotuvw        = {h j m o t u v w}
lowerletgit          = lowerletter + digit
lowerletter          = {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}
lowerletter a        = {a}
lowerletter b        = {b}
lowerletter c        = {c}
lowerletter d        = {d}
lowerletter e        = {e}
lowerletter f        = {f}
lowerletter g        = {g}
lowerletter i        = {i}
lowerletter k        = {k}
lowerletter l        = {l}
lowerletter n        = {n}
lowerletter p        = {p}
lowerletter q        = {q}
lowerletter r        = {r}
lowerletter s        = {s}
lowerletter x        = {x}
lowerletter y        = {y}
lowerletter z        = {z}
lowernoradletgit(n)  = lowerletgit - lowerradigit(n)
lowerradigit(1)      = {0 1}
lowerradigit(2)      = {0 1 2 3}
lowerradigit(3)      = {0 1 2 3 4 5 6 7}
lowerradigit(4)      = {0 1 2 3 4 5 6 7 8 9 a b c d e f}
minus                = {-}
nil                  = {∘}
nohexletter          = letter - hexletter
nomad                = {< > / × *}
noradletgit(n)       = letgit - radigit(n)
open                 = {(}
plus                 = {+}
point                = {.}
quote                = {"}
radigit(1)           = {0 1}
radigit(2)           = {0 1 2 3}
radigit(3)           = {0 1 2 3 4 5 6 7}
radigit(4)           = {0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F}
semicolon            = {;}
sign                 = {+ -}
stick                = {|}
strop                = {´}
sub                  = {[}
ten                  = {10}
tilde                = {~}
typo                 = {space .} + control
typoscore            = typo + underscore
underscore           = {_}
upperletgit          = upperletter + digit
upperletter          = {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}
```

APPENDIX 2: ERROR DIAGNOSTICS

error

1 Illegal character at the unit level.
Character skipped.
2 Illegal character at the format level.
Character skipped.
3 Unidentified point.
Point skipped.
4 Unidentified strop.
Strop skipped.
5 Bold preceded by underscore.
Underscore skipped.
6 Bold followed by underscore.
Underscore skipped.
7 Illegal bold word at the format level.
Bold word skipped.
8 No digits in fractional part of real denotation.
Zero inserted.
9 No digits in exponent part of real denotation.
Zero inserted.
10 No radix digits in bits denotation.
Zero inserted.
11 Illegal string item.
Character skipped.
12 Strop not followed by strop in character or string denotation.
Strop inserted.
13 End of file in character or string denotation.
Quote inserted.
14 End of file in comment.
Comment symbol inserted.