

**stichting
mathematisch
centrum**



AFDELING INFORMATICA

IW 3/73

MARCH

G. TEN VELDEN
MIXAL ASSEMBLY

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), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS (MOS) subject classification scheme: 68A10, 68A15

ACM-Computing Reviews-category 4.1, 4.2

Abstract

MIXAL is the assembly language for the (hypothetical) MIX computer, introduced by D.E. Knuth. In the present report, an assembly program for this language is described, based upon a slightly modified syntactic definition of MIXAL.

The assembler has been written first as an ALGOL 60 program. Care has been taken to provide it with a clear structure; in particular, the whole (12 pages) program contains only one goto statement. Next, the same algorithm has been implemented in MIXAL on the MIX computer. The latter version of the MIXAL assembler has been used as a MIXAL program to be assembled by the ALGOL 60 version. Both versions of the assembler have been reproduced in this report.

Contents

	page
1. Introduction.	1
2. Definition of MIXAL.	2
2.1 The syntax of MIXAL.	2
2.2 Remarks.	4
3. Reading, printing and recognition of basic syntactical units.	5
3.1 The low level reading procedure "read char".	5
3.2 Some auxiliary procedures, concerned with the output buffer.	6
3.3 The high level reading procedure "read synt unit".	7
3.4 The organisation of the symbol table.	9
3.5 The initialisation of the symbol table and the global variables.	14
3.6 Error messages.	15
4. Expressions and word values.	16
4.1 Expressions.	16
4.2 Word values.	17
5. The MIXAL instruction.	18
5.1 General outline.	18
5.2 Address value and field specification.	20
5.3 Future references.	21
6. The completion of the assembly process.	23
7. The ALGOL 60 procedure "mixal assembler".	25
8. MIXAL assembler described in MIXAL.	39
8.1 Table of error messages.	69
9. A loading program.	70
<u>References.</u>	74

1. Introduction.

In this report, an one-pass assembler for MIXAL is described, MIXAL being the assembly language for the (hypothetical) MIX computer, designed and described by Knuth [1].

The assembler has been written in ALGOL 60 as a boolean procedure, called "mixel assembler".

The assembler has been tested in the environment of a MIX simulator, written in ALGOL 60 by R.P. van de Riet [2].

The programs have been run on the EL X8 computer of the Mathematical Centre at Amsterdam, under supervision of the ALGOL 60 system, called MILLI [3].

The MIXAL system, consisting of assembler and simulator, accepts several MIXAL programs, textually being separated from each other by a job separator (a dollar symbol: \$), and which are translated and executed consecutively.

The interface of assembler and simulator consists of two parameters to procedure "mixel assembler", in addition to the boolean value, delivered by its procedure identifier. The first parameter "m" represents the memory of the MIX computer; actually it has to be an array, being declared with the bound pair 0:3999. All locations of this memory have to be set equal to zero, before calling procedure "mixel assembler"; those locations for which this is not the case, cannot be filled by the assembler (see section 5.3).

The second parameter "start address" is an output parameter of type integer, passing to the simulator the location at which the execution of the program has to be started.

At last, the boolean value, delivered by the procedure identifier of "mixel assembler", is the complement of the boolean variable "erroneous", which is local to procedure "mixel assembler"; "erroneous" indicates whether or not errors have been detected during assembly of the MIXAL program.

The assembler has to take into account that the simulator is a base ten machine (the bytesize being one hundred), with a special internal code for the characters, and a special operation code (a field specification (3:5) is coded as 35, being the value of $3 \cdot 10 + 5$, instead of $29 = 3 \cdot 8 + 5$).

Besides the ALGOL 60 version of the MIXAL assembler, we also have developed a version written in MIXAL, the algorithm being the same, except for some

very little details.

This MIXAL version has been reproduced in chapter 8, in the way it has been printed out by the MIXAL assembler in ALGOL 60, reproduced in chapter 7; thus we use this MIXAL version as a demonstration and a test case of the ALGOL 60 version. Finally, in chapter 9, we give a loading program, -appropriate to the MIX simulator, mentioned before- serving as a demonstration of the MIXAL version. As a thorough test for the MIXAL version, we applied the assembler to the MIXAL assembler in MIXAL itself, using however another MIX simulator, written in ELAN (being the assembly language of the EL X8 computer). This simulator satisfies the same specifications as the ALGOL 60 version (except for the memory size, which has been increased from 4000 to 10000), but it is much faster and can be operated by hand. The results of this test, however, have not been reproduced in this report.

2. Definition of MIXAL.

2.1 The syntax of MIXAL.

<MIX character> ::= <any available character on the MIX computer>
<number> ::= <digit> | <number><digit>
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
<symbol> ::= <identifier> | <digit><symbol>
<comment> ::= <empty> | <comment><MIX character>
<separator> ::= ; | <nocr> | " <comment><nocr>
<local forward> ::= <digit> F
<local label> ::= <digit> H
<local backward> ::= <digit> B
<local symbol> ::= <local forward> | <local label> | <local backward>
<undefined symbol> ::= <symbol>
<defined symbol> ::= <symbol>

<unary operator> ::= + | -
<binary operator> ::= + | - | * | / | // | :
<atomic expression> ::= <local backward> | <defined symbol> | <number> | *
<expression> ::= <atomic expression> | <unary operator><atomic expression> |
 <expression><binary operator><atomic expression>
<field specification> ::= (<expression>) | <empty>
<word value> ::= <expression><field specification> | <word value>,
 <expression><field specification>

<literal constant> ::= = <word value> =
<future reference> ::= <local forward> | <undefined symbol> |
 <literal constant>

<address part> ::= <future reference> | <expression> | <empty>
<index part> ::= , <expression> | <empty>
<address value> ::= <address part><index part>

<MIX instruction symbol> ::= ADD | ... | SUB
<pseudo instruction symbol> ::= EQU | ORIG
<constant instruction symbol> ::= CON | ALF
<END instruction symbol> ::= END
<MIX machine instruction> ::= <MIX instruction symbol>
 <address value><field specification>
<pseudo instruction> ::= <pseudo instruction symbol><word value>
<char> ::= <MIX character>
<character code> ::= . <char><char><char><char><char>
<MIX constant instruction> ::= CON <word value> |
 ALF <character code>
<proper instruction> ::= <MIX machine instruction> |
 <pseudo instruction> |
 <MIX constant instruction>
<proper END instruction> ::= <END instruction symbol> . <word value>

```
<label> ::= <undefined symbol> | <local label>
<MIXAL instruction> ::= <proper instruction> | <label> ; <MIXAL
                        instruction> | <separator> <MIXAL instruction>
<END instruction> ::= <proper END instruction> | <label> : <END
                        instruction> | <separator> <END instruction>
<MIXAL body> ::= <empty> | <MIXAL instruction> <separator> <MIXAL body>
<MIXAL program> ::= <MIXAL body> <END instruction> <separator>
```

2.2 Remarks.

The language MIXAL as defined in the preceding section, differs from the language defined by Knuth on a few points.

As one of the purposes has been to design a version of MIXAL being layout independent as much as possible (i.e. the position of any syntactical unit on the card or papertape should be of no significance), some additional punctuations have been introduced to prevent ambiguities; the main case being the colon following each label, the other cases being the quotation mark preceding comment, and the point in character code preceding the five MIX characters.

Layout independence means that spaces freely may be inserted into the MIXAL text; however, there are some exceptions:

- a) in character code, a space certainly is not layout, but stands for a space as MIX character,
- b) in numbers and symbols spaces may also be inserted as layout, but not two (or more) spaces consecutively.

The latter exception has been made as (in some situations) two symbols may succeed each other (or a symbol and a number), without being separated by another syntactical unit. Thus we use two (or more) spaces as separator between symbols and/or numbers.

A second difference is concerned with labels. The layout independence has made it possible to allow more than one label, preceding a proper instruction, each of these labels being equivalent to the same value.

As these changes do not essentially affect MIXAL as defined by Knuth, it suffices to refer to [1] for the meaning and use of the assembly language MIXAL.

3. Reading, printing and recognition of basic syntactical units.

We distinguish between two levels of basic reading procedures, the first one being concerned with the reading and printing of single MIX characters, and the second one combining these MIX characters to basic syntactical units, which are analyzed more closely.

3.1 The low level reading procedure "read char".

This procedure reads the next character of the MIXAL program from the input medium. Both the procedure identifier of "read char" and the global variable "char" get as value the internal code of this MIX character.

Furthermore, the listing of the MIXAL program -each line generally consisting of the address and code, corresponding to the MIXAL instruction, the line number, and the text of the MIXAL instruction- is arranged by "read char", storing the characters read into a buffer (the array "line"), and printing out this buffer at the right moment, i.e. generally after another part of the assembler has produced the code corresponding to the instruction of this line, and has placed it also in the buffer. The boolean variables "text" and "code" make it possible to indicate this moment outside procedure "read char". They indicate whether some text has been completed for printing, and whether address and code have been produced. Thus, "read char" builds up a line in the buffer, organized with two pointers "pos" and "last pos", running from zero upwards, the first one pointing at the first free position of the buffer, the latter one pointing at the lastly occupied position.

It will be clear that reading a <nocr> (new line carriage return, or new card) needs a special treatment, as, at the reading of this separator, the buffer generally may not be printed out (address and code may have to be determined yet). The <nocr> itself is not stored into the buffer, as it is not a MIX character actually, but, as an indication that a <nocr> just has been read, the pointer "pos" gets the value "-1", and the boolean variable "text" gets the value "true" (the latter being explained already).

When "read char" is called again, we may assume that address and code have been produced, if this were necessary. The value "true" for "text" (always) means that we may print this line before continuing, and the value "-1" for "pos" means that we have to start a new line at position zero, and that we also can store the new linenumber into the buffer.

During the initialisation of the assembler, when all MIX instruction symbols have to be read and stored into the symbol table, the high level reading procedures are used, which call "read char" in their turn. The text for the initialisation, however, is not read from an input medium, but is contained in the ALGOL 60 program by means of a string. Giving the boolean variable "from string" the value "true", the reading from this string is effectuated by means of the standard procedure "STRINGSYMBOL", called in "read from string", which delivers the representations of characters of a string [3].
remark:

When a dollar \$ (the job separator, mentioned in the introduction) is read, obviously the END instruction of the MIXAL program is missing. Having reported this error, we jump out of all nested procedure calls, back to the first statement of the assembler, in order to treat the next MIXAL program. We mention this jump explicitly here, as this is the only goto statement, appearing in the whole program.

3.2 Some auxiliary procedures, concerned with the output buffer.

The printing of the characters, contained in the buffer, is done by procedure "print line". This procedure prints the contents of the buffer (using the pointer "last pos"), and fills the buffer with spaces again.

In the first place, procedure "read char" is responsible for the filling of the buffer with text.

Furthermore, a procedure "buffer" is available, used for storing numbers into the buffer (used for address, code and linenumber). Procedure "buffer" has four formal parameters: "p", "n", "m" and "value". The first parameter "p" indicates the position in the buffer at which the units of the number, contained in "value", are placed. There is a possibility to insert spaces between the digits of the number, by means of the parameters "n" and "m".

The digits of the number are divided into "abs(n)" groups of "abs(m)" digits; between each two successive groups one space is inserted. The sign of "n" indicates whether or not (positive or negative, respectively) a sign has to be placed preceding the number; the sign of "m" indicates whether leading zero's are printed, or replaced by spaces (positive or negative, respectively).

Finally, a procedure "buffer symb" may be called, storing symbols from the symbol table into the buffer. Its first parameter indicates the place of the symbol in the symbol table (see section 3.4.), the second parameter indicates the position in the buffer at which the first character of the symbol is placed, and the third parameter is an output parameter, delivering the first free place in the buffer, following the last character of the symbol. Procedure "buffer symb" actually has been used for special purposes concerned with the completion of the assembly process (see section 6).

3.3 The high level reading procedure "read synt unit".

Procedure "read synt unit" combines a number of MIX characters to one basic syntactical unit. Most of the basic syntactical units consist of one MIX character only (*, :, , , , (,), =, +, -, *, / and :); they need no further analysis. The basic syntactical units, consisting of more than one character are:
a) number, b) symbol, c) separator and d) special divisor (//).

a) Reading a number, we don't know whether we are actually reading a number or a symbol after all, before we have read it completely. That is why we treat a number as if it were a symbol; this means that the digits temporarily are stored into the symbol table (see b.). In the boolean variable "contains letter" we keep up whether at least one letter has been read; if this is the case, we are concerned with a symbol (discussed below), otherwise we have read a number, of which the value is calculated by means of the digits stored in the symbol table. This value is assigned to the global variable "value of number", and the procedure identifier of "read synt unit", as well as the global variable "synt unit", get the constant value of "number" (value 102).

b) Reading a symbol, the letters and digits are stored meanwhile into the symbol table, in a way to be considered later on (3.4.). When all letters and digits have been read, procedure "look for symbol" looks up the symbol just read among the symbols which have been stored already in the symbol table. If this symbol is not found, a new information cell in the symbol table is created, containing the letters and digits of the symbol already. Thus, the new symbol has been joined to the symbol table by procedure "look for symbol". By means of the function designator "look for symbol", the place in the symbol table is passed on to the variable "last symbol". Besides, the variable "type" gets more information about the kind of symbol just read. After all, the procedure identifier of "read synt unit" and "synt unit" get the constant value of "symbol" (value 103).

c) The separator is read entirely, which means that comment is skipped, up to the first <nldr>, by means of procedure "read char", and the procedure identifier of "read synt unit" (as well as "synt unit") gets the constant value of "separator" (value 101). Having read a semicolon as separator, the variable "text" is set on "true", as we expect that one MIXAL instruction has been completed, and the assembler will produce address and code, before the next character is read (at which moment the buffer will be printed out). The variable "pos" is left unchanged in this case; so succeeding characters will stay at the right position of the line, however, this will be a next line.

d) The reading of a special divisor (//) does not lead to any difficulty.

In most cases, the reading of one syntactical unit consists of the reading of one character, but, in some cases, we know that we have read one whole syntactical unit, just when we have read the first character of the following syntactical unit (this is the case when reading numbers, symbols and the division operator /).

Thus we arrive at the impossible situation that we sometimes have read already the first character of a following syntactical unit, and in other cases we have not. This can be resolved by consequently reading one

character too many in all cases, but this has an unpleasant effect on the synchronisation of possible error messages. We have chosen the following solution:

Instead of unnecessarily reading one character too many, we insert one extra character (a space as layout character), by simply giving the variable "char" the constant value of "space" (value 66) in those cases where we do not need to read the first character of the following syntactical unit. As, at the reading of the next syntactical unit, spaces are skipped beforehand by procedure "read synt unit", there will be no harmful effect of this insertion.

The main program of the assembler, and the expression reading procedures, mainly read the MIXAL program by means of this procedure "read synt unit", which means that one need not worry, at that level, about the listing of the MIXAL program, the insertion of, and search for symbols in the symbol table, and the reading of numbers.

3.4 The organisation of the symbol table.

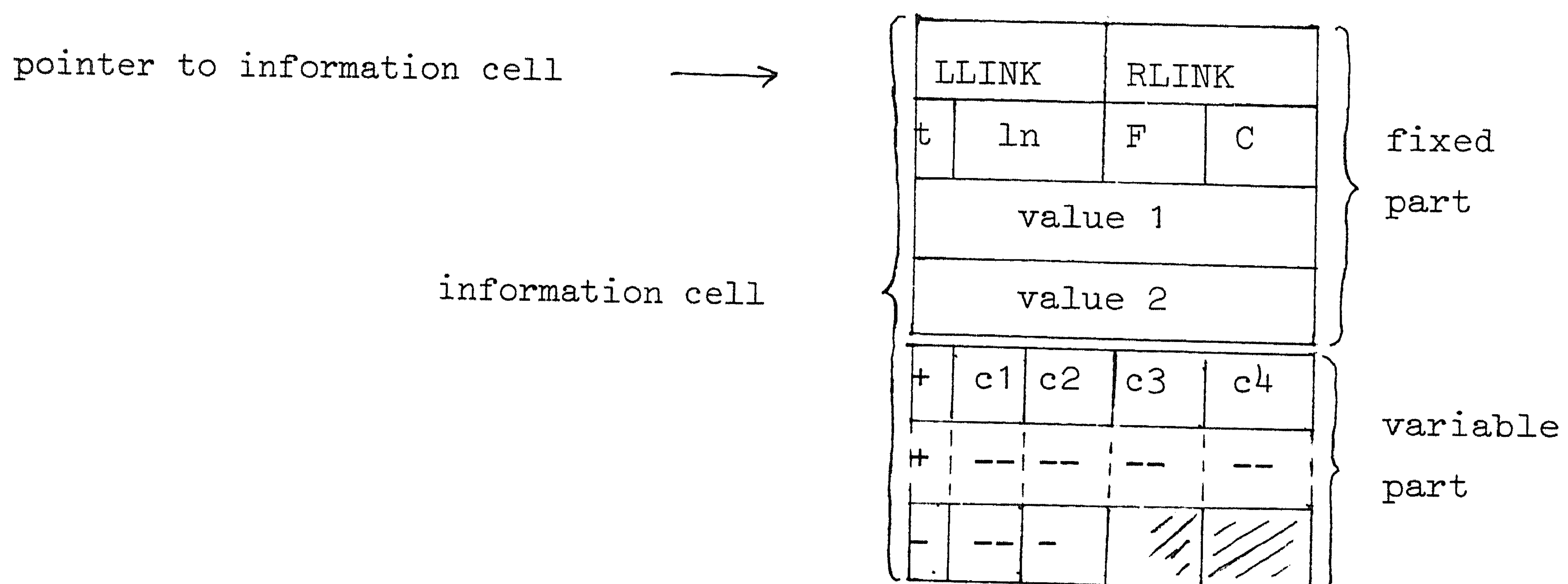
The symbol table is realised as an integer array "t", and consists of a set of information cells, each of which containing all information about one symbol. An information cell consists of a number of contiguous elements of "t", the address of the first element being used as the reference (pointer) of this information cell; the variable "last symbol" is used as a pointer to it. Local symbols, however, are not stored into this symbol table as other symbols are, but are detected explicitly in procedure "read synt unit", indicated by "type", while in these cases the variable "last symbol" gets as value the value of the digit, contained in the local symbol; this value can be seen as a pointer to an element of the arrays "h", "b", "f" and "defined local backward", into which the information concerning local symbols is stored. The set of information cells may be divided into three contiguous parts, the first two being filled during initialisation with the MIX instruction symbols, and with the constant instruction symbols, the END instruction symbol, and the pseudo instruction symbols. Into the third part, all other symbols, introduced by the MIXAL programmer, are stored. The boundary between the first

two parts is given by means of the variable "last inst" being a pointer to the information cell of the lastly stored MIX instruction symbol.

For the symbols contained in the second part, explicit pointers are used, viz.: "alf", "con", "end", "equ" and "orig".

Concerning the internal construction of information cells, there exists no difference between the distinct parts, as each instruction symbol (e.g. ADD, CON or END) may be used by the MIXAL programmer as its own symbol. For such a symbol there exists only one information cell (in the first or second part of the symbol table) into which both the information with respect to the MIX instruction, and the information concerning the symbol, used by the programmer as its own, are stored.

An information cell consists of two parts, a fixed part and a variable part, the latter being used to contain the internal codes of the letters and digits, the particular symbol consists of.



The last element of the variable part is made negative, as opposed to the other elements, indicating the end of the alpha numerical information. This element may contain fewer than four characters (the others always contain four of them), in which case it is filled up by dummy characters.

Next, the fixed part of the information cell is dealt with closely. The fields F and C are concerned with MIX instruction symbols only (the first part of the symbol table). F contains the standard field, and C the code of

the corresponding MIX instruction. These fields are filled during initialisation. The field *ln* is used to store the linenumber of the line, on which the symbol has occurred as label (has got a value). The field *t* (type) indicates whether the symbol in question is defined or undefined (negative or positive value, respectively), i.e. whether or not the symbol has become equivalent to some value by means of an occurrence as label.

For the fields value 1 and value 2, we have to distinguish between undefined and defined symbols, the status being indicated by the value of the field *t* as described before. At information cells of undefined symbols, the field value 1 may be used to hold a pointer to the information cell of the symbol, which directly precedes the symbol in question, both symbols being used as label preceding the same MIXAL instruction. If this preceding label happens to be a local label, the pointer to it has been inverted and decreased by one, forcing a negative value, indicating in this case a reference out of the symbol table.

In case the symbol in question is not preceded by another label, the initial value of zero (stored in procedure "look for symbol") is replaced by the value 9999, indicating an empty pointer (N.B. this value neither may be a reference into the symbol table, nor a reference to a local symbol). The field value 2 may hold (if the symbol has occurred as future reference without being defined already) a pointer to that memory address of the MIX computer, at which the instruction has been placed, the address part of which contained the textually last future reference of the symbol in question. When the symbol has not (yet) occurred as future reference, value 2 holds the empty pointer (N.B. in this case the value 9999 may not be a valid MIX address).

For local symbols, the fields value 1 and value 2 also exist, but they are recorded in the respective elements of the arrays "h" and "f".

Concerning defined symbols, both pointers in the fields value 1 and value 2 have become irrelevant, but merely the value, the symbol recently has become equivalent to, is important. This value is split up into two parts, called the head and the tail, as the numerical capacity of one element of "t" would not be sufficient to store a word value. The head, stored into the field value 1, has the value "abs (word value) ÷ 10¹⁶", and the tail, stored into the

field value 2, has the value "sign (word value)*(abs(word value)-head *10⁶)".

The remarks concerning defined symbols do not pertain to defined local symbols, as the fields value 1 and value 2 may be used again at a next occurrence of the local symbol as label. So, a distinct field is used for storing the value of the local symbol. This field is recorded in the array "b", and in an auxiliary array "defined local backward" we keep up in a boolean value whether the local symbol in question has occurred as local label at least once.

The search technique in the symbol table is a binary search technique in a tree-structured symbol table. To make a binary search applicable, we first have to define an order relation between symbols. The alphabetical ordering turned out to be an appropriate order relation for our purpose. As we also have to deal with digits, we have extended this ordering, defining the digits ordered naturally, preceding the letters (e.g. MIX, MIX 1, MIX 10, MIX 9, MIXAL, in this order). The tree structure of the symbol table is defined as follows:

- 1) Each node (actually an information cell) has at most two subtrees (called the left and right subtree, which are trees satisfying this definition), which are referred to by the pointers, contained in the fields LLINK and RLINK; if a subtree is not present, the corresponding field contains an empty reference (0).

- 2) All nodes in the left subtree of some node contain information cells of symbols, alphabetically preceding the symbol of this node, and all nodes of the right subtree contain information cells of symbols, succeeding the symbol of this node.

Due to this organisation, it is possible, after one comparison of an arbitrary symbol with the symbol contained in the root of the tree, to decide, in which of the two subtrees we have to search for this arbitrary symbol (if the symbols, just compared happen to be not the same). At each next comparison, we may exclude a whole class of symbols from the searching process. Trying to search for a symbol, which not yet has been inserted into the tree, we eventually come out at an empty pointer. At this point we know that an

information cell of this symbol does not occur in the symbol table, and we easily may join the new symbol to the symbol table, by replacing the empty pointer by a real pointer to the information cell (the variable part of which existing already) of the new symbol. Having filled in two empty pointers in the fields LLINK and RLINK of the newly created information cell, and initial values for the other fields (all within procedure "look for symbol"), we have extended the symbol table by one symbol, while the conditions, the tree structure satisfies, are still fulfilled. For, if we have to search for the recently inserted symbol, starting at the root of the tree, we do follow the same path through the tree as before, up to the node which recently has contained the empty pointer (nothing has been changed until so far). As this empty pointer has been replaced by the real pointer to the information cell of the symbol, we are searching for, we do not end in the blind alley now, but do find the appropriate information cell.

Finally, we direct our attention to the advantages gained by using the binary search. It may be clear that, offering new symbols to be inserted in alphabetical order, only branches of the tree to the right will originate, in which case we certainly do not gain any advantage over the linear search method. The order in which the MIXAL programmer introduces his symbols cannot be predicted, but, fortunately, we are able to introduce the MIX instruction symbols during initialisation, ordered in such a way, that the base of the tree will be filled homogeneously. The number of MIX instruction symbols amounts to 154 (including ALF, CON, etc.). Given a tree of seven levels, we can store into it $\sum_{i=0}^6 2^i = 127$ symbols. The remaining 27 symbols easily can be stored into the eighth level. This means that each of these instruction symbols can be found after eight comparisons at the most; the average number of comparisons amounting to:

$$1/154 * (\sum_{i=0}^6 2^i * (i+1) + 27 * 8) = 1001/154 = 6.5.$$

Thus we have made a considerable profit on the linear search method, which needs an average number of $154/2 = 77$ comparisons for an instruction symbol.

3.5 The initialisation of the symbol table and the global variables.

During initialisation, characters are read from the string, contained in procedure "read from string", as discussed already in section 3.1. This string satisfies the following syntactical structure.

```

<MIX characters> ::= |*;:.( ] = +-*/*: azbfh
<standard field and code> ::= <number (2.1)>
<MIX instruction definition> ::= <empty>|
    <MIX instruction symbol (2.1)><standard field and code>
<MIX instruction definitions> ::= <empty> | <MIX instruction definition >
    < separator > < MIX instruction definitions >
<constant instruction symbols> ::= ALF  CON
<pseudo instruction symbols> ::= EQU  ORIG

<initial string> ::= <MIX characters>
    <MIX instruction definitions>
    ( < constant instruction symbols >
    <END instruction symbol (2.1)>
    <pseudo instruction symbols> )

```

First, we read the MIX characters by means of procedure "read char", and assign their internal code to the variables:

"underlining", "bar", "space", "asterisk", etc. These variables, strictly speaking, are constants, introduced to make the program more readable.

Having assigned to the other constants and variables their (initial) values, and having filled the outputbuffer "line" with spaces, we continue with filling the first part of the symbol table with the MIX instruction symbols. Having constructed explicitly the root of the tree, the symbol table consists of (viz. an information cell with initialized fields LLINK, RLINK, t, value 1 and value 2), the MIX instruction definitions are read by means of procedure "read synt unit ", the insertion to the symbol table being automatically now. Following each symbol, we read the standard field and code, the value of this number being: (standard field * 100 + code), and record both quantities in the fields F and C respectively of the information

cell in question. Having read all MIX instruction definitions, terminated by a paren, the first part of the symbol table is completed by assigning to "last inst" the pointer to the last information cell.

In addition, the next five symbols are read, and pointers to their information cells are recorded in the variables "alf", "con", "end", "equ" and "orig", respectively.

Finally, having checked whether we have reached the closing parenthesis, we assign to the boolean variable "from string" the value "false" (the effect of which being discussed already in section 3.1), and finish the initialisation process.

3.6 Errormessages.

When during the assembly process an error is detected, procedure "error" is called. The actual parameter of "error" is a string, describing (in the English language) the kind of error detected. In procedure "error", first all characters contained in the outputbuffer are printed out by means of a call of "print line". Next, at a new line, the actual parameter of error is printed out. The pointer "pos" (3.2), holding the position in the buffer, remains unchanged; thus, following characters (when reading on) will be printed out at their correct position, some lines lower however.

We already have mentioned the synchronisation of errormessages (3.3).

Synchronisation means that the message is printed, textually immediately following those MIX characters, the errormessage is concerned with. In some cases however, we have read one character too many (see 3.3) when detecting an error, or even worse, a whole syntactical unit (including another following character). The latter phenomenon we meet at the reading of expressions (section 4). Reading too much means, in this context, that some text, belonging to following syntactical units, has been read already (while not being considered), merely to determine whether or not a whole unit (e.g. an expression) has been completed. In these cases, a possible errormessage will not be given in the right context.

Having detected and reported an error, the assembly process generally can be continued, and other errors may be detected and reported in the same way.

4. Expressions and word values.

4.1 Expressions.

As appears from the definitions in section 2.1, no priority rules exist for arithmetic operators in expressions; this means that operations are applied in order from left to right. Together with the fact that parenthesis are not allowed within expressions, this makes expressions very easy to evaluate; we do not need a stack for storing intermediate results, as these results immediately are used as left operand of an operator, the right operand being a single atomic expression, the value of which easily and directly is determined by means of procedure "read atomic expression". Of the atomic expressions, only symbols give rise to some further investigation. Symbols in expressions only may occur as defined symbol, or as defined local symbol, the values of which are denoted in the combined field value 1 and value 2, and in the array "b", respectively. The other possible occurrences of symbols are not admitted here, so they cause an error message, viz. "undefined symbol in expression", or "undefined local backward". How we can determine whether or not a symbol has been defined, has been explained already in section 3.4. The other possibilities for atomic expressions are very easy to deal with. When we are concerned with a number, we find its equivalent value in the global variable "value of number"; at this point it is important to observe that we have not read too much, as the following syntactical unit may be a number, the value of which would destroy the value of the global variable "value of number".

When we are concerned with an asterisk (called "self"), we only have to deliver the value of the location counter "lc" (see section 5).

For atomic expressions, we have to check, within procedure "read atomic expression", whether the equivalent value is not too large (exceeding the MIX word value). When executing arithmetical operations, by means of procedure "arithm", also capacity overflow may occur. In these cases an error message is given by the procedures concerned, viz. "overflow (atomic expression)", and "overflow (+)", "overflow (*)", etc., respectively, and, as resulting value, zero is delivered. The consequence of these tests,

within "read atomic expression" and "arithm", is that in procedure "read expression" only valid intermediate results occur, and no test on capacity overflow is necessary.

It is clear now that an expression can be evaluated in a straightforward manner by means of the procedures "read atomic expression" and "arithm". For reasons, later on to be explained, the first syntactical unit of an expression always has been read already when calling "read expression". Within this procedure we read the next syntactical unit and whenever this is an arithmetic operator, we go on reading an atomic expression, and executing the operation; otherwise, when no operator is read, we have done with the expression, and then, we did read one syntactical unit too much. This syntactical unit cannot be a syntactical unit belonging to the following MIXAL instruction, provided that two distinct MIXAL instructions are separated by a separator.

4.2 Word values.

A word value is used to denote the contents of one MIX word. According to the definition, we can describe a word value as:

$E_1 (F_1), E_2 (F_2), \dots, E_n (F_n)$, $n \geq 1$ and E_i and F_i being expressions for $i=1, \dots, n$.

The value E_i is meant to be stored into the fields of a certain MIX word, specified by (F_i) , for $i=1, \dots, n$ consecutively. The field specification (F_i) may be omitted, in which case a standard field (0:5) applies. The desired result, delivered by procedure "read w value", is the eventual value which would remain at the memory address CON when executing the next fragment of a MIXAL program:

STZ CON; LDA C1; STA CON(F_1); ...; LDA Cn; STA CON(F_n), where $C_1, \dots, C_n$ denote the addresses at which the values of $E_1, \dots, E_n$ are stored respectively. Using literal constants, the fragment also can read:

STZ CON; LDA = E_1 =; STA CON (F_1); ...; LDA = E_n =; STA CON (F_n).

The F_i ($i=1, \dots, n$), being the value of a field specification, must be of the form $L_i \cdot 10 + R_i$ (denoted in MIXAL as $L_i:R_i$), where L_i and R_i have to satisfy: $0 \leq L_i \leq R_i \leq 5$.

Reading a word value, one syntactical unit too many is read (as it was the case in "read expression"), when testing whether or not we have done with the word value.

Also the word value (just as an expression) is followed by another syntactical unit which does not belong to the following MIXAL instruction. This means that we never incorrectly read (even when errors are detected) syntactical units belonging to following MIXAL instructions, at the level of the procedures "read expression" and "read w value".

5. The MIXAL instruction.

5.1 General outline.

An interesting feature of the MIXAL instruction is the fact that more than one label may precede the proper instruction. The first syntactical unit of an instruction has to be a symbol, being a label or an instruction symbol, the latter being the first syntactical unit of the proper instruction.

Finding out which of these two kinds of symbols we are concerned with, we have to consider the context in which the symbol occurs, as each instruction symbol also may be used as label. Checking whether the next syntactical unit is a colon, we are able to recognize a label as such.

However, if no label occurs, we thus have read the first syntactical unit belonging to the address value (if not empty), to the word value, or to succeeding character code.

When this last syntactical unit happens to be a symbol as well (this is very well possible), the global variables "last symbol" and "type", originally referring to the instruction symbol, have been overwritten; unless we have saved the original value of "last symbol" by means of a local variable of procedure "read mixal program" (actually called "inst"), we would have lost all information about the instruction symbol.

Another consequence of this looking for a colon is that the first syntactical unit of an expression, possibly succeeding the instruction symbol, has been read already, when we eventually come to the processing of the expression. (see section 4.1).

Now, the question arises, whether it is possible to give a symbol a value (make the symbol a defined symbol), immediately following the recognition of the symbol occurring as label. Unfortunately this value depends on the kind of proper instruction following. If this proper instruction happens to be the EQU pseudo instruction, the equivalent value for the label(s) is (are) denoted by the word value which follows EQU, otherwise the label(s) will become equivalent to the value which the location counter "lc" has at the moment

Consequently, we first have to read all labels preceding the proper instruction, including the instruction symbol (and the next syntactical unit to recognize the instruction symbol as such), before determining the equivalent value for these labels. Meanwhile, we retain all labels by linking them; each label is made to refer to its predecessor, and the first one refers to nothing, i.e. contains an empty pointer.

To hold this reference, the field value 1 of the concerning information cells in the symbol table is used. We already have seen that, for local labels, an element of the array "h" corresponds to the field value 1, and that a negative reference stands for a reference to "h" and a positive reference stands for a reference into the symbol table; this holds both for references contained in the symbol table and in the array "h" as well. Thus we have chained all labels, and a pointer to the last occurring label is kept in the variable "last label". However, we carefully have to avoid circular lists, originating when one symbol occurs more than once as label preceding the same instruction.

Symbols, which have been inserted in a chain as described above, can be recognized by the following two properties for their information cells:

- 1) The field t is positive (irrelevant in the case of local labels), which means that the symbol has not yet become a defined one;
- 2) The field value 1 has a value, unequal to the initial value zero, which means that the symbol has occurred as label, as value 1 contains a reference (possibly an empty reference).

Subsequently, we treat the proper instruction entirely, and finally we update the chained list of labels by following the references, while storing into the information cells the equivalent value of the symbols, and making

the fields t negative, to indicate the defined status of the symbols (by means of procedure "update").

If one of the chained symbols occurs in some expression, contained in the proper instruction, it has not yet the defined status, and an error message follows. In the case of the EQU pseudo instruction, this error message is appropriate, as the value of a symbol may not depend on itself. In the other cases, the value of such a symbol could have been known. For congruence reasons with the previous case, we decided however, to maintain the general scheme of treatment; the advantage being that one rule can be given, stating, at which moment symbols get a value, viz. after the entire instruction has been dealt with.

Some examples may serve as illustration:

- 1) F: EQU F+10; In this case, the value of F would depend on itself; we rightly get an error message
- 2) F: ADD F+10; An error message is given, as the value of F has not yet been assigned to F, when evaluating the expression F+10, although it is known.
- 3) F: ADD **10; The alternative construction of 2), using an asterisk which has the same value as F gets afterwards.

At the moment the equivalent value for the labels is known, we keep this value in the variable "equiv". When ultimately the procedure "update" is called, we pass on the value of "equiv" as actual parameter of "update".

5.2 Address value and field specification.

The following two facts, mentioned before, are recalled. Firstly, the pointer to the information cell, belonging to the instruction symbol, is held in the variable "instr", local to procedure "read mixal program". This pointer is passed on to procedure "read mix machine inst" as actual parameter. The first action of procedure "read mix machine inst" will be then to fetch the standard field and code of the instruction from the fields F and C of the information cell concerned, and to assign these values to the variables "f part" and "c part", respectively.

Secondly, the first syntactical unit of address value and field specification has been read already at this moment. We start the actual processing by testing whether the address part contains a future reference, analyzing the present syntactical unit. When a future reference does occur, the variable "a part" will get the value of a reference, being discussed in the following section; otherwise, if the address part doesn't happen to be empty -i.e. if the present syntactical unit is not a comma (indicating an index part), nor a left parenthesis (indicating a field specification), nor a separator (indicating the absence of all quantities)-, we may expect an expression, treated by means of procedure "read expression", the value being assigned to the variable "a part". Of the quantities indexpart and field specification, easily can be determined now whether or not they are empty.

An empty index part gives rise to the assignment of the value zero to the variable "i part", and an empty field specification causes the standard field remaining in the variable "f part".

After all, the word value which has to be stored at the address indicated by "lc", can be assembled easily by means of the variables "a part", "i part", "f part" and "c part", and actually it is stored into the MIX memory by means of procedure "produce" (which also produces the output of address and code by means of the outputbuffer).

5.3 Future references.

Syntactically, a future reference may occur only as address part of a MIX machine instruction. The value for this future reference will become known in the future, when treating following instructions. The intention is, that, for the instruction in which the future reference occurs, code is produced as usual, except for the address part, which will be filled in later on: at the moment that the corresponding (local) label gets a value by means of procedure "update"; for literal constants it means at the end of the program (see section 6).

It is evident, that a symbol may occur as future reference more than once. This means that we have to update a number of MIX addresses, when the symbol in question gets its value. As for all these addresses it holds, that the

address part of the code has not yet been filled in, we can use these two bytes for linking the MIX addresses; thus we chain all addresses corresponding to a future reference of the same symbol. The beginning of this chain is kept in the symbol table, in the field value 2 of the information cell (for local forwards corresponding with an element of array "f"), and the chain is ended by means of an empty reference.

In the same way, a chain of literal constants is created, the access to it being kept in the variable "last constant". The difference between the chain of literal constants, and a chain of symbolic future references, is, that each MIX word in the first chain ultimately will obtain an unique address part (viz. the address at which the constant word value is stored by the assembler), while each MIX word of one other chain gets the same value in its address part. Chaining the MIX addresses, at which literal constants occur, is not sufficient; also the corresponding word values have to be retained somewhere until the end of the assembly process, when they are treated. We use the end of the array "t" for this purpose. For each word value to be stored, two consecutive elements of array "t" are used, identically to the way of coding a word value in the fields value 1 and value 2.

Describing the chaining of labels, we have seen that overwriting of references carefully has been avoided. Also in the case of future references, overwriting may take place, possibly leading to disastrous effects (such as an infinite loop of the assembler).

As we are able indeed, by means of the ORIG pseudo instruction, to start writing at each address we want, we necessarily have to protect ourselves against overwriting any address which contains a reference. As these addresses are hardly distinguishable from other ones - we would have to follow all references, starting at many places in the symbol table, and at the variable "last constant", to determine whether the address in question forms part of one of the chains - we decided to forbid each attempt to write on an address, which does not contain the value zero, the initial value of all addresses (see introduction).

We are aware of the fact that we unfortunately also forbid innocent

overwriting, but that is the price we have to pay, when we decide to desist from the very expensive algorithm, described above.

When the assembler detects an address, containing a value, unequal to the initial value zero, the error message "address already used" is given. Meeting an address, which does contain the value zero, necessarily one of the four following situations occurs:

- 1) the address never has been used,
- 2) the address has been filled by means of CON 0
- 3) the address has been filled by means of ALF .000000
- 4) the address has been filled by means of NOP 0

In the first three cases, the address does not contain a future reference, so there exist no danger (although it is a little inconsistent to do permit the overwriting in some cases). The address part of the instruction NOP may contain a future reference, previously occurred at address zero (in the address part of the NOP instruction, a reference to address zero is filled in). However, this may occur only once in a MIXAL program; thus, when we retain this address in a special variable "dangerous", we protect ourselves against overwriting of references, when testing whether a certain address contains the value zero, and this address is not the dangerous one.

6. The completion of the assembly process.

Having detected the END instruction, and having read the following word value, and assigned its value to the variable "start address", we have to deal with those symbolic future references, which have not yet been treated, and with the literal constants, by means of procedure "insert constants". At first, all information cells of local symbols and ordinary symbols are examined, the latter ones by means of procedure "traverse tree", which recursively walks through the tree, investigating the symbols in alphabetical order, and which calls at each node of the tree the procedure "define" as actual parameter for "investigate" (local symbols have been treated explicitly, but analogously to "define"). Only those (local) symbols, which have been used as future reference, without having occurred as (local)

label, need some special treatment. The information cells of these symbols, satisfy the following properties:

- 1) The field `t` is positive (irrelevant in the case of local labels), indicating that the symbol has not yet become a defined one;
- 2) The field value 2 contains a reference, unequal to the empty pointer, indicating that the symbol has been used as future reference;
- 3) The field value 1 still contains the initial value zero, indicating that the symbol has not occurred as label preceding the END instruction (this chain of symbols has not yet been updated).

For each of these symbols, consecutively one MIX word is reserved, by means of procedure "produce" (overwriting of addresses, which have been used already, still is impossible), and the addresses of these words are filled in at the locations at which the future references occurred, by means of procedure "update" (also procedure "produce" is called, for updating an address; now the overwriting test mechanism is eliminated, by testing whether the location counter does not point at this address).

To inform the MIXAL programmer, for which symbols a new MIX word is reserved, output is delivered, immediately following the END instruction, consisting of the symbol in question, followed by a colon and the pseudo instruction `CON 0` illustrating what happened.

Next, a number of words have to be reserved for the literal constants. Walking through the chain of occurrences of literal constants, we reserve for each one a consecutive MIX word and put the corresponding word value in it by means of procedure "produce", and we update the address part with the actual address of the corresponding word value. Having dealt with the insertions, the possible chain of labels, preceding the END instruction is updated.

Here we remark that property 3) for symbols to be inserted, is necessary, as we don't want that the labels preceding the END instruction, get their value by means of "insert constants", but by means of the call of "update", at the end of each instruction.

After all, for the convenience of the MIXAL programmer, a dump of the symbol table is given by means of procedure "dump symbol tabel". Thus, an alphabetical list of symbols used is printed out, each symbol preceded

by the value it has become equivalent to, and the linenumber of the line at which the symbol occurred as label. This information easily is fetched from the symbol table, which is walked through, in alphabetical order of symbols, by means of procedure "traverse tree" with the actual parameter "dump" for "investigate".

7. The ALGOL 60 procedure "mixal assembler".

Next follows the complete text of the MIXAL assembler in ALGOL 60 without further comment.

```
boolean procedure mixal assembler(m, start0); array m; integer start0;  
begin
```

```
boolean erroneous, from string, future, code, text;  
integer underlining, bar, space, asterisk, semicolon, colon, comma,  
point, paren, thesis, equal sign, plus, minus, times, div1, dec, quote,  
letter a, letter z, letter b, letter f, letter h, nlcr, tab,  
div2, symbol, number, sep,  
undefined, defined, local label, local backward, local forward,  
equ, orig, con, alf, end, last inst,  
last symbol, type, char, last constant, synt unit,  
read ptr, pos, last pos, line number, table ptr1,  
lit cons ptr, u1, b1, aux;  
real start,lc, dangerous, value of number, time 0;  
boolean array defined local backward[0 : 9];  
integer array line[-40 : 99], h,f[0 : 9], t[1 : 2000];  
real array b[0 : 9];
```

```
integer procedure read char;  
if from string  
then begin char:= read from string;  
read char:=char:= if char = 127 then 69  
else if char > 35 then char - 27  
else char  
end  
else begin integer s;  
if text  $\vee$  code then print line;  
if pos = -1  
then begin if line number < 6708  
then line number:= line number + 1;  
buffer(-4,-1,-4, line number); pos:=0  
end;  
s:= resym; for s:=s while s = 134 do s:= resym;  
if s = 135 then s:= 119;  
char:= if s = 127 then 69  
else if s > 35 then s - 27 else s;  
if char = underlining then u1:=1000  
else if char = bar then b1:=1000  
else begin if pos > 99  $\wedge$  char  $\neq$  nlcr  
then begin print line; pos:=0 end;  
if char = tab  $\vee$  char = nlcr  
then s:=ext(space);  
s:=2  $\times$  b1 + u1 + s; u1:=b1:=0;  
if s = 2028  $\vee$  s = 2055  
then begin errorrm( $\{$ end of input $\}$ );  
for aux:=0 step 1 until 3999 do  
m[aux]:=0; new page; goto begin  
end;  
line[pos]:=s; last pos:=pos;  
if char = tab then pos:=(pos+9):8 $\times$ 8  
else if char = nlcr  
then begin text:=true; pos:= -1 end  
else pos:=pos + 1  
end;  
end;
```

```

    if char > 99 then errorrm({unknown character});
    read char:=char
end read char;

```

```

real procedure read 5 char;
if synt unit = point
then begin real value; integer i, c; value:=0;
    for i:=1 step 1 until 5 do
        begin c:=if char = nlcr then space else read char;
            value:=value × 100 + (if c = nlcr then space else c)
        end;
        if char ≠ nlcr then char:= space;
        read synt unit; read 5 char:= value
    end
else begin errorrm({. required}); read 5 char:=0 end;

```

```

procedure print line;
begin integer i, s; boolean u, b;
    for i:= -40 step 1 until last pos do
        begin s:=line[i]; u:=b:=false;
            if s > 2000
            then begin b:=true; s:=s - 2000 end;
            if s > 1000
            then begin u:=true; s:=s - 1000 end;
            if u then prsym(ext(underlining));
            if b then prsym(ext(bar));
            prsym(s); line[i]:=ext(space)
        end;
        prsym(ext(nlcr)); code:=text:=false; last pos:=0
    end print line;

```

```

procedure errorrm(s); string s;
begin print line; printtext(s); prsym(ext(nlcr)); erroneous:=true end;

```

```

integer procedure read synt unit;
begin for char:=char while char = space ∨ char = tab do read char;
    if char < letter z
    then begin integer p, n, q1, q, i, value;
        boolean contains letter, compl;
        contains letter:= letter a < char ∧ char < letter z;
        p:=table ptr1 + 4; q1:=char; q:=read char; n:=1;
        for value:=q1 + 1, value while 7 compl do
            begin if char = space then read char;
                if q = space then q:=char;
                compl:= letter z < char;
                if compl then value:= - value × 1001(4-n);
                if compl ∨ n = 4
                then begin t[p]:=value; value:=0;
                    p:=p + 1; test on overflow(p)
                end;
                if 7 compl
                then begin n:=if n < 4 then n + 1 else 1;
                    value:=value × 100 + char + 1;

```

```

        if letter a ≤ char ∧ char ≤ letter z
        then contains letter := true;
        read char
    end
end;
if p = table ptr1 + 5 ∧ n = 2 ∧ q1 < letter a ∧
(q = letter b ∨ q = letter f ∨ q = letter h)
then begin synt unit := symbol;
    type := if q = letter b then local backward
            else if q = letter f then local forward
            else local label;
    last symbol := q1
end
else if contains letter
    then begin synt unit := symbol;
        last symbol := look for symbol(p);
        type := if t[last symbol+1] < 0
                then defined else undefined
            end
        else begin synt unit := number; value of number := 0;
            for p := table ptr1 + 4, p+1 while t[p-1] > 0 do
                begin value := abs(t[p]);
                    for i := 3, i-1 while value ≠ 0 do
                        begin aux := 100/i; q := value : aux;
                            value := value - q × aux;
                            value of number := value of number × 10 + q - 1
                        end
                    end
                end
            end
        end
    end
else if char = quote ∨ char = nlcr
    then begin synt unit := sep;
        for char := char while char ≠ nlcr do read char;
            char := space
        end
    else if char = semicolon
        then begin synt unit := sep; char := space; text := true end
        else if char ≠ div1
            then begin synt unit := char; char := space end
            else if read char = div1
                then begin synt unit := div2; char := space end
                else synt unit := div1;
            read synt unit := synt unit
        end read synt unit;

```

```

integer procedure look for symbol(ptr); value ptr; integer ptr;
begin integer p, next, i, wp, w, comp, llink, rlink;
    for p := 1, next while p ≠ 0 do
        begin for i := 4, i+1 while comp = 0 ∧ wp > 0 ∧ w > 0 do
            begin wp := t[p + i]; w := t[table ptr1 + i];
                comp := abs(w) - abs(wp)
            end;
            if comp = 0 then comp := if wp > 0 then -1 else
                if w > 0 then 1 else 0;
            if comp = 0

```

```

then begin look for symbol:=p; next:=0;
      if table ptr1 = 1 then table ptr1:=ptr
    end
else begin aux:=t[p]; llink:=aux :  $_{10}^4$ ; rlink:= aux - llink  $\times$   $_{10}^4$ ;
      next:=if comp < 0 then llink else rlink;
      if next = 0
      then begin look for symbol:=table ptr1;
            if comp < 0
            then llink:=table ptr1
            else rlink:=table ptr1;
            t[p]:=llink  $\times$   $_{10}^4$  + rlink;
            t[table ptr1 ]:=0;
            t[table ptr1 + 1]:=0;
            t[table ptr1 + 2]:=0;
            t[table ptr1 + 3]:=9999;
            table ptr1:=ptr
          end
        end
      end
end look for symbol;

```

```

procedure test on overflow(ptr); value ptr; integer ptr;
if ptr > lit cons ptr
then begin errorm({overflow symbol table});
      for char:=char while true do read char
end test on overflow;

```

```

real procedure read atomic expression;
begin real at expr; read atomic expression:=at expr:=0;
  if synt unit = symbol
  then begin if type = defined
            then begin aux:=t[last symbol + 3];
                  at expr:=sgn(aux) $\times$ (t[last symbol+2] $\times$  $_{10}^6$ +abs(aux))
                end
            else if type = local backward
            then begin if defined local backward[last symbol]
                      then at expr:=b[last symbol]
                      else errorm({undefined local backward})
                    end
            else if type = local forward
            then errorm({local forward in expression})
            else if type = local label
            then errorm({local label in expression})
            else errorm({undefined symbol in expression});
          read synt unit
        end
    else if synt unit = number
    then begin at expr:=value of number; read synt unit end
    else if synt unit = asterisk
    then begin at expr:=lc; read synt unit end
    else errorm({atomic expression required});
  if abs(at expr) <  $_{10}^{10}$ 
  then read atomic expression:=at expr
  else errorm({overflow(atomic expression)});
end read atomic expression;

```

```

real procedure read expression;
begin real expr; integer operator; read expression:=0;
  expr:=if 1 (synt unit = plus  $\vee$  synt unit = minus)
    then read atomic expression else 0;
  for synt unit:=synt unit
  while synt unit = plus  $\vee$  synt unit = minus  $\vee$  synt unit = times  $\vee$ 
    synt unit = div1  $\vee$  synt unit = div2  $\vee$  synt unit = dec
  do begin operator:=synt unit; read synt unit;
    expr:=arithm(expr, operator, read atomic expression)
  end;
  read expression:=expr
end read expression;

```

```

real procedure arithm(a, op, b); value a,op,b; real a,b; integer op;
if op = plus
then begin real value; value:=a + b; arithm:=0;
  if abs(value) <  $10^{10}$  then arithm:=value
  else error(overflow (+))
end
else if op = minus
then begin real value; value:=a - b; arithm:=0;
  if abs(value) <  $10^{10}$  then arithm:=value
  else error(overflow (-))
end
else if op = times
then begin integer s,h1,t1,h2,t2,c; real r0,r1; arithm:=0;
  s:=sgn(a  $\times$  b); a:=abs(a); b:=abs(b);
  h1:=entier(a/ $10^6$ ); t1:=a - h1  $\times$   $10^6$ ;
  h2:=entier(b/ $10^6$ ); t2:=b - h2  $\times$   $10^6$ ;
  r0:=t1  $\times$  t2; c:=entier(r0/ $10^6$ ); r0:=r0 - c  $\times$   $10^6$ ;
  r1:=t1h2 + h1t2 + c; c:=entier(r1/ $10^4$ ); r1:=r1 - c $\times 10^4$ ;
  if h1  $\times$  h2 + c = 0 then arithm:=s  $\times$  (r1  $\times$   $10^6$  + r0)
  else error(overflow (x))
end
else if op = div1
then begin arithm:=0; if b  $\neq$  0
  then arithm:=sgn(a  $\times$  b)  $\times$  entier(abs(a/b))
  else error(overflow (/))
end
else if op = div2
then begin arithm:=0; if b  $\neq$  0  $\wedge$  abs(a) < abs(b)
  then arithm:=sgn(a $\times$ b)entier(abs(a/b) $\times 10$ )
  else error(overflow (//))
end
else begin real value; value:=a  $\times$  10 + b; arithm:=0;
  if abs(value) <  $10^{10}$  then arithm:=value
  else error(overflow (:))
end arithm;

```

```

real procedure read w value;
begin real con, a, field, aux1, aux2, aux3;
  integer l,r,s; boolean comma read;
  con:=0;
  for synt unit:=synt unit, synt unit while comma read do

```

```

begin a:=read expression;
  if synt unit = paren
  then begin read synt unit; field:= read expression;
    if field = 5 then con:= a
    else if field = 0 then con:= sgn(a) × abs(con)
    else if field = 15 then con:= sgn(con) × abs(a)
    else begin l:=entier(field/10); r:=field - l × 10;
      if 0 < l ∧ l < r ∧ r < 5
      then begin if l ≠ 0 then s:=sgn(con)
        else begin s:=sgn(a); l:=1 end;
        a:=abs(a); con:=abs(con);
        aux1:=100l(6-l); aux2:=100l(5-r);
        aux3:=aux1/aux2;
        con:=s × (entier(con/aux1) × aux1 +
          (a - entier(a/aux3)×aux3) × aux2 +
          con - entier(con/aux2) × aux2)
        end
      else errorm({illegal field})
      end;
    if synt unit = thesis then read synt unit
    else errorm({ required })
    end
  else con:=a;
  if synt unit = comma
  then begin comma read:=true; read synt unit end
  else comma read:=false
  end;
read w value:=con
end read w value;

```

```

procedure read mixal program;
begin integer inst, last label; real equiv; last label:= 9999;
  for inst:= 0, inst while inst ≠ end do
  for synt unit:=colon, synt unit while synt unit = colon do
  if read synt unit ≠ symbol
  then begin if synt unit ≠ sep then errorm({symbol required});
    synt unit:= colon
    end
  else begin inst:=last symbol;
    if read synt unit = colon
    then begin if type = undefined
      then begin if t[last symbol+2] = 0
        then begin t[last symbol+2]:=last label;
          last label:=last symbol
          end
        else errorm({label used twice})
        end
      else if type = local label
      then begin if h[last symbol] = 0
        then begin h[last symbol]:=last label;
          last label:=last symbol-1
          end
        else errorm({local label used twice})
        end
      else errorm({illegal label})
      end
    end
  end
end

```

```

else begin equiv:=lc;
    if inst < last inst
    then produce(lc,read mix machine inst(inst))
    else if inst = equ
        then equiv:=read w value
        else if inst = orig
            then lc:=read w value
            else if inst = con
                then produce(lc,read w value)
                else if inst = alf
                    then produce(lc,read 5 char)
                    else if inst = end
                        then begin start:=read w value;
                                insert constants;
                                equiv:=lc
                                end
                        else
                            error('unknown instruction');
    if synt unit ≠ sep
    then error('separator required');
    for synt unit:=synt unit while synt unit ≠ sep do
        read synt unit;
    update(last label, equiv); last label:=9999
end
end
end read mixal program;

```

```

boolean procedure possible(address); value address; real address;
possible:=if 0 < address ∧ address < 3999
    then m[address] = 0 ∧ address ≠ dangerous
    else false;

```

```

real procedure read mix machine inst(instruction); value instruction;
integer instruction;
begin real a part,i part,f part,c part;
    a part:=i part:=0; aux:=abs(t[instruction + 1]); f part:=aux : 102;
    c part:=aux - f part × 102; f part:=f part - f part : 102 × 102;
    if synt unit = symbol
    then begin if type = undefined
        then begin a part:=t[last symbol + 3]; future:=true;
            if possible(lc) then t[last symbol + 3]:=lc;
            read synt unit
            end
        else if type = local forward
            then begin a part:=f[last symbol]; future:=true;
                if possible(lc) then f[last symbol]:=lc;
                read synt unit
                end
            else a part:=read expression
            end
    end
else if synt unit = equal sign
    then begin real value;
        a part:=last constant; future:=true;
        if possible(lc) then last constant:=lc;
        read synt unit; value:=read w value;
    end

```

```

        lit cons ptr:=lit cons ptr - 2;
        test on overflow(table ptr1);
        t[lit cons ptr + 1]:=aux:=entier(abs(value)/106);
        t[lit cons ptr + 2]:=sgn(value)×(abs(value) - aux×106);
        if synt unit = equal sign then read synt unit
        else error(⌋ required⌋)
    end
else if 1 (synt unit=comma ∨ synt unit=paren ∨ synt unit=sep)
    then a part:=read expression;
if synt unit = comma
then begin read synt unit; i part:=read expression end;
if synt unit = paren
then begin read synt unit; f part:=read expression;
    if synt unit = thesis then read synt unit
    else error(⌋ required⌋)
end;
if abs(a part) > 9999
then begin error(⌋abs(a part) > 9999⌋); a part:=0 end;
if i part < 0 ∨ i part > 99
then begin error(⌋i part < 0 ∨ i part > 99⌋); i part:=0 end;
if f part < 0 ∨ f part > 99
then begin error(⌋f part < 0 ∨ f part > 99⌋); f part:=0 end;
read mix machine inst:=sgn(a part) ×
    (abs(a part) × 106 + i part × 104 + f part × 102 + c part)
end read mix machine inst;

```

```

procedure produce(address,value);value address,value;real address,value;
begin if code then print line;
    if 1 (0 < address ∧ address < 3999)
    then begin error(⌋wrong address⌋);buffer(-28,1,-10,address) end
    else begin if address = lc ∧ (address = dangerous ∨ m[address] ≠ 0)
        then error(⌋address already used⌋)
        else begin if future ∧ value = 0 then dangerous:=address;
            m[address]:=value;
            buffer(-14, if future then -3 else 5, 2, value)
        end;
        buffer(-35, -2, 2, address);
        if address = lc then lc:=lc+1 else line[-40]:=ext(asterisk)
    end;
    future:=false; code:= true
end produce;

```

```

procedure update(label,equiv);value label,equiv;integer label;real equiv;
for label:=label while label ≠ 9999 do
begin integer symb, link, address;
    if label > 0
    then begin symb:=label; label:=t[symb + 2];
        link:=t[symb + 3]; aux:=t[symb + 1];
        t[symb + 1]:= -(aux + (line number+1 - aux×104)×104);
        t[symb + 2]:=aux:=entier(abs(equiv)/106);
        t[symb + 3]:=sgn(equiv) × (abs(equiv) - aux × 106)
    end
    else begin symb:=-label-1; label:=h[symb]; h[symb]:=0;
        link:=f[symb]; f[symb]:=9999; b[symb]:=equiv;
    end
end

```

```

        defined local backward[symb]:=true
    end;
    for link:=link while link  $\neq$  9999 do
    if abs(equiv) > 9999
    then begin link:=9999; errorrm(abs(future ref) > 9999) end
    else begin address:=link; link:=entier(m[address]/106);
        produce(address,sgn(equiv) $\times$ (m[address]+(abs(equiv)-link) $\times$ 106))
    end
end update;

```

```

procedure buffer(p,n,m,value); value p,n,m,value; integer p,n,m; real value;
begin integer i, j, q; real aux1;
    if n > 0
    then line[p - (abs(m)+1) $\times$ n - 1]:=ext(if value<0 then minus else plus);
    aux1:=104abs(n $\times$ m); value:=abs(value) - entier(abs(value)/aux1)  $\times$  aux1;
    for i:=abs(n)-1 step -1 until 0 do
    for j:=abs(m)-1 step -1 until 0 do
    begin aux1:=104(m  $\times$  i + j); q:=entier(value/aux1);
        value:=value - q  $\times$  aux1;
        if q > 0  $\vee$  (i = 0  $\wedge$  j = 0) then m:=abs(m);
        if m > 0 then line[p - ((m+1)  $\times$  i + j)]:=ext(q)
    end
end buffer;

```

```

procedure traverse tree(investigate, symbol); value symbol;
integer symbol; procedure investigate;
if symbol  $\neq$  0
then begin integer llink, rlink; aux:=t[symbol];
    llink:=aux : 104; rlink:=aux - llink  $\times$  104;
    traverse tree(investigate, llink);
    investigate(symbol);
    traverse tree(investigate, rlink)
end traverse tree;

```

```

procedure buffer symb(s, lb, ub); value s, lb; integer s, lb, ub;
if s < 0
then begin line[lb]:= -s-1; line[lb+1]:=ext(letter h); ub:=lb+2 end
else begin integer p, q, value, i; ub:=lb;
    for p:=s + 4, p + 1 while t[p - 1] > 0 do
    begin value:=abs(t[p]);
        for i:=3, i - 1 while value  $\neq$  0 do
        begin aux:=1004i; q:=value : aux;
            value:=value - q  $\times$  aux;
            if ub = 99
            then begin line[99]:=2000 + ext(space);
                last pos:=99; print line; ub:=0
            end;
            line[ub]:=q - 1; ub:=ub + 1
        end
    end
end buffer symb;

```

```

procedure buffer constant(s); value s; integer s;
begin integer ptr;
    if code  $\vee$  text then print line;
    buffer symb(s,0,ptr); line[ptr]:=ext(colon);
    if ptr > 15 then begin last pos:=ptr; print line end;
    buffer symb(con,16,ptr); last pos:=ptr + 5; line[last pos]:=ext(0)
end buffer constant;

```

```

procedure insert constants;
begin procedure define(symbol); value symbol; integer symbol;
    if t[symbol+1] > 0  $\wedge$  t[symbol+2] = 0  $\wedge$  t[symbol+3]  $\neq$  9999
    then begin integer equiv; equiv:=lc;
        buffer constant(symbol); produce(lc, 0);
        t[symbol+2]:=9999; update(symbol, equiv)
    end define;
    integer symb, address, ln;
    code:=true; ln:=line number; line number:=6709;
    for symb:= -1 step -1 until -10 do
    if f[-symb-1]  $\neq$  9999
    then begin integer equiv; equiv:=lc;
        buffer constant(symb); produce(lc, 0);
        h[-symb-1]:=9999; update(symb, equiv)
    end;
    traverse tree(define, 1);
    line number:=ln;
    for address:=last constant while address  $\neq$  9999 do
    begin real value; last constant:=entier(m[address]/ $10^6$ );
        value:=m[address] + (lc - last constant)  $\times 10^6$ ;
        aux:=t[lit cons ptr + 2];
        produce(lc, sgn(aux) $\times$ (t[lit cons ptr + 1]  $\times 10^6$  + abs(aux)));
        lit cons ptr:=lit cons ptr + 2;
        produce(address, value)
    end
end insert constants;

```

```

procedure dump(symbol); value symbol; integer symbol;
if t[symbol+1] < 0
then begin integer ptr; real value; aux:=t[symbol+3];
    value:=sgn(aux) $\times$ (t[symbol+2] $\times 10^6$ +abs(aux));
    line number:=abs(t[symbol+1])  $\div 10^4$  - 1;
    buffer(-14, 5, 2, value);
    if line number > 6708 then line[-3]:= ext(plus)
    else buffer(-4, -1, -4, line number);
    buffer symb(symbol, 0, ptr); last pos:= ptr - 1;
    print line
end dump;

```

```

procedure dump symbol table;
begin new page;
    printtext(value and line number of defined symbols:);
    prsym(ext(nlcr)); prsym(ext(nlcr));
    traverse tree(dump, 1); new page
end dump symbol table;

```

```

procedure initialize;
begin integer i;
    procedure rc(s); integer s; s:=read char;
    erroneous:=false; from string:=true; read ptr:=0;
    rc(underlining); rc(bar); rc(space); rc(asterisk); rc(semicolon);
    rc(colon); rc(comma); rc(point); rc(paren); rc(thesis); rc(equal sign);
    rc(plus); rc(minus); rc(times); rc(div1); rc(dec); rc(letter a);
    rc(letter z); rc(letter b); rc(letter f); rc(letter h); read char;
    quote:=94; nlcr:=92; tab:=91;
    sep:=101; number:=102; symbol:=103; div2:=104;
    undefined:=0; defined:=1; local label:=2; local backward:=3;
    local forward:=4;
    table ptr1:=1; lit cons ptr:=2000; t[1]:=t[3]:=0; t[4]:=9999;
    for i:= -40 step 1 until 99 do line[i]:=ext(space);
    for i:=0 step 1 until 9 do
        begin h[i]:=0; f[i]:=9999; defined local backward[i]:=false end;
    last constant:=9999; lc:=0; dangerous:= -1;
    future:=code:=text:=false;
    pos:=last pos:=line number:=u1:=b1:=0;
    for synt unit:=sep, synt unit while synt unit = sep do
        if read synt unit = symbol
        then begin read synt unit;
            t[last symbol + 1]:=104 + read atomic expression
        end
    else if synt unit  $\neq$  sep  $\wedge$  synt unit  $\neq$  paren then errorrm( $\langle$ er 01 $\rangle$ );
    if synt unit = paren
    then begin last inst:=last symbol;
        read synt unit; alf:=last symbol;
        read synt unit; con:=last symbol;
        read synt unit; end:=last symbol;
        read synt unit; equ:=last symbol;
        read synt unit; orig:=last symbol;
        if read synt unit  $\neq$  thesis then errorrm( $\langle$ er 03 $\rangle$ )
    end
    else errorrm( $\langle$ er 02 $\rangle$ );
    if erroneous then
        begin errorrm( $\langle$ error detected during initialisation of assembler $\rangle$ );
            exit
        end;
    from string:=false
end initialize;

```

```

integer procedure read from string;
begin read from string:=stringsymbol(read ptr,  $\langle$  x;:,.( )=+-x/:azbfh

```

j3np	553;	ent5	265;	ld4	514;	dec6	166;	inc6	066
j1	447;	slax	206;	cmp6	576;	enn5	365;	fsub	602
j1p	251;	j6n	056;	jxp	257;	ldx	517;	st4	534
cmp2	572;	dec2	162;	enn1	361;	ent1	261;	fadd	601
inc2	062;	j1n	051;	j2np	552;	j4p	254;	janp	550

jxn	057;	ld2	512;	ld6	516;	nop	000;	src	506
stj	240;	char	105;	cmp4	574;	cmpx	577;	dec4	164
decx	167;	enn3	363;	enna	360;	ent3	263;	enta	260
fdiv	604;	in	044;	inc4	064;	incx	067;	j1np	551
j2n	052;	j2p	252;	j4n	054;	j5np	555;	j6p	256

jbus	042;	jnov	347;	jxnp	557;	ld1	511;	ld3	513
ld5	515;	lda	510;	move	107;	out	045;	sra	106
st2	532;	st6	536;	stz	541;	add	501;	cmp1	571
cmp3	573;	cmp5	575;	cmpa	570;	dec1	161;	dec3	163
dec5	165;	deca	160;	div	504;	enn2	362;	enn4	364

enn6	366;	ennx	367;	ent2	262;	ent4	264;	ent6	266
entx	267;	fcmp	670;	fmul	603;	hlt	205;	inc1	061
inc3	063;	inc5	065;	inca	060;	ioc	043;	j1nn	351
j1nz	451;	j1z	151;	j2nn	352;	j2nz	452;	j3n	053
j3p	253;	j4np	554;	j5n	055;	j5p	255;	j6np	556

jan	050;	jap	250;	jg	647;	jmp	047;	jred	046
jxn	357;	jxnz	457;	jxz	157;	ld1n	521;	ld2n	522
ld3n	523;	ld4n	524;	ld5n	525;	ld6n	526;	ldan	520
ldxn	527;	mul	503;	num	005;	sla	006;	slc	406
srax	306;	st1	531;	st3	533;	st5	535;	sta	530

stx	537;	sub	502;	j2z	152;	j3nn	353;	j3nz	453
j3z	153;	j4nn	354;	j4nz	454;	j4z	154;	j5nn	355
j5nz	455;	j5z	155;	j6nn	356;	j6nz	456;	j6z	156
jann	350;	janz	450;	jaz	150;	je	547;	jge	747
jle	947;	jne	847;	jov	247;	jsj	147;		

```

      (      alf   con   end   equ   orig )      †);
      read ptr:=read ptr + 1
end read from string;

```

```

integer procedure sgn(a); value a; real a;
sgn:=if a < 0 then -1 else +1;

```

```

integer procedure ext(c); value c; integer c;
ext:=if c = 69 then 127
      else if c > 35 then c + 27
      else c;

```

```

begin: initialize; time 0:= time;
      read mixal program; print line;
prsym(ext(nlcr)); printtext(†start address: †); fixt(10,0,start);
prsym(ext(nlcr));
if 1 (0 < start ^ start < 3999)
then errorrm(†wrong start address†)

```

```
else start0:=start;  
if erroneous then error(mixal program incorrect);  
prsym(ext(nlcr)); printtext(translation time:);  
absfixt(4,2,time - time 0); printtext(sec.);  
dump symbol table;  
mixal assembler:= 7 erroneous  
  
end mixal assembler;
```

```
1  "CHAPTER 8      MIXAL ASSEMBLER DESCRIBED IN MIXAL.
2
3
4
5  " UNIT NUMBERS FOR APPARATUS
6  READER:      EQU 16
7  CPUNCH:      EQU 17
8  PRINTER:     EQU 18
9
10
11  " INTERNAL CODE OF CHARACTERS
12  LETTER B:    EQU 11
13  LETTER F:    EQU 15
14  LETTER H:    EQU 17
15  LETTER Z:    EQU 35
16  PLUS:        EQU 37
17  MINUS:       EQU 38
18  TIMES:
19  ASTERISK:    EQU 39
20  DIV:         EQU 40
21  EQUAL SIGN:  EQU 43
22  COMPA:       EQU 60
23  POINT:       EQU 61
24  DEC:
25  COLON:       EQU 63
26  SEMICOLON:   EQU 64
27  SPACE:       EQU 66
28  PAREN:       EQU 71
29  THESIS:      EQU 72
30  NLCR:        EQU 92
31  QUOTE:       EQU 94
32
33  " CONSTANTS USED BY ASSEMBLER
34  SEP:         EQU 101
35  NUMBER:      EQU 102
36  SYMBOL:      EQU 103
37  DIV2:        EQU 104
38  UNDEFINED SYMB: EQU 0
39  DEFINED SYMB: EQU 1
40  LOCAL LABEL: EQU 2
41  LOCAL BACKWARD: EQU 3
42  LOCAL FORWARD: EQU 4
43  VALUE1:      EQU 0 : 2
44  VALUE2:      EQU 4 : 5
45  MOD:         EQU 0
46  N:           EQU - 1
47  LENGTH OF TABLE: EQU 1500
48
49
50  " WORKING SPACE OF ASSEMBLER
51  OR'G 1200
52
53  BEGIN OF ASS:
54
55  CHAR:        CON 0
56  5 CHAR:      CON 0
57  LINE NUMBER: CON 0
58
59  12 00 + 00 00 00 00 00
60  12 01 + 00 00 00 00 00
61  12 02 + 00 00 00 00 00
```

2001/3-126	D 2166Z.1	GERTENVELDEN	23	23	23
12 03	+	00 00 00 00 00	59	CODE:	CON
12 04	+	00 00 00 00 00	60	TEXT:	CON
12 05	+	00 00 00 00 00	61	SYNT UNIT:	CON
12 06	+	00 00 00 00 00	62	LAST SYMBOL:	CON
12 07	+	00 00 00 00 00	63	TYPE:	CON
12 08	+	00 00 00 00 00	64	VALUE OF NUMBER:CON	CON
12 09	+	00 00 00 00 00	65	EPHNECUS:	CON
12 10	+	00 00 00 00 00	66	AT EXPR:	CON
12 11	+	00 00 00 00 00	67	EXPR:	CON
12 12	+	00 00 00 00 00	68	VALUE:	CON
12 13	+	00 00 00 00 00	69	CONSTANT:	CON
12 14	+	00 00 00 00 00	70	ADDRESS:	CON
12 15	+	00 00 00 00 00	71	A PART:	CON
12 16	+	00 00 00 00 00	72	I PART:	CON
12 17	+	00 00 00 00 00	73	F PART:	CON
12 18	+	00 00 00 00 00	74	INSTRUCTION:	CON
12 19	+	00 00 00 00 00	75	LC:	CON
12 20	+	00 00 00 00 00	76	FUTURE:	CON
12 21	+	00 00 00 00 00	77	DANGEROUS:	CON
12 22	+	00 00 00 00 00	78	LAST LABEL:	CON
12 23	+	00 00 00 00 00	79	EQUIV:	CON
12 24	+	00 00 00 00 00	80	INST:	CON
12 25	+	00 00 00 00 00	81	LAST CONSTANT:	CON
12 26	+	00 00 00 00 00	82	LIT CONS PTR:	CON
12 27	+	00 00 00 00 00	83	START ADDRESS:	CON
12 28	+	00 00 00 00 00	84		
12 29	+	00 00 00 00 00	85		
12 30	+	00 00 00 00 00	86		
12 31	+	00 00 00 00 00	87	BACKWARD:	ORIG
12 32	+	00 00 00 00 00	88	HERE:	ORIG
12 33	+	00 00 00 00 00	89	INBUF X:	ORIG
12 34	+	00 00 00 00 00	90	INBUF:	ORIG
12 35	+	00 00 00 00 00	91	OUTBUF X:	ORIG
12 36	+	00 00 00 00 00	92	OUTBUF:	ORIG
12 37	+	00 00 00 00 00	93	TABLE:	ORIG
12 38	+	00 00 00 00 00	94		
12 39	+	00 00 00 00 00	95		
12 40	+	00 00 00 00 00	96		
12 41	+	00 00 00 00 00	97	" BEGIN OF PERMANENT PART OF ASSEMBLER	
12 42	+	00 00 00 00 00	98		
12 43	+	00 00 00 00 00	99		
12 44	+	00 00 00 00 00	100	TABLE PTR:	CON
12 45	+	00 00 00 00 00	101	LAST INST:	CON
12 46	+	00 00 00 00 00	102	ALF:	CON
12 47	+	00 00 00 00 00	103	CON:	CON
12 48	+	00 00 00 00 00	104	END:	CON
12 49	+	00 00 00 00 00	105	END:	CON
12 50	+	00 00 00 00 00	106	CRIG:	CON
12 51	+	00 00 00 00 00	107	BASE:	CON
12 52	+	00 00 00 00 00	108	BYTESIZE: S:	EQU
12 53	+	00 00 00 00 00	109	MAX BYTE:	CON
12 54	+	00 00 00 00 00	110	END MARKER:	
12 55	+	00 00 00 00 00	111	MAX ADDRESS:	CON
12 56	+	00 00 00 00 00	112	MAX WORD:	CON
12 57	+	00 00 00 00 00	113	LENGTH OF MEMORY:	
12 58	+	00 00 00 00 00	114	HUNDREDTHOUSAND:CON	CON
12 59	+	00 00 00 00 00	115	SPACES:	ALF
12 60	+	00 00 00 00 00	116	JOB SEPARATOR:	ALF
12 61	+	00 00 00 00 00	117	ERROR:	ALF
12 62	+	00 00 00 00 00	118		
12 63	+	00 00 00 00 00	119		
12 64	+	00 00 00 00 00	120		
12 65	+	00 00 00 00 00	121		
12 66	+	00 00 00 00 00	122		
12 67	+	00 00 00 00 00	123		
12 68	+	00 00 00 00 00	124		
12 69	+	00 00 00 00 00	125		
12 70	+	00 00 00 00 00	126		
12 71	+	00 00 00 00 00	127		
12 72	+	00 00 00 00 00	128		
12 73	+	00 00 00 00 00	129		
12 74	+	00 00 00 00 00	130		
12 75	+	00 00 00 00 00	131		
12 76	+	00 00 00 00 00	132		
12 77	+	00 00 00 00 00	133		
12 78	+	00 00 00 00 00	134		
12 79	+	00 00 00 00 00	135		
12 80	+	00 00 00 00 00	136		
12 81	+	00 00 00 00 00	137		
12 82	+	00 00 00 00 00	138		
12 83	+	00 00 00 00 00	139		
12 84	+	00 00 00 00 00	140		
12 85	+	00 00 00 00 00	141		
12 86	+	00 00 00 00 00	142		
12 87	+	00 00 00 00 00	143		
12 88	+	00 00 00 00 00	144		
12 89	+	00 00 00 00 00	145		
12 90	+	00 00 00 00 00	146		
12 91	+	00 00 00 00 00	147		
12 92	+	00 00 00 00 00	148		
12 93	+	00 00 00 00 00	149		
12 94	+	00 00 00 00 00	150		
12 95	+	00 00 00 00 00	151		
12 96	+	00 00 00 00 00	152		
12 97	+	00 00 00 00 00	153		
12 98	+	00 00 00 00 00	154		
12 99	+	00 00 00 00 00	155		
12 100	+	00 00 00 00 00	156		
12 101	+	00 00 00 00 00	157		
12 102	+	00 00 00 00 00	158		
12 103	+	00 00 00 00 00	159		
12 104	+	00 00 00 00 00	160		
12 105	+	00 00 00 00 00	161		
12 106	+	00 00 00 00 00	162		
12 107	+	00 00 00 00 00	163		
12 108	+	00 00 00 00 00	164		
12 109	+	00 00 00 00 00	165		
12 110	+	00 00 00 00 00	166		
12 111	+	00 00 00 00 00	167		
12 112	+	00 00 00 00 00	168		
12 113	+	00 00 00 00 00	169		
12 114	+	00 00 00 00 00	170		
12 115	+	00 00 00 00 00	171		
12 116	+	00 00 00 00 00	172		
12 117	+	00 00 00 00 00	173		
12 118	+	00 00 00 00 00	174		
12 119	+	00 00 00 00 00	175		
12 120	+	00 00 00 00 00	176		
12 121	+	00 00 00 00 00	177		
12 122	+	00 00 00 00 00	178		
12 123	+	00 00 00 00 00	179		
12 124	+	00 00 00 00 00	180		
12 125	+	00 00 00 00 00	181		
12 126	+	00 00 00 00 00	182		
12 127	+	00 00 00 00 00	183		
12 128	+	00 00 00 00 00	184		
12 129	+	00 00 00 00 00	185		
12 130	+	00 00 00 00 00	186		
12 131	+	00 00 00 00 00	187		
12 132	+	00 00 00 00 00	188		
12 133	+	00 00 00 00 00	189		
12 134	+	00 00 00 00 00	190		
12 135	+	00 00 00 00 00	191		
12 136	+	00 00 00 00 00	192		
12 137	+	00 00 00 00 00	193		
12 138	+	00 00 00 00 00	194		
12 139	+	00 00 00 00 00	195		
12 140	+	00 00 00 00 00	196		
12 141	+	00 00 00 00 00	197		
12 142	+	00 00 00 00 00	198		
12 143	+	00 00 00 00 00	199		
12 144	+	00 00 00 00 00	200		
12 145	+	00 00 00 00 00	201		
12 146	+	00 00 00 00 00	202		
12 147	+	00 00 00 00 00	203		
12 148	+	00 00 00 00 00	204		
12 149	+	00 00 00 00 00	205		
12 150	+	00 00 00 00 00	206		
12 151	+	00 00 00 00 00	207		
12 152	+	00 00 00 00 00	208		
12 153	+	00 00 00 00 00	209		
12 154	+	00 00 00 00 00	210		
12 155	+	00 00 00 00 00	211		
12 156	+	00 00 00 00 00	212		
12 157	+	00 00 00 00 00	213		
12 158	+	00 00 00 00 00	214		
12 159	+	00 00 00 00 00	215		
12 160	+	00 00 00 00 00	216		
12 161	+	00 00 00 00 00	217		
12 162	+	00 00 00 00 00	218		
12 163	+	00 00 00 00 00	219		
12 164	+	00 00 00 00 00	220		
12 165	+	00 00 00 00 00	221		
12 166	+	00 00 00 00 00	222		
12 167	+	00 00 00 00 00	223		
12 168	+	00 00 00 00 00	224		
12 169	+	00 00 00 00 00	225		
12 170	+	00 00 00 00 00	226		
12 171	+	00 00 00 00 00	227		
12 172	+	00 00 00 00 00	228		
12 173	+	00 00 00 00 00	229		
12 174	+	00 00 00 00 00	230		
12 175	+	00 00 00 00 00	231		
12 176	+	00 00 00 00 00	232		
12 177	+	00 00 00 00 00	233		
12 178	+	00 00 00 00 00	234		
12 179	+	00 00 00 00 00	235		
12 180	+	00 00 00 00 00	236		
12 181	+	00 00 00 00 00	237		
12 182	+	00 00 00 00 00	238		
12 183	+	00 00 00 00 00	239		
12 184	+	00 00 00 00 00	240		
12 185	+	00 00 00 00 00	241		
12 186	+	00 00 00 00 00	242		
12 187	+	00 00 00 00 00	243		
12 188	+	00 00 00 00 00	244		
12 189	+	00 00 00 00 00	245		
12 190	+	00 00 00 00 00	246		
12 191	+	00 00 00 00 00	247		
12 192	+	00 00 00 00 00	248		
12 193	+	00 00 00 00 00	249		
12 194	+	00 00 00 00 00	250		
12 195	+	00 00 00 00 00	251		
12 196	+	00 00 00 00 00	252		
12 197	+	00 00 00 00 00	253		
12 198	+	00 00 00 00 00	254		
12 199	+	00 00 00 00 00	255		
12 200	+	00 00 00 00 00	256		
12 201	+	00 00 00 00 00	257		
12 202	+	00 00 00 00 00	258		
12 203	+	00 00 00 00 00	259		
12 204	+	00 00 00 00 00	260		
12 205	+	00 00 00 00 00	261		
12 206	+	00 00 00 00 00	262		
12 207	+	00 00 00 00 00	263		
12 208	+	00 00 00 00 00	264		
12 209	+	00 00 00 00 00	265		
12 210	+	00 00 00 00 00	266		
12 211	+	00 00 00 00 00	267		
12 212	+	00 00 00 00 00	268		
12 213	+	00 00 00 00 00	269		
12 214	+	00 00 00 00 00	270		
12 215	+	00 00 00 00 00	271		
12 216	+	00 00 00 00 00	272		
12 217	+	00 00 00 00 00	273		
12 218	+	00 00 00 00 00	274		
12 219	+	00 00 00 00 00	275		
12 220	+	00 00 00 00 00	276		
12 221	+	00 00 00 00 00	277		
12 222	+	00 00 00 00 00	278		

2YU173-126 D 21662.1 GERTENVELDEN

24

28 44	+	01 01 01 01 01	119	CORRECTION:	CON	1(1:1), 1(2:2), 1(3:3), 1(4:4), 1(5:5)
28 45			120			
28 46	+	12 04 00 05 10	121			
28 47	+	12 03 00 05 01	122			
28 48			123	" SUBROUTINES OF ASSEMBLER		
28 49			124			
28 50	+	12 79 00 05 10	125	READ CHAR:	" RA IS AN OUTPUT PARAMETER FOR THE CODE OF THE CHARACTER,	
28 51	+	28 42 00 45 70	126		" R11 AND R12 ARE PERMANENT READING POINTERS,	
28 52	+	00 98 00 02 66	127		STJ 9F	" SAVE RETURN ADDRESS
28 53			128		LDA TEXT	" IF TEXT
28 54	+	00 00 47	129		ADD CODE	" OR CODE
28 55			130		JAP PRINT LINE	" THEN PRINT THIS LINE
28 56	+	12 79 00 05 10	131		J2NP READC	" JUMP IF BUFFER NOT EXHAUSTED
28 57	+	28 42 00 45 70	132		LDA INBUF + 15	" POSSIBLE JOB SEPARATOR
28 58			133		CMPLA JOB SEPARATOR (4 : 5)	" IS IT A JOB SEPARATOR ?
28 59	+	00 98 00 02 66	134		JNE IF	" ELSE CONTINUE THE READING
28 60			135		ENT6 98	" REPORT ERROR:
28 61			136		JMP ERRORM	" END OF INPUT
28 62			137		JMP RESET TABLE	" RESET THE INITIAL SYMBOL TABLE
28 63			138		JMP NEXT PROGRAM	" TREAT THE NEXT PROGRAM
28 64			139	1H:	LDA LINE NUMBER	" INCREASE
28 65	+	12 02 00 05 10	140		INCA 1	" LINE NUMBER
28 66	+	28 57 00 38 47	141		STA LINE NUMBER	" BY ONE
28 67	+	00 01 00 00 60	142		JMP BUFFER LN	" 'PRINT' THE LINE NUMBER
28 68	+	00 16 00 02 62	143		ENT2 16	" ONE AFTER LAST WORD OF BUFFER
28 69	+	28 62 00 16 42	144		JBUS * (READER)	" WAIT FOR CARD READER
28 70	+	12 64 00 02 61	145		ENT1 INBUF	" MOVE NEXT CARD FROM APPARATUS
28 71	+	12 48 00 16 07	146		MOVE INBUF X (16)	" BUFFER TO INPUT BUFFER
28 72	+	12 79 00 05 10	147		LDA INBUF + 15	" POSSIBLE JOB SEPARATOR
28 73	+	28 42 00 45 70	148		CMPLA JOB SEPARATOR (4 : 5)	" IS IT A JOB SEPARATOR ?
28 74	+	28 69 00 05 47	149		JE * + 2	" THEN SKIP NEXT INSTRUCTION
28 75	+	12 48 00 16 44	150		IN INBUF X (READER)	" READ NEXT CARD
28 76	+	00 01 00 01 62	151	2H:	DEC2 1	" PREVIOUS WORD OF BUFFER
28 77	+	00 00 52	152		J2N 3F	" JUMP IF NOT AVAILABLE
28 78	+	12 64 02 05 20	153		LDAN INBUF, 2	" CONTENTS OF THE WORD (INVERTED)
28 79	+	26 11 00 15 70	154		CMPLA SPACES (1 : 5)	" IF SPACES
28 80	+	28 69 00 05 47	155		JE 2B	" THEN LOOP
28 81	+	12 65 02 00 30	156	3H:	STA INBUF + 1, 2 (0 : 3)	" END OF BUFFER INDICATOR
28 82	+	28 74 00 00 52	157		ENT2 -16	" INITIALIZE READING PTR (TO WORDS)
28 83	+	00 16 00 02 62	158	READC:	ENN3 5 : 5 /N ,3	" PREVIOUS CHARACTER IN BYTE 5 ?
28 84	+	00 55 03 03 63	159		J3P 3F	" ELSE JUMP
28 85	+	28 76 00 05 52	160		ENT3 5 : 5	" INITIALIZE READING PTR (TO BYTES)
28 86	+	00 55 00 02 63	161		INC2 1	" NEXT WORD
28 87	+	00 01 00 00 62	162		J2NP 2F	" JUMP IF WORD IS AVAILABLE
28 88	+	00 01 00 02 62	163	1H:	ENT2 1	" POSITION = 1
28 89	+	00 92 00 02 60	164		ENT4 NLCR	" END OF LINE READ
28 90	+	12 04 00 05 30	165		STA TEXT	" TEXT OF ONE LINE COMPLETED
28 91	+	12 79 02 05 10	166	2H:	JMP 8F	" READY
28 92	+	26 85 00 05 52	167		LDA INBUF + 15, 2	" END OF BUFFER INDICATOR ?
28 93	+	28 81 00 00 50	168		JAN 1B	" THEN SKIP REMAINING SPACES
28 94	+	00 66 03 03 63	169	3H:	ENN3 6 : 6 /N ,3	" POINTER TO NEXT BYTE
28 95	+	28 87 00 02 53	170		ST3 * + 1 (4 : 4)	" MODIFY NEXT INSTRUCTION
28 96	+	12 79 02 00 10	171		LDA INBUF + 15, 2 (MOD)	" 'READ' THE CHARACTER
28 97	+	28 91 00 44 33	172		ST3 * + 1 (4 : 4)	" MODIFY NEXT INSTRUCTION
28 98	+	13 27 02 00 30	173		STA OUTBUF + 8 + 15, 2 (MOD)	" 'PRINT' THE CHARACTER

[illegible]

```
29 35 + 13 08 00 05 37 226 STX OUTBUF + 4 (3 : 5)
29 36 + 12 11 00 05 10 227 LDA EXPR " VALUE OF EXPRESSION
29 37 + 00 00 00 01 05 228 CHAR
29 38 + 13 10 00 05 30 229 STA " 'PRINT'
29 39 + 13 11 00 05 37 230 STX OUTBUF + 6 " 'THE VALUE
29 40 + 29 16 00 00 47 231 JMP PRINT LINE " PRINT THIS LINE
29 41 + 00 01 00 02 66 232 ENT6 1 " NOTE
29 42 + 12 09 00 05 36 233 ST6 " THE ERROR
29 43 + 00 00 00 00 47 234 JMP MOD " RETURN
*29 26 + 29 43 00 02 40 235
236
237 READ SYNT UNIT: " RA IS AN OUTPUT PARAMETER FOR THE CODE OF THE SYNTACTICAL UNIT
238 " R11 DELIVERS A POINTER TO A SYMBOL READ
239 STJ 9F " SAVE RETURN ADDRESS
1HI LDA CHAR " LAST CHARACTER
240 DECA SPACE " IS IT A SPACE ?
241 LETGIT " ELSE JUMP
242 JANZ READ CHAR " SKIP
243 JMP 1B " SPACES
244 DECA LETTER Z - SPACE
245 LETGIT:
246 JAP SEPARATOR " JUMP IF NOT A LETTER OR DIGIT
247 DECA 9 - LETTER Z " IS IT A LETTER ?
248 STA COMPL (0 : 2) " NOTE THIS (MODIFY INSTRUCTION)
249 LD4 TABLE PTR1 " FREE SPACE
250 DECA 3 " IN SYMBOL TABLE
251 LD6 LIT CONS PTR " END OF SPACE FOR SYMBOLS
252 DECA 0,4 " SPACE AVAILABLE ?
253 J6P EXIT " ELSE EXIT
254 STZ TABLE,4 " CLEAN SPACE FOR CHARACTERS
255 INCA 9 + 1 " MODIFY THE CHARACTER (ADD 1)
256 STA TABLE,4 (1 : 1) " STORE THE FIRST CHARACTER
257 ENT5 1 : 1 " INITIALIZE BYTE POINTER
258 JMP READ CHAR " READ NEXT CHARACTER
259 DECA SPACE " IS IT A SPACE ?
260 JAZ READ CHAR " THEN SKIP ONE
261 LDA CHARACTER " CHARACTER
262 DECA LETTER Z " IS IT A LETTER OR A DIGIT ?
263 JANP 2F " THEN JUMP
264 ENN1 1 " INDICATOR FOR END OF
265 ST1 TABLE,4 (0 : 0) " ALPHA NUMERICAL INFORMATION
266 JMP COMPL " SYNTACTICAL UNIT COMPLETED
267 DECA 9 - LETTER Z " IS THE CHARACTER A LETTER ?
2HI
268 JANP * + 2 " ELSE JUMP
269 ST4 COMPL (0 : 2) " NOTE OCCURRENCE OF LETTER
270 ENN5 5 : 5 /N,5 " PREVIOUS CHARACTER IN BYTE 5 ?
271 J5P 3F " ELSE JUMP
272 ENT5 5 : 5 " INITIALIZE BYTE POINTER
273 DECA 1 " NEXT WORD WITH ALPHA NUMERICALS
274 LD6 LIT CONS PTR " END OF SPACE FOR SYMBOLS
275 DECA 0,4 " SPACE AVAILABLE ?
276 J6P EXIT " ELSE EXIT
277 STZ TABLE,4 " CLEAN SPACE FOR CHARACTERS
278 ENN5 6 : 6 /N,5 " POINTER TO NEXT BYTE
3HI
279 INCA 9 + 1 " MODIFY THE CHARACTER (ADD 1)
280 ST5 * + 1 (4 : 4) " MODIFY NEXT INSTRUCTION
```

0 21667.1 GERTENVELDEN				27		
29 86	+	13 28	04 00 30	STA	TABLE ,4 (MOD)	" STORE THE LETTER OR DIGIT
29 87	+	29 63	00 00 47	JMP	READ	" PROCEED WITH NEXT CHARACTERS
29 88	+	00 00	00 02 60	ENTA	MOD	" AT LEAST ONE LETTER READ ?
*29 74	+	29 88	00 02 34			
*29 71	+	29 88	00 00 47			
*29 53	+	29 88	00 02 30			
29 89			00 02 50	JAP	CONTAINS LETTER	" THEN JUMP, AND TREAT SYMBOL
29 90	+	28 28	00 05 11	L01	TABLE PTR1	" POINTER TO NEW INFO CELL
29 91	+	12 02	00 05 41	STZ	VALUE OF NUMBER	" INITIALIZE VALUE OF NUMBER
29 92	+	00 00	04 02 66	ENT6	0 ,4	" END OF INFO JUST STORED
29 93	-	00 04	01 01 66	DEC6	- 4 ,1	" LENGTH OF INFO
29 94			00 02 56	J6P	3F	" JUMP IF ONLY ONE WORD WITH DIGITS
29 95			00 01 56	J6Z	2F	" JUMP IF ONLY TWO WORDS WITH DIGITS
29 96	+	13 24	02 05 10	L0A	TABLE - 4 ,1	" FIFTH TILL TENTH DIGIT
29 97	+	28 44	00 05 02	S09	CORRECTION	" CANCEL MODIFICATION OF CHARACTERS
29 98	+	00 05	00 03 06	SRAX	5	" SHIFT DIGITS INTO RX
29 99	+	13 25	02 05 10	L0A	TABLE - 3 ,1	" FIRST TILL FIFTH DIGIT
30 00	+	28 44	00 05 02	SUB	CORRECTION	" CANCEL MODIFICATION OF CHARACTERS
30 01	+	00 00	00 00 05	NUM	VALUE OF NUMBER	" NUMERICAL VALUE OF TEN DIGITS
30 02	+	12 08	00 05 17	L0X	OVERFLOW	" MORE DIGITS TREATED ALREADY ?
30 03	+	12 08	00 02 57	JXP	VALUE OF NUMBER	" THEN OVERFLOW NOW
30 04	+	00 02	00 01 61	STA	2	" STORE VALUE OF THE TEN DIGITS
30 05	+	29 92	00 00 47	DEC1	18	" POINTER TO NEXT DIGITS
30 06	+	12 08	00 05 10	JMP	VALUE OF NUMBER	" CONTINUE THE PROCESS
30 07	+	30 07	00 01 56	LDA	"	" SHIFT VALUE OBTAINED
*29 95	+	28 40	00 05 03	MUL	HUNDREDTHOUSAND	" OVER FIVE DECIMAL POSITIONS
30 08	+		00 04 50	JANZ	OVERFLOW	" JUMP IF OVERFLOW OCCURS
30 09	+	12 08	00 05 37	STX	VALUE OF NUMBER	" STORE INTERMEDIATE VALUE
30 10	+	13 29	04 05 10	LDA	TABLE + 1 ,4	" FIVE LAST BUT FIVE DIGITS
30 11	+	28 44	00 05 02	S02	CORRECTION	" CANCEL MODIFICATION OF CHARACTERS
30 12	+	00 05	00 03 06	SRAX	5	" SHIFT DIGITS INTO RX, RA = 0
30 13	+	00 00	00 00 05	NUM	VALUE OF NUMBER	" NUMERICAL VALUE OF FIVE DIGITS
30 14	+	12 08	00 05 01	ADD	VALUE OF NUMBER	" ADD TO INTERMEDIATE VALUE
30 15	+	12 08	00 05 30	STA	0	" RESULTING VALUE (NO OVERFLOW)
30 16	+	00 00	00 02 60	ENTA		" RA = 0
30 17	+	30 27	00 02 56	ENNX	5 : 5 /N ,5	" NUMBER OF BYTES IN LAST WORD
*29 94	-	00 55	05 03 67	DIV	= 1 : 1 =	" WHICH HAVE NOT BEEN FILLED IN
30 18	+	30 21	00 02 30	STA	* + 1 (0 : 2)	" MODIFY NEXT INSTRUCTION
30 19	+	00 00	00 02 66	ENT6	MOD	" NUMBER OF BYTES NOT FILLED IN
30 20	+	28 40	00 05 10	LDA	HUNDREDTHOUSAND	" ONE, FOLLOWED BY
30 21	+	00 00	00 01 05	CHAR	0 ,6	" FIVE DECIMAL ZEROS
30 22	+	00 00	00 03 06	SRAX		" SHIFT OUT THE BYTES NOT FILLED IN
30 23	+	00 00	00 00 05	NUM		" NUMERICAL VALUE INDICATING
30 24	+	12 08	00 05 10	STA	AUX	" THE DECIMAL SHIFT OF VALUE OF NUMBER
30 25	+	00 05	00 03 66	LDA	VALUE OF NUMBER	" VALUE OF NUMBER
30 26	+	00 05	00 05 03	MUL	AUX	" SHIFT OVER DECIMAL POSITIONS
30 27	+	12 08	00 05 10	JANZ	OVERFLOW	" JUMP IF OVERFLOW OCCURS
30 28	+	00 04	00 05 37	STX	VALUE OF NUMBER	" STORE INTERMEDIATE VALUE
30 29	+	13 28	04 05 17	LDX	TABLE ,4	" LAST DIGITS IN RX
30 30	+	28 44	00 05 10	LDA	CORRECTION	" (MODIFIED) ZEROS IN RA
30 31	+	00 05	00 03 66	ENNK	- 5 ,6	" NUMBER OF DIGITS IN LAST WORD
30 32	+	00 00	06 02 06	SLAX	0 ,6	" SHIFT DIGITS INTO RA
30 33	+	28 44	00 05 02	SUB	CORRECTION	" CANCEL MODIFICATION OF CHARACTERS
30 34	+	00 05	00 03 06	SRAX	5	" SHIFT DIGITS INTO RX, RA = 0
30 35	+	00 00	00 00 05	NUM	VALUE OF NUMBER	" NUMERICAL VALUE OF LAST DIGITS
30 36	+	12 08	00 05 01	ADD	VALUE OF NUMBER	" ADD TO INTERMEDIATE VALUE
30 37	+	12 08	00 05 30	STA	VALUE OF NUMBER	" RESULTING VALUE (NO OVERFLOW)
30 38	+	01 02	00 02 60	ENTA	NUMBER	" THIS SYNTACTICAL UNIT

C90173-126 D 2166Z.1 GERTENVELDEN				28		
30 41	+	12 05 00 05 30	336	STA	SYNT UNIT	") IS A NUMBER
30 42	+	00 00 47	337	JMP	9F	" RETURN
30 43	+	12 08 00 05 41	338	STZ	VALUE OF NUMBER	" RESULTING VALUE
30 29	+	30 43 00 04 50				
30 09	+	30 43 00 04 50				
30 03	+	30 43 00 02 57				
30 44	+	00 11 00 02 66	339	ENT6	11	" REPORT ERROR:
30 45	+	29 26 00 00 47	340	JMP	ERRORM	" VALUE OF NUMBER TOO LARGE
30 46	+	30 40 00 00 47	341	JMP	1B	" READY
			342			
30 47	+	28 28 00 05 11	343	LD1	TABLE PTR1	" POINTER TO NEW INFO CELL
30 89	+	30 47 00 02 50				
30 48	+	00 03 04 01 61	344	DEC1	3,4	" ONLY ONE WORD WITH CHARACTERS ?
30 49	+	00 04 51	345	JXNZ	LOOK FOR SYMBOL	" ELSE ORDINARY SYMBOL
30 50	+	00 22 00 01 65	346	DEC5	2 : 2	" TWO CHARACTERS ?
30 51	+	00 04 55	347	JXNZ	LOOK FOR SYMBOL	" ELSE ORDINARY SYMBOL
30 52	+	13 28 04 11 11	348	LD1	TABLE ,4 (1 : 1)	" FIRST CHARACTER
30 53	+	00 10 00 01 61	349	DEC1	9 + 1	" IS IT A DIGIT ?
30 54	+	00 02 51	350	JXNZ	LOOK FOR SYMBOL	" ELSE ORDINARY SYMBOL
30 55	+	13 28 04 22 17	351	LDX	TABLE ,4 (2 : 2)	" SECOND CHARACTER
30 56	+	00 12 00 01 67	352	DECX	LETTER B + 1	" IS IT LETTER B ?
30 57	+	00 01 57	353	JXZ	B	" THEN JUMP
30 58	+	00 04 00 01 67	354	DECX	LETTER F - LETTER B	
30 59	+	00 01 57	355	JXZ	F	" JUMP IF IT IS LETTER F
30 60	+	00 02 00 01 67	356	DECX	LETTER H - LETTER F	
30 61	+	00 04 57	357	JXNZ	LOOK FOR SYMBOL	" JUMP IF IT IS NOT LETTER H
30 62	+	00 02 00 02 67	358	ENTX	LOCAL LABEL	" TYPE LOCAL LABEL
30 63	+	00 00 47	359	JMP	IF	" READY
30 64	+	00 04 00 02 67	360	ENTX	LOCAL FORWARD	" TYPE LOCAL FORWARD
30 65	+	30 64 00 01 57				
30 66	+	00 00 47	361	JMP	IF	" READY
30 67	+	00 03 00 02 67	362	ENTX	LOCAL BACKWARD	" TYPE LOCAL BACKWARD
30 68	+	30 66 00 01 57				
30 69	+	12 07 00 05 37	363	STX	TYPE	" STORE THE TYPE
30 70	+	30 67 00 00 47				
30 71	+	30 67 00 00 47				
30 72	+	00 10 01 03 61	364	ENN1	9 + 1 ,1	" POINTER (NEGATIVE-1) TO LOCAL SYMBOL
30 73	+	00 00 47	365	JMP	7F	" LOCAL SYMBOL FOUND
30 74	+	14 99 00 02 61	366	ENT1	LENGTH OF TABLE - 1	" ROOT OF BINARY TREE
30 75	+	30 70 00 04 57	367			
30 76	+	30 70 00 02 51				
30 77	+	30 70 00 04 55				
30 78	+	30 70 00 04 51				
30 79	+	00 03 01 02 65				
30 80	+	28 28 00 05 16				
30 81	+	00 03 00 01 66				
30 82	+	13 28 06 05 10				
30 83	+	13 28 05 15 70				
30 84	+	00 08 47				
30 85	+	13 28 05 05 17				
30 86	+	00 05 57				
30 87	+	00 05 50				
30 88	+	00 01 00 01 65				
30 89	+	00 01 00 01 66				
30 90	+	30 74 00 00 47				
30 91	+	30 83 00 05 57				
30 92	+	00 00 47				
30 93	+	00 06 47				
30 94	+	00 00 47				
30 95	+	00 06 47				

ADDRESS	DATA	INSTRUCTION	COMMENT
363	LDA	TABLE, 1 (2 : 3) " LEFT BRANCH	
364	JANZ	1F	" JUMP IF NOT EMPTY
365	ENNZ	0, 1	" NEW THREAD
366	LDA	TABLE PTR1	" NEW LEFT BRANCH
367	STA	TABLE, 1 (2 : 3) " STORE NEW LEFT BRANCH	
368	JMP	NEW	" CREATE NEW INFO CELL
369	STA	* + 1 (0 : 2)	" MODIFY NEXT INSTRUCTION
370	ENT1	MOD	" POINTER TO NEXT INFO CELL
371	JMP	NEXT	" CONTINUE THE SEARCH
372	LDA	TABLE, 1	" RIGHT BRANCH
373	JAP	1B	" JUMP IF IT IS NOT A THREAD
374	LDXN	TABLE, 1 (4 : 5) " TAKE THE THREAD (NEGATIVE)	
375	LDA	TABLE PTR1	" NEW RIGHT BRANCH
376	STA	TABLE, 1 (0 : 0) " NOT A THREAD	
377	STA	TABLE, 1 (4 : 5) " STORE RIGHT BRANCH	
378	LDA	TABLE PTR1	" POINTER TO NEW INFO CELL
379	STX	TABLE, 1	" STORE A THREAD
380	STZ	TABLE - 1, 1	" INITIALIZE
381	LDA	END MARKER	" NEW
382	STA	TABLE - 2, 1	" INFO CELL
383	DECA	1	" INCREASE TABLE POINTER
384	STA	6F	" ORDINARY SYMBOL STORED
385	JMP	TABLE PTR1	" POINTER TO NEW SYMBOL
386	LDA	LENGTH OF TABLE - 1	" FINAL THREAD
387	END MARKER	" JUMP IF ONLY ONE SYMBOL IN TREE	
388	NEW	TABLE - 1, 1 (0 : 1)	
389	TYPE	" SAVE TYPE OF SYMBOL	
390	LAST SYMBOL	" POINTER TO THIS SYMBOL	
391	SYMBOL	" THIS SYNTACTICAL UNIT	
392	SYNT UNIT	" IS A SYMBOL	
393	9F	" RETURN	
394	QUOTE - LETTER Z	" IS THE CHARACTER A QUOTE ?	
395	ELSE JUMP		
396	READ NEXT CHARACTER		
397	IS IT A NLCR ?		
398	ELSE SKIP UNTIL NLCR		
399	NLCR READ		
400	IS IT A NLCR ?		
401	ELSE JUMP		
402	" THIS SYNTACTICAL UNIT		
403	" IS A SYMBOL		
404	" RETURN		
405	" IS THE CHARACTER A QUOTE ?		
406	ELSE JUMP		
407	READ NEXT CHARACTER		
408	IS IT A NLCR ?		
409	ELSE SKIP UNTIL NLCR		
410	NLCR READ		
411	IS IT A NLCR ?		
412	ELSE JUMP		
413	" THIS SYNTACTICAL UNIT		
414	" IS A SYMBOL		
415	" RETURN		
416	" IS THE CHARACTER A QUOTE ?		
417	ELSE JUMP		
418	READ NEXT CHARACTER		
419	IS IT A NLCR ?		
420	ELSE SKIP UNTIL NLCR		
421	NLCR READ		
422	IS IT A NLCR ?		
423	ELSE JUMP		
424	" THIS SYNTACTICAL UNIT		
425	" IS A SEPARATOR		
426	SEMICOLON - NLCR	" IS THE CHARACTER A SEMICOLON ?	
427	ELSE JUMP		
428	" TEXT OF ONE INSTRUCTION		
429	" COMPLETED NOW		

2901/3-126 D 2166Z.1 GERTENVELDEN

31 33	+	31 27 00 00 47	430	JMP	30	3B	" SEPARATOR READ
31 34	-	00 24 00 01 60	431	DECA		DIV1 - SEMICOLON	" IS THE CHARACTER A DIVISION SIGN ?
31 35	+	31 34 00 04 50				OTHER	" ELSE JUMP
31 36	+	28 45 00 00 47	432	JANZ		READ CHAR	" READ NEXT CHARACTER
31 37	+	00 40 00 01 60	433	DECA		DIV1	" IS IT ALSO A DIVISION SIGN ?
31 38	+	00 04 00 04 50	434	JANZ		5F	" ELSE JUMP
31 39	+	01 04 00 02 60	435	ENTA		DIV2	") THIS SYNTACTICAL UNIT
31 40	+	00 00 00 00 47	436	JMP		8F	") IS A SPECIAL DIVISION OPERATOR
31 41	+	00 40 00 02 60	437	ENTA		DIV1	" COMMON DIVISION OPERATOR
31 42	+	31 41 00 04 50	438				
31 43	+	12 05 00 05 30	439	STA		SYNT UNIT	" THIS SYNTACTICAL UNIT
31 44	+	00 00 00 00 47	440	JMP		9F	" RETURN
31 45	+	00 40 00 00 60	441	INCA		DIV1	" OTHER SINGLE CHARACTER
31 46	+	12 05 00 05 30	442	STA		SYNT UNIT	" THIS SYNTACTICAL UNIT
31 47	+	31 45 00 00 47					
31 48	+	31 45 00 00 47	443	ENTX		SPACE	" SIMULATE A SPACE
31 49	+	00 66 00 02 67	444	STX		CHAR	" INSERT IT
31 50	+	12 00 00 05 37	445	JMP		MOD	" RETURN
31 51	+	00 00 00 00 47					
31 52	+	31 46 00 00 47					
31 53	+	31 48 00 00 47					
31 54	+	31 48 00 00 47					
31 55	+	31 48 00 00 47					
31 56	+	31 48 00 00 47					
31 57	+	31 48 00 00 47					
31 58	+	31 48 00 00 47					
31 59	+	31 48 00 00 47					
31 60	+	31 48 00 00 47					
31 61	+	31 48 00 00 47					
31 62	+	31 48 00 00 47					
31 63	+	31 48 00 00 47					
31 64	+	31 48 00 00 47					
31 65	+	31 48 00 00 47					
31 66	+	31 48 00 00 47					
31 67	+	31 48 00 00 47					
31 68	+	31 48 00 00 47					
31 69	+	31 48 00 00 47					
31 70	+	31 48 00 00 47					
31 71	+	31 48 00 00 47					
31 72	+	31 48 00 00 47					
31 73	+	31 48 00 00 47					
31 74	+	31 48 00 00 47					
31 75	+	31 48 00 00 47					
31 76	+	31 48 00 00 47					
31 77	+	31 48 00 00 47					

RA IS AN OUTPUT PARAMETER FOR THE VALUE OF THE AT EXPR

9F " SAVE RETURN ADDRESS

READ SYNT UNIT " READ ONE SYNTACTICAL UNIT

* + 3

9F " SAVE RETURN ADDRESS

SYNT UNIT " LAST SYNTACTICAL UNIT

SYMBOL " IS IT A SYMBOL ?

SYMB " THEN JUMP

NUMBER - SYMBOL " OR IS IT A NUMBER ?

NUMB " THEN JUMP

ASTERISK - NUMBER

ASTSK " JUMP IF IT IS AN ASTERISK

21 " REPORT ERROR:

ERRORM " ATOMIC EXPRESSION REQUIRED

C " RESULTING VALUE

8F " READY

LAST SYMBOL " POINTER TO INFO CELL

TYPE " TYPE OF SYMBOL

DEFINED SYMB " IS IT A DEFINED SYMBOL ?

DEFINED " THEN JUMP

LOCAL BACKWARD - DEFINED SYMB

LOC DEFINED " JUMP IF IT IS A LOCAL BACKWARD

23 " REPORT ERROR:

1B " UNDEFINED SYMBOL IN EXPRESSION

TABLE - 2 , 1

8F " READY

HERE , 1 (3 : 3) " CORRESPONDING LABEL OCCURRED ?

2F " THEN JUMP

22 " REPORT ERROR:

1B " UNDEF. LOCAL BACKWARD IN EXPRESSION

2901/3-126	D 2166Z.1 GERTENVELDEN	31	2H:	LOA	BACKWARD ,1	" VALUE OF LOCAL BACKWARD
31 75	+ 12 38 01 05 10	478		LOA		
32 75	+ 31 78 00 02 50					
31 77	00 00 00 47	479		JMP	8F	" READY
32 80	+ 12 08 00 05 10	480	NUMB:	LOA	VALUE OF NUMBER	" VALUE OF NUMBER
33 57	+ 31 80 00 01 50					
31 81	00 00 00 47	481		JMP	8F	" READY
32 82	+ 12 19 00 05 10	482	ASTSK:	LOA	LC	" VALUE OF LOCATION COUNTER
33 59	+ 31 82 00 01 50					
31 83	+ 12 10 00 05 30	483	8H:	STA	AT EXPR	" VALUE OF ATOMIC EXPRESSION
32 81	+ 31 83 00 00 47					
33 79	+ 31 83 00 00 47					
31 85	+ 31 83 00 00 47					
32 65	+ 31 83 00 00 47					
31 84	+ 00 00 00 00 47	484	9H:	JMP	MOD	" RETURN
33 22	+ 31 84 00 02 40					
32 49	+ 31 84 00 02 40					
READ EXPRESSION: " RA IS AN OUTPUT PARAMETER FOR THE VALUE OF THE EXPRESSION						
31 82	00 02 40	485		STC	9F	" SAVE RETURN ADDRESS
32 82	00 02 40	486		STZ	EXPR	" INITIALIZE EXPR
33 87	+ 12 11 00 05 41	487		LOA	SYNT UNIT	" LAST SYNTACTICAL UNIT
32 88	+ 00 37 00 01 60	488		DECA	PLUS	" IS IT AN ADDING OPERATOR ?
33 89	00 01 50	489		JAZ	ADD	" THEN JUMP
31 90	+ 00 01 00 01 60	490		DECA	MINUS - PLUS	" IS IT A SUBTRACTING OPERATOR ?
32 91	00 01 50	491		JAZ	SUB	" THEN JUMP
33 92	+ 31 52 00 00 47	492		JMP	AT EXPRESSION	" EXPECT AN ATOMIC EXPRESSION
31 93	+ 12 11 00 05 30	493	NEXT OPERAND:	STA	EXPR	" SAVE THE VALUE OF IT
32 94	00 02 47	494		JOV	ARITHM OV	" IF OVERFLOW THEN REPORT IT
33 95	+ 29 44 00 00 47	495		JMP	READ SYNT UNIT	" READ NEXT SYNTACTICAL UNIT
31 96	+ 00 37 00 01 60	496		DECA	PLUS	" IS IT AN ADDING OPERATOR ?
32 97	00 01 50	497		JAZ	ADD	" THEN JUMP
33 98	+ 00 01 00 01 60	498		DECA	MINUS - PLUS	" IS IT A SUBTRACTING OPERATOR ?
32 99	00 01 50	499		JAZ	SUB	" THEN JUMP
31 00	+ 00 01 00 01 60	500		DECA	TIMES - MINUS	" IS IT A MULTIPLICATION OPERATOR ?
32 01	00 01 50	501		JAZ	MUL	" THEN JUMP
33 02	+ 00 01 00 01 60	502		DECA	DIV1 - TIMES	" IS IT A COMMON DIVISION OPERATOR ?
32 03	00 01 50	503		JAZ	DIV X	" THEN JUMP
33 04	+ 00 64 00 01 60	504		DECA	DIV2 - DIV1	" IS IT A SPECIAL DIVISION OPERATOR ?
32 05	00 01 50	505		JAZ	DIV A	" THEN JUMP
33 06	+ 00 41 00 01 60	506		DECA	DEC - DIV2	" IS IT A DECIMAL OPERATOR ?
32 07	00 01 50	507		JAZ	TIMES BASE PLUS	" THEN JUMP
33 08	+ 12 11 00 05 10	508		LOA	EXPR	" RESULTING VALUE OF EXPRESSION
32 09	00 00 00 00 47	509		JMP	MOD	" RETURN
31 10	+ 32 09 00 02 40	510				
32 10	+ 12 11 00 05 10	511	TIMES BASE PLUS:	LDA	EXPR (1 : 5)	" ABSOLUTE VALUE OF EXPRESSION
33 11	00 01 50	512		MUL	BASE	" MULTIPLY BY TEN (BASE)
32 12	+ 28 35 00 05 03	513		STX	EXPR (1 : 5)	" RESTORE THE VALUE
33 13	+ 12 11 00 05 37	514		ADD	MAX WORD	" FORCE OVERFLOW IF RA # 0
32 14	+ 28 38 00 05 01	515	ADD:	JMP	READ AT EXPRESSION	
33 15	+ 31 49 00 00 47	516				
32 16	+ 32 14 00 01 50	517				
31 17	+ 32 14 00 01 50	518				
32 18	+ 32 14 00 01 50	519		ADD	EXPR	" EXECUTE ADDING OPERATION
33 19	+ 12 11 00 05 01	520		JMP	NEXT OPERAND	" PROCEED WITH NEXT OPERANDS
32 20	+ 31 93 00 00 47	521	SUB:	JMP	READ AT EXPRESSION	
33 21	+ 31 49 00 00 47	522				
32 22	+ 32 17 00 01 50					
31 23	+ 32 17 00 01 50					
32 24	+ 12 11 00 05 10			LOA	EXPR	") EXECUTE

2901/3-126	D 21662.1	GERTENVELDEN	33
32 69	+	29 26 00 00 47	576
32 70	+	32 62 00 00 47	577
32 71	+	12 12 00 05 17	578
32 72	+	32 71 00 04 50	579
32 73	+	12 13 00 05 37	580
32 74	+	29 44 00 00 47	581
32 75	+	32 74 00 01 50	582
32 76	+	12 05 00 05 10	583
32 77	+	32 75 00 00 47	584
32 78	+	00 60 00 01 60	585
32 79	+	32 42 00 01 50	586
32 80	+	12 13 00 05 10	587
32 81	+	00 00 00 00 47	588
32 82	+	32 79 00 02 40	589
32 83	+	28 37 00 05 10	590
32 84	+	12 22 00 05 30	591
32 85	+	29 44 00 00 47	592
32 86	+	01 03 00 01 60	593
32 87	+	00 01 50	594
32 88	+	00 02 00 01 60	595
32 89	+	32 83 00 01 50	596
32 90	+	00 51 00 02 66	597
32 91	+	29 26 00 00 47	598
32 92	+	32 83 00 00 47	599
32 93	+	12 24 00 05 31	600
32 94	+	32 91 00 01 50	601
32 95	+	29 44 00 00 47	602
32 96	+	00 63 00 01 60	603
32 97	+	12 07 00 05 10	604
32 98	+	00 01 50	605
32 99	+	00 02 00 01 60	606
33 00	+	00 52 00 02 66	607
33 01	+	32 89 00 00 47	608
33 02	+	13 26 01 02 66	609
33 03	+	33 01 00 01 50	610
33 04	+	12 02 00 05 10	611
33 05	+	13 27 01 23 30	612
33 06	+	33 06 00 00 47	613
33 07	+	12 48 01 02 66	614
33 08	+	33 05 00 01 50	615
33 09	+	00 00 06 02 10	616
33 10	+	00 53 00 02 66	617
33 11	+	32 89 00 00 47	618
33 12	+	12 22 00 05 31	619
33 13	+	32 83 00 00 47	620
33 14	+	12 19 00 05 10	621
33 15	+	33 14 00 04 50	622
33 16	+	00 00 06 02 30	623
33 17	+	00 00 00 00 47	624
33 18	+	32 83 00 00 47	625
33 19	+	12 19 00 05 10	626
33 20	+	33 14 00 04 50	627
33 21	+	00 00 06 02 30	628
33 22	+	00 00 00 00 47	629
33 23	+	32 83 00 00 47	630
33 24	+	12 19 00 05 10	631
33 25	+	33 14 00 04 50	632
33 26	+	00 00 06 02 30	633
33 27	+	00 00 00 00 47	634
33 28	+	32 83 00 00 47	635
33 29	+	12 19 00 05 10	636
33 30	+	33 14 00 04 50	637
33 31	+	00 00 06 02 30	638
33 32	+	00 00 00 00 47	639
33 33	+	32 83 00 00 47	640
33 34	+	12 19 00 05 10	641
33 35	+	33 14 00 04 50	642
33 36	+	00 00 06 02 30	643
33 37	+	00 00 00 00 47	644
33 38	+	32 83 00 00 47	645
33 39	+	12 19 00 05 10	646
33 40	+	33 14 00 04 50	647
33 41	+	00 00 06 02 30	648
33 42	+	00 00 00 00 47	649
33 43	+	32 83 00 00 47	650
33 44	+	12 19 00 05 10	651
33 45	+	33 14 00 04 50	652
33 46	+	00 00 06 02 30	653
33 47	+	00 00 00 00 47	654
33 48	+	32 83 00 00 47	655
33 49	+	12 19 00 05 10	656
33 50	+	33 14 00 04 50	657
33 51	+	00 00 06 02 30	658
33 52	+	00 00 00 00 47	659
33 53			

2901/5-125	D 21662.1 GERTENVELDEN	34							
33 12	+	12 23 00 05 30	626	STA	EQUIV	" INITIALIZE EQUIV			
33 16	+	12 24 00 05 10	627	LDA	INST	" INSTRUCTION SYMBOL			
33 17	+	28 29 00 05 02	628	SUB	LAST INST	" IS IT A MIX MACHINE INSTRUCTION ?			
33 18	+	00 00 00 50	629	JAN	6F	" ELSE JUMP			
33 19	+	12 24 00 05 14	630	LDA	INST	" POINTER TO INSTRUCTION SYMBOL			
33 20	+	00 00 00 47	631	JMP	READ MIX MACHINE INST				
33 21	+	12 19 00 05 17	632	LDX	LC	" LOCATION COUNTER			
33 22	+	00 00 00 47	633	JMP	PRODUCE	" PRODUCE CODE			
33 23	+	12 05 00 05 10	634	LDA	SYNT UNIT	" SYNTACTICAL UNIT			
33 24	+	01 01 00 01 60	635	DECA	SEP	" IS IT A SEPARATOR ?			
33 25	+	00 01 00 01 50	636	JAZ	2F	" THEN C.K.			
33 26	+	00 55 00 02 66	637	ENT6	55	" REPORT ERROR:			
33 27	+	29 26 00 00 47	638	JMP	ERRORM	" SEPARATOR REQUIRED			
33 28	+	29 44 00 00 47	639	JMP	READ SYNT UNIT	") SKIP			
33 29	+	01 01 00 01 60	640	DECA	SEP	") UNTIL			
33 30	+	33 28 00 04 50	641	JANZ	1B	") SEPARATOR			
33 31	+	12 22 00 05 15	642	LDS	LAST LABEL	" CHAIN OF LABELS			
33 32	+	33 31 00 01 50	643	JMP	UPDATE	" UPDATE THIS CHAIN			
33 33	+	32 81 00 00 47	644	JMP	NEXT INST	" PROCEED WITH NEXT INSTRUCTION			
33 34	+	12 24 00 05 10	645	LDA	INST	" INSTRUCTION SYMBOL			
33 35	+	33 34 00 00 50	646	SUB	ORIG	" IS IT ORIG ?			
33 36	+	28 34 00 05 02	647	JANZ	6F	" ELSE TRY OTHER			
33 37	+	32 39 00 00 47	648	JMP	READ W VALUE	" READ THE WORD VALUE			
33 38	+	12 19 00 05 30	649	STA	LC	" NEW VALUE FOR LOCATION COUNTER			
33 39	+	33 23 00 00 47	650	JMP	5B	" GENERAL TREATMENT			
33 40	+	12 24 00 05 10	651	LDA	INST	" INSTRUCTION SYMBOL			
33 41	+	33 40 00 04 50	652	SUB	EQU	" IS IT EQU ?			
33 42	+	28 33 00 05 02	653	JANZ	6F	" ELSE TRY OTHER			
33 43	+	32 39 00 00 47	654	JMP	READ W VALUE	" READ THE WORD VALUE			
33 44	+	12 23 00 05 30	655	STA	EQUIV	" NEW VALUE FOR EQUIV			
33 45	+	33 23 00 00 47	656	JMP	5B	" GENERAL TREATMENT			
33 46	+	12 24 00 05 10	657	LDA	INST	" INSTRUCTION SYMBOL			
33 47	+	33 46 00 04 50	658	SUB	CON	" IS IT CON ?			
33 48	+	28 31 00 05 02	659	JANZ	6F	" ELSE TRY OTHER			
33 49	+	32 39 00 00 47	660	JMP	READ W VALUE	" READ THE WORD VALUE			
33 50	+	33 21 00 00 47	661	JMP	4B	" GENERAL TREATMENT			
33 51	+	12 24 00 05 10	662	LDA	INST	" INSTRUCTION SYMBOL			
33 52	+	33 51 00 04 50	663	SUB	ALF	" IS IT ALF ?			
33 53	+	28 30 00 05 02	664	JANZ	6F	" ELSE TRY OTHER			
33 54	+	28 94 00 00 47	665	JMP	READ 5 CHAR	" READ THE FIVE CHARACTERS			
33 55	+	33 21 00 00 47	666	JMP	4B	" GENERAL TREATMENT			
33 56	+	12 24 00 05 10	667	LDA	INST	" INSTRUCTION SYMBOL			
33 57	+	33 56 00 04 50	668	SUB	END	" IS IT END ?			
33 58	+	28 32 00 05 02	669	JANZ	6F	" ELSE TRY OTHER			
33 59	+	32 39 00 00 47	670	JMP	READ W VALUE	" READ THE WORD VALUE			
33 60	+	12 27 00 05 30	671	STA	START ADDRESS	" VALUE FOR START ADDRESS			
33 61	+	00 00 00 47	672	JMP	INSERT CONSTANTS	" INSERTION OF CONSTANTS			
33 62	+	12 19 00 05 10	673	LDA	LC	" LOCATION COUNTER (AFTER INSERT)			
33 63	+	12 23 00 05 30	674	STA	EQUIV	" NEW VALUE FOR EQUIV			
33 64	+	12 22 00 05 15	675	LDS	LAST LABEL	" CHAIN OF LABELS PRECEDING END			
33 65	+	00 00 00 47	676	JMP	UPDATE	" UPDATE THIS CHAIN			
33 66	+	33 66 00 02 40	677	JMP	MOD	" RETURN			
33 67	+	00 54 00 02 66	678	ENT6	54	" REPORT ERROR:			

34 11	+	13 28 06 05 30	731	STA	TABLE ,6	") STACK
34 12	+	00 01 00 00 66	732	INC6	1	") THE
34 13	+	12 26 00 05 36	733	ST6	LIT CONS PTR	") WORD VALUE
34 14	+	28 28 00 05 14	734	LD4	TABLE PTR1	" TABLE POINTER
34 15	+	00 00 06 01 64	735	DECA	0 ,6	" SPACE AVAILABLE ?
34 16	+	00 00 00 00 54	736	JAN	EXIT	" ELSE EXIT
34 17	+	12 05 00 05 10	737	LDA	SYNT UNIT	" SYNTACTICAL UNIT
34 18	+	00 43 00 01 60	738	DECA	EQUAL SIGN	" IS IT AN EQUAL SIGN ?
34 19	+	12 25 00 02 66	739	ENT6	LAST CONSTANT	" CHAIN OF LITERAL CONSTANTS
34 20	+	00 02 36	740	ST6	FUT REF (0 : 2)	" MODIFY INSTRUCTION
34 21	+	00 01 50	741	JAZ	2F	" READ, AND EXTEND CHAIN
34 22	+	00 61 00 02 66	742	ENT6	61	" REPORT ERROR:
34 23	+	29 26 00 00 47	743	JMP	ERRORM	" = REQUIRED
34 24	+	00 00 47	744	JMP	FUT REF	" EXTEND CHAIN OF FUTURE REFERENCES
34 25	+	12 07 00 05 10	745	LDA	TYPE	" UNDEFINED SYMBOL ?
34 26	+	34 25 00 01 50				
34 27	+	13 26 01 02 66	746	JANZ	LOC	" ELSE TRY LOCAL FORWARD
34 28	+	00 00 47	747	ENT6	TABLE - 2 ,1	" POINTER TO FIELD VALUE2
34 29	+	00 04 00 01 60	748	JMP	1F	" TREAT FUTURE REFERENCE
34 30	+	34 29 00 04 50	749	DECA	LOCAL FORWARD	" LOCAL FORWARD OCCURRING ?
34 31	+	34 05 00 04 50	750	JANZ	COMMON EXPR	" ELSE TRY COMMON EXPRESSION
34 32	+	12 48 01 02 66	751	ENT6	HERE ,1	" POINTER TO FIELD VALUE2
34 33	+	00 02 36	752	ST6	FUT REF (0 : 2)	" MODIFY INSTRUCTION
34 34	+	34 32 00 00 47	753	JMP	2H1	" READ SYNT UNIT
34 35	+	29 44 00 00 47	754	ENT6	MOD	" NEXT SYNTACTICAL UNIT
34 36	+	34 33 00 01 50				" RESTORE R16
34 37	+	00 00 06 45 10	755	LDA	0 ,6 (VALUE2)	") CHAIN OF FUTURE REFERENCES
34 38	+	12 15 00 05 30	756	STA	APART	") INTO A PART
34 39	+	00 01 00 02 64	757	ENT4	1	
34 40	+	12 20 00 05 34	758	ST4	FUTURE	" NOTE OCCURRENCE OF FUTURE REFERENCE
34 41	+	12 19 00 05 17	759	LDX	LC	" LOCATION COUNTER
34 42	+	35 70 00 00 47	760	JMP	POSSIBLE	" IS IT A POSSIBLE ADDRESS ?
34 43	+	00 05 51	761	JANP	CHECK COMMA	" ELSE SKIP ADMINISTRATION
34 44	+	12 19 00 05 14	762	LD4	LC	" LOCATION COUNTER
34 45	+	00 00 04 05 10	763	LDA	0 ,4	" CONTENTS OF THIS ADDRESS
34 46	+	00 04 50	764	JANZ	CHECK COMMA	" JUMP IF USED ALREADY
34 47	+	12 21 00 05 10	765	LDA	DANGEROUS	" DANGEROUS ADDRESS
34 48	+	00 00 04 01 60	766	DECA	0 ,4	" EQUAL TO THIS ADDRESS ?
34 49	+	00 01 50	767	JAZ	CHECK COMMA	" THEN SKIP ADMINISTRATION
34 50	+	00 00 06 45 34	768	ST4	0 ,6 (VALUE2)	" JOIN THE ADDRESS TO THE CHAIN
34 51	+	12 05 00 05 10	769	LDA	SYNT UNIT	" SYNTACTICAL UNIT
34 52	+	34 49 00 01 50				
34 53	+	34 49 00 04 50				
34 54	+	00 00 01 60	770	DECA	COMMA	" IS IT A COMMA ?
34 55	+	29 44 00 00 47	771	JANZ	CHECK PAREN	" ELSE TRY FIELD SPECIFICATION
34 56	+	34 52 00 01 50	772	JMP	READ SYNT UNIT	" NEXT SYNTACTICAL UNIT
34 57	+	31 85 00 00 47	773	JMP	READ EXPRESSION	" READ 1 PART
34 58	+	12 16 00 05 30	774	STA	1 PART	" SAVE THE VALUE
34 59	+	12 05 00 05 10	775	LDA	SYNT UNIT	" SYNTACTICAL UNIT
34 60	+	34 55 00 04 50				
34 61	+	00 71 00 01 60	776	DECA	PAREN	" IS IT A PAREN ?
34 62	+	00 00 45	777	JANZ	CHECK VALUES	" ELSE JUMP

```

2901/3-126      0 2166Z.1 GERTENVELDEN      37
34 28      + 29 44 00 00 47      778      PAREN CHECKED:  JMP      READ SYNT UNIT  " NEXT SYNTACTICAL UNIT
34 29      + 34 58 00 01 50      779      READ EXPRESSION  " READ F PART
34 30      + 31 85 00 00 47      780      F PART          " SAVE THE VALUE
34 31      + 12 17 00 05 30      781      SYNT UNIT      " SYNTACTICAL UNIT
34 32      + 12 05 00 05 10      782      THESIS        " IS IT A THESIS ?
34 33      + 00 72 00 01 60      783      DECA          " ELSE ERROR
34 34      + 29 44 00 04 50      784      JANZ         " NEXT SYNTACTICAL UNIT
34 35      + 28 37 00 05 10      785      JMP          " MAX, VALUE OF A PART
34 36      + 34 65 00 04 50      786      LDA          " ABS(A PART) O.K. ?
34 37      + 12 15 00 05 70      787      CMPA        " THEN CHECK I PART
34 38      + 00 63 00 02 66      788      JGE         " REPORT ERROR:
34 39      + 29 26 00 00 47      789      ENT6        " VALUE OF A PART NOT O.K.
34 40      + 12 15 00 05 41      790      JMP         " REPLACE ERRONEOUS A PART
34 41      + 28 36 00 05 10      791      STZ         " MAX, VALUE OF I PART
34 42      + 34 71 00 07 47      792      LDA          1H:
34 43      + 12 16 00 05 70      793      CMPA        " I PART TOO LARGE ?
34 44      + 00 63 00 02 66      794      JLE         " THEN ERROR
34 45      + 12 16 00 05 17      795      LDX         " I PART 2 0 ?
34 46      + 00 64 00 02 66      796      JXNN        " THEN CHECK F PART
34 47      + 34 76 00 04 47      797      ENT6        " REPORT ERROR:
34 48      + 29 26 00 00 47      798      JMP         " VALUE OF I PART NOT O.K.
34 49      + 12 16 00 05 41      799      STZ         " REPLACE ERRONEOUS I PART
34 50      + 28 36 00 05 10      800      LDA         " MAX VALUE OF F PART
34 51      + 12 17 00 05 70      801      CMPA        " F PART TOO LARGE ?
34 52      + 34 80 00 03 57      802      JL          " THEN ERROR
34 53      + 12 17 00 05 17      803      LDX         " F PART 2 0 ?
34 54      + 00 65 00 02 66      804      JXNN        " THEN CHECKS O.K.
34 55      + 34 84 00 04 47      805      ENT6        " REPORT ERROR:
34 56      + 29 26 00 00 47      806      JMP         " VALUE OF F PART NOT O.K.
34 57      + 12 17 00 05 41      807      STZ         " REPLACE ERRONEOUS F PART
34 58      + 12 15 00 05 10      808      LDA         " )
34 59      + 34 87 00 03 57      809      JMP         INSTRUCTION (0 : 2)  " ASSEMBLE
34 60      + 29 44 00 01 50      810      STA         " I PART
34 61      + 12 18 00 02 30      811      LDA         INSTRUCTION (3 : 3)  " ) THE
34 62      + 12 16 00 05 10      812      STA         " ) WHOLE
34 63      + 12 18 00 03 30      813      LDA         F PART
34 64      + 12 17 00 05 10      814      STA         INSTRUCTION (4 : 4)  " ) INSTRUCTION
34 65      + 12 18 00 04 30      815      LDA         " RESULTING VALUE
34 66      + 00 00 00 00 47      816      JMP         MOD
34 67      + 34 94 00 02 40      817      ENT6        " RETURN
34 68      + 00 62 00 02 66      818      JMP         " REPORT ERROR:
34 69      + 34 95 00 04 50      819      JMP         " ) REQUIRED
34 70      + 29 26 00 00 47      820      JMP         CHECK VALUES
34 71      + 34 65 00 00 47      821      " RA CONTAINS THE VALUE TO BE PRODUCED,
34 72      + 34 98 00 00 47      822      " RX CONTAINS THE ADDRESS
34 73      + 33 70 00 00 47      823      STJ         " SAVE RETURN ADDRESS
34 74      + 00 71 00 02 66      824      JMP         " POSSIBLE ADDRESS TO BE FILLED IN ?
34 75      + 29 26 00 00 47      825      JIP         " JUMP IF POSSIBLE ADDRESS
34 76      + 34 96 00 00 47      826      ENT6        " REPORT ERROR:
34 77      + 34 97 00 00 47      827      JMP         " WRONG ADDRESS

```

04

38

021667.1 GERTENVELDEN

021667.1 GERTENVELDEN

Address	Instruction	Comment
827	LDX	" IMPOSSIBLE ADDRESS
828	JXNM	" JUMP IF NOT NEGATIVE
829	ST4	" 'PRINT'
830	ST4	" 'MINUS SIGN
831	JMP	" 'PRINT'
832	ENT4	" 'PLUS SIGN
833	ST4	" 'RA = IMPOSSIBLE ADDRESS, RX = 0
834	SLAX	" CONVERT TO CHARACTERS
835	CHAR	" 'PRINT'
836	ST4	" 'PRINT'
837	STX	" THEN
838	JMP	" GIVE UP
839	LD4	" IF CODE PRODUCED ALREADY
840	J4P	" THEN PRINT THE LINE
841	LD1	" POSSIBLE ADDRESS
842	LDX	" LOCATION COUNTER
843	DECX	" ARE THEY UNEQUAL ?
844	JXNZ	" THEN SKIP TEST
845	LDX	" CONTENTS OF ADDRESS, ZERO ?
846	JXNZ	" ELSE ERROR
847	LDX	" ADDRESS UNEQUAL
848	DECX	" TO DANGEROUS ONE ?
849	JXNZ	" THEN O.K.
850	ENT4	" REPORT ERROR;
851	JMP	" ADDRESS ALREADY USED
852	LD1	" PRODUCE OUTPUT ONLY
853	JMP	" SAVE THE VALUE TO BE PRODUCED
854	ST4	"
855	JANZ	" JUMP IF IT IS NOT ZERO
856	LDX	" FUTURE REFERENCE TO BE PRODUCED ?
857	JXNP	" ELSE JUMP
858	ST1	" DANGEROUS FUTURE REFERENCE PRODUCED
859	CHAR	" CONVERT THE VALUE TO CHARACTERS
860	ST4	"
861	STX	" 'PRINT' SIX DECIMAL POSITIONS
862	LD4	" FUTURE REFERENCE ?
863	J4P	" THEN OMIT A PART
864	ST4	" 'PRINT' A PART
865	JANZ	" JUMP IF VALUE > 0
866	ENT4	"
867	ST4	" 'PRINT' MINUS SIGN
868	JMP	"
869	ENT4	"
870	ST4	" 'PRINT' PLUS SIGN
871	LD4	" VALUE TO BE PRODUCED
872	ST4	"
873	ENT4	" STORE IT INTO MEMORY
874	CHAR	" CONVERT TO CHARACTERS
875	STX	" 'PRINT' IT
876	LDX	" LOCATION COUNTER
877	DECX	" UNEQUAL TO ADDRESS ?

290270-126 D 2166Z.1 GERTENVELDEN

```

36 44 + 36 32 00 00 47
36 45 00 02 40
36 46 + 12 03 00 05 10
36 47 + 12 04 00 05 01
36 48 + 29 16 00 02 50
36 49 + 00 08 00 02 64
36 50 + 00 11 00 02 65
36 51 + 36 24 00 00 47
36 52 + 00 63 00 02 60
36 53 + 36 15 00 00 47
36 54 + 13 16 00 11 10
36 55 + 00 66 00 01 60
36 56 + 29 16 00 04 50
36 57 + 00 12 00 02 64
36 58 + 00 11 00 02 65
36 59 + 28 31 00 05 12
36 60 + 36 24 00 00 47
36 61 + 00 00 00 02 62
36 62 + 36 62 00 02 32
36 63 + 00 01 00 00 64
36 64 + 13 04 04 22 41
36 65 + 00 00 00 00 47
36 66 + 36 65 00 02 40

```

982
 983
 984
 985
 986
 987
 988
 989
 990
 991
 992
 993
 994
 995
 996
 997
 998
 999
 1000
 1001
 1002
 1003
 1004
 1005
 1006
 1007
 1008
 1009
 1010
 1011
 1012
 1013
 1014
 1015
 1016
 1017
 1018
 1019
 1020
 1021
 1022
 1023
 1024
 1025
 1026
 1027
 1028
 1029
 1030
 1031
 1032
 1033
 1034

1M: 1M: 9M: 9H:

-58-

```

DEFINE: " R12 CONTAINS A POINTER TO THE SYMBOL TO BE DEFINED
          9F
          STJ " SAVE RETURN ADDRESS
          LDA TABLE - 1,2 (0:1) " DEFINED SYMBOL ?
          JAP 9F " THEN READY
          LDA TABLE - 2,2 (VALUE1) " CONTENTS OF FIELD VALUE1
          JANZ 9F " READY IF NOT INITIAL VALUE
          ENTZ TABLE - 2,2 " POINTER TO FIELD VALUE2
          " R 3 CONTAINS A POINTER TO FIELD VALUE2
          9F
          STJ " SAVE RETURN ADDRESS (AGAIN)
          LDA 0,3 (VALUE2) " CHAIN OF FUTURE REFERENCES
          CMPA END MARKER " EMPTY CHAIN ?
          JE 9F " THEN READY
          JAP BUFFER CONSTANT " PRINT A CONSTANT INSTRUCTION
          LC " LOCATION COUNTER
          STX EQU'V " NEW VALUE FOR EQUIV
          0 " WORD VALUE OF CONSTANT INSTRUCTION
          PRODUCE " PRODUCE IT
          END MARKER " ) SIMULATE OCCURRENCE
          STA 0,3 (VALUE1) " ) AS LABEL (FOR UPDATE)
          ENT5 0,2 " PARAMETER FOR UPDATE
          JMP UPDATE " UPDATE THE CHAIN
          9H: MOD " RETURN

```

INSERT CONSTANTS: " NO PARAMETERS " SAVE RETURN ADDRESS
 STJ 9F

2001/3-126 D 2166Z.1 GERTENVELDEN

```

*35 61 + 36 86 00 00 47
35 87 + 00 01 00 32 60
35 88 + 12 03 00 05 30
35 89 - 00 10 00 02 62
35 90 + 12 48 02 02 63
35 91 + 36 72 00 00 47
35 92 + 00 01 00 00 62
35 93 + 36 90 00 00 52
35 94 + 36 66 00 02 64
35 95 + 36 01 00 00 47
35 96 + 12 25 00 05 15
35 97 + 28 37 00 05 75
35 98 + 00 00 00 05 47
35 99 + 36 98 00 02 40
36 00 + 00 00 05 02 10
36 01 + 12 25 00 05 30
36 02 + 12 19 00 05 17
36 03 + 00 00 05 02 37
36 04 + 12 26 00 05 16
36 05 + 00 01 00 01 66
36 06 + 12 26 00 05 36
36 07 + 13 28 05 05 10
36 08 + 34 98 00 00 47
36 09 + 00 00 05 02 67
36 10 + 00 00 05 05 10
36 11 + 34 98 00 00 47
36 12 + 36 96 00 00 47

```

1H: 1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046

2H: 1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063

9H: 1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076

```

1 CODE
2 A NEW LINE IF NECESSARY
3 INITIALIZE LOCAL SYMBOL POINTER
4 POINTER TO FIELD VALUE2
5 DEFINE THE SYMBOL IF NECESSARY
6 INCREASE POINTER, READY ?
7 ELSE CONTINUE
8 PARAMETER FOR TRAVERSE TREE
9 DEFINE SYMBOLS IF NECESSARY
10 CHAIN OF LITERAL CONSTANTS
11 IS IT EMPTY ?
12 THEN RETURN

```

```

13 DELETE LAST CONSTANT
14 FROM CHAIN
15 LOCATION COUNTER
16 UPDATE A PART OF ADDRESS
17 UNSTACK
18 VALUE FOR
19 LITERAL CONSTANT
20 VALUE OF IT
21 PRODUCE THE INSERTED WORD VALUE
22 UPDATED ADDRESS
23 NEW CONTENTS OF IT
24 PRODUCE OUTPUT
25 CONTINUE

```

```

" R4 CONTAINS THE LINE NUMBER
" R5 SAVE RETURN ADDRESS
" CONVERT LINE NUMBER TO CHARACTERS
OUTBUF+8 - 2 (2 : 5) " STORE THEM INTO BUFFER
SPACE " INTERNAL CODE OF A SPACE
OUTBUF+8 - 2 (2 : 2) " FIRST DECIMAL
9F " READY IF NOT A ZERO
OUTBUF+8 - 2 (2 : 2) " REPLACE THE ZERO BY A SPACE
OUTBUF+8 - 2 (2 : 3) " SECOND DECIMAL
9F " READY IF NOT A ZERO
OUTBUF+8 - 2 (3 : 3) " REPLACE THE ZERO BY A SPACE
OUTBUF+8 - 2 (4 : 4) " THIRD DECIMAL
9F " READY IF NOT A ZERO
OUTBUF+8 - 2 (4 : 4) " REPLACE THE ZERO BY A SPACE
MOD " RETURN

```

```

" R12 CONTAINS A POINTER TO THE SYMBOL
" R13 SAVE RETURN ADDRESS
9F
TABLE - 1,2 (0 : 1)
9F " READY IF SYMBOL UNDEFINED YET
TABLE - 2,2 " VALUE OF SYMBOL
" PRINT
" THIS
OUTBUF+8 - 5
OUTBUF+8 - 4 " VALUE
1F " JUMP IF THE VALUE IS NEGATIVE

```

```

1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087

```

```

3/ 26 + 13 27 02 01 10
3/ 27 + 00 00 01 50
3/ 28 + 13 26 02 01 10
3/ 29 + 00 00 00 01 05
3/ 30 + 13 07 00 05 30
3/ 31 + 13 08 00 05 37
3/ 32 + 00 00 00 05 50
3/ 33

```

04

290173-126	D 2166Z.1 GERTENVELDEN	43			
3/ 34	+ 00 37 00 02 60	1088	ENT	PLUS	") 'PRINT' PLUS SIGN
3/ 35	+ 13 06 00 44 30	1089	STA	OUTBUF+8 - 6 (4 : 4)	
3/ 36	+ 00 38 00 02 60	1090	JMP	2F	
3/ 37	+ 00 37 00 02 60	1091	ENT	MINUS	") 'PRINT' MINUS SIGN
3/ 38	+ 13 06 00 44 30	1092	STA	OUTBUF+8 - 6 (4 : 4)	
3/ 39	+ 13 27 02 23 10	1093	LOA	TABLE - 1, 2 (2 : 3)	" LINE OF DEFINING OCCURRENCE
3/ 40	+ 00 34 50	1094	JMP	3F	" JUMP IF IT HAS BEEN OCCURRED
3/ 41	+ 00 37 00 02 60	1095	ENT	PLUS	") 'PRINT' A PLUS SIGN
3/ 42	+ 13 11 00 11 30	1096	STA	OUTBUF+8 - 1 (1 : 1)	") INSTEAD OF LINE NUMBER
3/ 43	+ 00 00 00 47	1097	JMP	4F	
3/ 44	+ 37 12 00 00 47	1098	JMP	BUFFER LN	" 'PRINT' THE LINE NUMBER
3/ 45	+ 37 44 00 04 50	1099	ENT	8	") POSITION IN BUFFER
3/ 46	+ 00 11 00 02 65	1100	ENT	1 : 1	") FOR SYMBOL
3/ 47	+ 36 24 00 00 47	1101	JMP	BUFFER SYMB	" 'PRINT' THE SYMBOL
3/ 48	+ 29 16 00 00 47	1102	JMP	PRINT LINE	" PRINT THE LINE
3/ 49	+ 00 00 00 00 47	1103	JMP	MOD	" RETURN
3/ 50	+ 37 49 00 01 50				
3/ 51	+ 37 49 00 02 40				
3/ 52	+ 28 28 00 05 11	1104	INITIALIZE:	" NO PARAMETERS	" SAVE RETURN ADDRESS
3/ 53	+ 13 28 01 02 61	1105	STJ	9F	") FIRST ADDRESS PRECEDING RELEVANT
3/ 54	+ 00 00 01 05 51	1106	LDI	TABLE PTR1	") INFORMATION FOR ASSEMBLER
3/ 55	+ 00 01 00 01 61	1107	ENT	TABLE, 1	" JUMP IF ADDRESS ZERO TREATED
3/ 56	+ 37 53 00 00 47	1108	JMP	0, 1	" INITIALIZE ADDRESS ON ZERO
3/ 57	+ 28 37 00 05 20	1109	STZ	1	" NEXT ADDRESS
3/ 58	+ 37 57 00 00 51	1110	DEC	1B	" CONTINUE
3/ 59	+ 12 21 00 05 30	1111	LDAN	END MARKER	
3/ 60	+ 28 37 00 05 10	1112	STA	DANGEROUS	" INITIALIZE DANGEROUS
3/ 61	+ 12 25 00 05 30	1113	LDI	END MARKER	
3/ 62	+ 12 38 00 05 30	1114	STA	LAST CONSTANT	" INITIALIZE LAST CONSTANT
3/ 63	+ 12 39 00 02 61	1115	STA	HERE - 10	") INITIALIZE
3/ 64	+ 12 38 00 09 07	1116	ENT	HERE - 9	") ARRAYS 'H' AND 'I'
3/ 65	+ 12 02 00 05 41	1117	MOVE	HERE - 10 (9)	
3/ 66	+ 13 04 00 02 61	1118	STZ	LINE NUMBER	" INITIALIZE LINE NUMBER
3/ 67	+ 28 41 00 01 07	1119	ENT	OUTBUF	
3/ 68	+ 13 04 00 23 07	1120	MOVE	SPACES (1)	") INITIALIZE
3/ 69	+ 00 01 00 02 62	1121	MOVE	OUTBUF (23)	") OUTPUT BUFFER
3/ 70	+ 00 55 00 02 63	1122	IN	INBUF X (READER)	" READ THE FIRST CARD
3/ 71	+ 00 66 00 02 60	1123	ENT	1	") INITIALIZE
3/ 72	+ 12 00 00 05 30	1124	ENT	5 : 5	") READING
3/ 73	+ 00 00 00 00 47	1125	ENT	SPACE	") POINTERS
3/ 74	+ 37 73 00 02 61	1126	STA	CHAR	")
3/ 75	+ 37 73 00 02 40	1127	JMP	MOD	" RETURN
3/ 76	+ 00 02 61	1128			
3/ 77	+ 37 73 00 02 31	1129			
3/ 78	+ 14 99 00 02 61	1130			
3/ 79	+ 37 78 00 02 61	1131			
3/ 80	+ 00 02 40	1132			
3/ 81	+ 00 02 61	1133			
3/ 82	+ 37 73 00 02 31	1134	INIT TABLE:	" NO PARAMETERS	" SAVE RETURN ADDRESS
3/ 83	+ 37 64 00 00 47	1135	STJ	9F	") RETURN ADDRESS AFTER INIT READING
3/ 84	+ 14 99 00 02 61	1136	ENT	1F	") INTO RETURN CF INIT READING
3/ 85	+ 37 78 00 02 61	1137	STI	9B (0 : 2)	" INITIALIZE READING POINTERS
3/ 86	+ 37 78 00 02 61	1138	JMP	INIT READING	" INITIALIZE READING POINTERS
3/ 87	+ 37 78 00 02 61	1139	ENT	LENGTH OF TABLE - 1	" POINTER TO ROOT OF TREE

" NEXT FOLLOWS THE PIECE OF PROGRAM, SERVING AS A VERY SIMPLE MONITORING
 " PROGRAM, ACCEPTING SEVERAL MIXAL PROGRAMS, BEING SEPARATED FROM EACH OTHER
 " BY A JOB SEPARATOR (" " IN THE COLUMNS 79 AND 80).
 " THESE MIXAL PROGRAMS ARE TRANSLATED AND EXECUTED (IF CORRECT) CONSECUTIVELY.
 " THE JOB SEPARATOR IS DETECTED BY THE ROUTINE "READ CHAR" ONLY; SO DURING
 " EXECUTION OF A PROGRAM, THE JOB SEPARATOR WILL BE IGNORED.
 " AS THE ASSEMBLER ALWAYS READS ONE CARD IN ADVANCE, THE CARD SUCCEEDING
 " THE END INSTRUCTION IS READ TOO BY THE ROUTINE "MIXAL ASSEMBLER".
 " THIS FACT HAS TO BE TAKEN INTO ACCOUNT WHEN IN - INSTRUCTIONS ARE PERFORMED
 " BY A PARTICULAR MIXAL PROGRAM.
 " THE MONITOR MAKES USE OF THE AUXILIARY ROUTINES "SKIP INPUT" AND
 " "PUNCH TRAJECT".

START0: LD2 TABLE PTR1
 ENT2 TABLE + 1 ,2 " START ADDRESS OF TRAJECT
 ENT3 0 ,2 ") LENGTH OF
 DEC3 ") TRAJECT (NEGATIVE)
 JMP PUNCH TRAJECT " PUNCH PERMANENT OF ASSEMBLER
 JMP INIT TABLE " INITIALIZE THE SYMBOL TABLE
 JMP MIXAL ASSEMBLER " TRANSLATE THE PROGRAM
 NEXT PROGRAM:
 JOC 0 (PRINTER) " NEW PAGE
 JBUS * (READER) " SKIP THE FIRST INPUT CARD
 JAMP 0 ,1 " EXECUTE THE PROGRAM IF CORRECT
 JMP SKIP INPUT " SKIP REST OF INPUT OF PROGRAM
 JMP NEXT PROGRAM " TREAT THE NEXT PROGRAM

SKIP INPUT: " NO PARAMETERS
 STJ 9F " SAVE RETURN ADDRESS
 1H: IN INBUF (READER) " READ NEXT CARD
 JBUS * (READER) " WAIT FOR CARD READER
 LDA INBUF + 15 " POSSIBLE JOB SEPARATOR
 CMPA JOB SEPARATOR (4 : 5) " IS IT A JOB SEPARATOR ?
 JNE 1B " ELSE CONTINUE
 JMP MOD " RETURN
 9H: MOD

PUNCH TRAJECT: " R12 CONTAINS THE START ADDRESS,
 " R13 CONTAINS THE LENGTH OF THE TRAJECT (NEGATIVE)
 STJ 9F " SAVE RETURN ADDRESS
 ENT1 OUTBUF - 1 ")
 MOVE SPACES (1) ") INITIALIZE
 MOVE OUTBUF - 1 (16) ") OUTPUT BUFFER
 ENT5 PLUS " CODE OF PLUS SIGN
 ENT6 MINUS " CODE OF MINUS SIGN
 ENT7 0 ,2 " START ADDRESS
 CHAR " CONVERT IT TO CHARACTERS
 STX OUTBUF " STORE THEM INTO THE OUTPUT BUFFER
 ENNA 0 ,3 " LENGTH OF TRAJECT (POSITIVE)
 CHAR " CONVERT IT TO CHARACTERS
 STX OUTBUF + 2 " STORE THEM INTO THE OUTPUT BUFFER
 DEC2 0 ,3 " START ADDRESS + LENGTH OF TRAJECT
 ST2 LOAD (0 : 2) " MODIFY INSTRUCTION
 OUT: OUTBUF (CPUNCH) " PUNCH A CARD
 JBUS * (CPUNCH) " WAIT FOR CARD PUNCH

36 72 + 28 28 00 05 12
 36 76 + 13 29 02 02 62
 36 77 + 00 00 02 02 63
 36 78 00 01 63
 36 79 00 00 47
 36 80 37 74 00 00 47
 36 81 + 38 49 00 00 47
 36 82 + 38 81 00 00 47
 36 83 + 00 00 00 18 43
 36 84 + 38 83 00 16 42
 36 85 + 00 00 01 03 50
 36 86 00 00 47
 36 87 + 38 81 00 00 47
 36 88 00 02 40
 36 89 + 38 87 00 00 47
 36 90 + 12 64 00 16 44
 36 91 + 38 89 00 16 42
 36 92 + 12 79 00 05 10
 36 93 + 28 42 00 45 70
 36 94 + 38 88 00 08 47
 36 95 + 00 00 00 00 47
 36 96 + 38 93 00 02 40

36 74 00 02 40
 36 75 + 38 94 00 00 47
 36 76 + 13 03 00 02 61
 36 77 + 28 41 00 01 07
 36 78 + 13 03 00 16 07
 36 79 + 00 37 00 02 65
 36 80 + 00 38 00 02 66
 36 81 + 00 00 02 02 60
 36 82 + 00 00 00 01 05
 36 83 + 13 04 00 05 37
 36 84 + 00 00 03 03 60
 36 85 + 00 00 00 01 05
 36 86 + 13 06 00 05 37
 36 87 + 00 00 03 01 62
 36 88 + 00 00 02 02 32
 36 89 + 13 04 00 17 45
 36 90 + 39 09 00 17 42

04

290173-126		D 2166Z.1 GERTENVELDEN		47		04	
39 10	+	00 00 00 01 53	1301	9H:	J3Z	MOD	" RETURN IF TRAJECT COMPLETED
*38 94	+	39 10 00 02 40	1302		ENT1	OUTBUF	") INITIALIZE
39 11	+	13 04 00 02 61	1303		MOVE	OUTBUF - 1 (16)	") OUTPUT BUFFER AGAIN
39 12	+	13 03 00 06 07	1304		ENN4	15	" INITIALIZE POSITION ON CARD
39 13	+	00 15 00 03 64	1305	LOAD:	LDA	MOD ,3	" NEXT WORD
39 14	+	00 00 03 05 10					
*39 07	+	39 14 00 02 32					
39 15		00 00 50	1306		JAN	1F	" JUMP IF NEGATIVE
39 16	+	13 19 04 33 35	1307		ST5	OUTBUF + 15 ,4 (3 : 3)	" STORE PLUS SIGN INTO BUFFER
39 17		00 00 47	1308		JMP	2F	
39 18	+	13 19 04 33 36	1309	1H:	ST6	OUTBUF + 15 ,4 (3 : 3)	" STORE MINUS SIGN INTO BUFFER
*39 12	+	39 18 00 00 50					
39 19	+	00 00 00 01 05	1310	2H:	CHAP		" CONVERT WORD TO CHARACTERS
*39 17	+	39 19 00 00 47					
39 20	+	13 20 04 05 30	1311		STA	OUTBUF + 16 ,4	") STORE THEM
39 21	+	13 21 04 05 37	1312		STX	OUTBUF + 17 ,4	") INTO OUTPUT BUFFER
39 22	+	00 03 00 00 64	1313		INC4	3	" NEXT POSITION ON CARD
39 23	+	00 01 00 00 63	1314		INC3	1	" NEXT WORD FROM MEMORY
39 24	+	39 08 00 01 54	1315		J4Z	OUT	" JUMP IF CARD COMPLETED
39 25	+	39 14 00 04 53	1316		J3NZ	LOAD	" JUMP IF TRAJECT NOT COMPLETED
39 26	+	39 08 00 00 47	1317		JMP	OUT	" ELSE PUNCH LAST CARD
			1318				
			1319	END OF ASS:	END	START0	
			1320	AUX:	CON	0	
39 27	+	00 00 00 00 00					
*32 34	+	39 27 00 05 70					
*32 33	+	39 27 00 05 37					
*30 25	+	39 27 00 05 03					
*30 26	+	39 27 00 05 30					
39 28	+	00 00 00 00 11					
*30 19	+	39 28 00 05 04					
*35 78	+	39 29 00 01 63					
*35 78	+	39 29 00 00 67					
*35 76	+	39 29 00 01 67					

START ADDRESS: +J875

TRANSLATION TIME: 330.54 SEC.

+	00	00	00	00	38	71	1238	EXIT
+	00	00	00	00	12	11	67	EXPR
+	00	00	00	00	00	64	360	F
+	00	00	00	00	01	09	406	FOUND
+	00	00	00	00	12	17	73	F PART
+	00	00	00	00	34	34	754	FUTREF
+	00	00	00	00	34	25	745	FUTSYMB
+	00	00	00	00	12	20	76	FUTURE
+	00	00	00	00	12	48	90	HERE
+	00	00	00	00	28	40	115	HUNDREDTHOUSAND
+	00	00	00	00	12	64	91	INSUF
+	00	00	00	00	12	48	90	INBUF
+	00	00	00	00	07	50	1107	INITIALIZE
+	00	00	00	00	37	64	1121	INITREADING
+	00	00	00	00	37	74	1134	INITTABLE
+	00	00	00	00	36	86	1034	INSERTCONSTANTS
+	00	00	00	00	12	24	80	INST
+	00	00	00	00	12	18	74	INSTRUCTION
+	00	00	00	00	06	08	938	INVESTIGATE
+	00	00	00	00	12	16	72	IPART
+	00	00	00	00	28	42	117	JOBSEPARATOR
+	00	00	00	00	12	25	81	LASTCONSTANT
+	00	00	00	00	28	29	191	LASTINST
+	00	00	00	00	12	22	78	LASTLABEL
+	00	00	00	00	12	06	62	LASTSYMSOL
+	00	00	00	00	12	19	75	LC
+	00	00	00	00	00	86	383	LEFT
+	00	00	00	00	28	39	114	LENGTHOFMEMORY
+	00	00	00	00	15	00	48	LENGTHOFTABLE
+	00	00	00	00	29	50	245	LETGIT
+	00	00	00	00	00	11	12	LETTERB
+	00	00	00	00	00	15	13	LETTERF
+	00	00	00	00	00	17	14	LETTERH
+	00	00	00	00	00	35	15	LETTERZ
+	00	00	00	00	12	02	58	LINENUMBER
+	00	00	00	00	34	08	728	LITCONS
+	00	00	00	00	12	26	82	LITCONSPTR
+	00	00	00	00	39	14	1305	LOAD
+	00	00	00	00	34	29	749	LOC
+	00	00	00	00	00	03	42	LOCALBACKWARD
+	00	00	00	00	00	04	43	LOCALFORWARD
+	00	00	00	00	00	02	41	LOCALLABEL
+	00	00	00	00	31	74	474	LOCDEFINED
+	00	00	00	00	03	05	615	LOCCLAB
+	00	00	00	00	30	70	367	LOOKFORSYMBOL
+	00	00	00	00	35	65	892	LOOP1
+	00	00	00	00	35	85	912	LOOP2
+	00	00	00	00	28	37	111	MAXADDRESS
+	00	00	00	00	28	36	109	MAXBYTE
+	00	00	00	00	28	38	112	MAXWORD
+	00	00	00	00	00	38	17	MINUS
+	00	00	00	00	08	49	1216	MIXALASSEMBLER
+	00	00	00	00	00	00	46	MOD
+	00	00	00	00	32	21	525	MUL
-	00	00	00	00	00	01	47	N
+	00	00	00	00	01	01	398	NEW
+	00	00	00	00	30	71	388	NEXT
+	00	00	00	00	32	81	591	NEXTINST
+	00	00	00	00	31	93	496	NEXTOPERAND
+	00	00	00	00	38	81	1265	NEXTPROGRAM

+	00	00	00	00	32	42	549	NEXTSPEC
+	00	00	00	00	32	83	593	NEXTSYMBOL
+	00	00	00	00	00	92	30	NLCR
+	00	00	00	00	31	80	480	NUMS
+	00	00	00	00	01	02	36	NUMBER
+	00	00	00	00	37	80	1140	OPERS
+	00	00	00	00	28	34	106	ORIG
+	00	00	00	00	31	44	441	OTHER
+	00	00	00	00	39	08	1299	OUT
+	00	00	00	00	13	04	93	OUTBUF
+	00	00	00	00	12	80	92	OUTBUF
+	00	00	00	00	30	43	338	OVERFLOW
+	00	00	00	00	00	71	28	PAREN
+	00	00	00	00	34	58	778	PARENCHECKED
+	00	00	00	00	00	37	16	PLUS
+	00	00	00	00	00	61	23	POINT
+	00	00	00	00	33	70	685	POSSIBLE
+	00	00	00	00	00	18	8	PRINTER
+	00	00	00	00	29	16	204	PRINTLINE
+	00	00	00	00	34	98	822	PRODUCE
+	00	00	00	00	37	97	1157	PSEUDOINSTS
+	00	00	00	00	38	94	1285	PUNCHTRAJECT
+	00	00	00	00	00	94	31	QUOTE
+	00	00	00	00	29	63	258	READ
+	00	00	00	00	28	94	179	READ5CHAR
+	00	00	00	00	31	49	449	READATEXPRESS
+	00	00	00	00	28	76	158	READC
+	00	00	00	00	28	45	127	READCHAR
+	00	00	00	00	00	16	6	READER
+	00	00	00	00	31	85	488	READEXPRESS
+	00	00	00	00	33	87	707	READMIXMACHINEINST
+	00	00	00	00	32	80	590	READPROGRAM
+	00	00	00	00	29	44	239	READSYNTUNIT
+	00	00	00	00	32	39	546	READWVALUE
+	00	00	00	00	38	16	1179	RESETTABLE
+	00	00	00	00	30	95	392	RIGHT
+	00	00	00	00	01	00	108	S
+	00	00	00	00	00	64	26	SEMICOLON
+	00	00	00	00	01	01	35	SEP
+	00	00	00	00	31	19	410	SEPARATOR
+	00	00	00	00	38	87	1274	SKIPINPUT
+	00	00	00	00	00	66	27	SPACE
+	00	00	00	00	28	41	116	SPACES
+	00	00	00	00	32	71	578	STANDARD
+	00	00	00	00	38	75	1257	START0
+	00	00	00	00	38	80	1264	START1
+	00	00	00	00	12	27	83	STARTADDRESS
+	00	00	00	00	32	17	521	SUB
+	00	00	00	00	31	64	464	SYMB
+	00	00	00	00	01	03	37	SYMBOL
+	00	00	00	00	12	05	61	SYNTUNIT
+	00	00	00	00	13	28	54	TABLE
+	00	00	00	00	28	28	100	TABLEPTR1
+	00	00	00	00	12	04	60	TEXT
+	00	00	00	00	00	72	29	THESIS
+	00	00	00	00	00	39	19	TIMES
+	00	00	00	00	32	10	514	TIMESBASEPLUS
+	00	00	00	00	36	01	931	TRAVERSE TREE
+	00	00	00	00	33	14	525	TREATINSTRUCTION
+	00	00	00	00	12	07	63	TYPE

2901/3-126 D 21662.1 GERTENVELDEN 51 04

+ 00 00 00 00 00 39 UNDEFINESYMB
+ 00 00 00 33 01 611 UNDEFSYM
+ 00 00 00 35 64 891 UPDATE
+ 00 00 00 12 12 68 VALUE
+ 00 00 00 00 02 44 VALUE1
+ 00 00 00 00 45 45 VALUE2
+ 00 00 00 12 08 64 VALUEOFNUMBER
+ 00 00 00 38 31 1194 VISIT
+ 00 00 00 32 68 575 WRONGFIELD

2901/3-126 D 21662.1 GERTENVELDEN 52 04

1	JNP	553;	ENT5	265;	LD4	514;	DEC6	166;	INC6	066
2	JL	447;	SLAX	206;	CMP6	576;	ENNS	365;	FSUB	602
3	J1P	251;	J5N	056;	JXP	257;	LDX	517;	ST4	534
4	CMP2	572;	DEC2	162;	ENN1	361;	ENT1	261;	FADD	601
5	INC2	062;	J1N	051;	J2NP	552;	J4P	254;	JANP	550
6										
7	JXN	057;	LD2	512;	LD6	516;	NOP	000;	SRC	506
8	STJ	240;	CHAR	105;	CMP4	574;	CMPX	577;	DEC4	164
9	DECX	167;	ENN3	363;	ENNA	360;	ENT3	263;	ENTA	260
10	FDIV	604;	IN	042;	INC4	064;	INCX	067;	J1NP	551
11	J2N	052;	J2P	252;	J4N	054;	J5NP	555;	J5P	256
12										
13	JBUS	042;	JNOV	347;	JXNP	557;	LD1	511;	LD3	513
14	LD5	515;	LD4	510;	MOVE	107;	OUT	045;	SRA	106
15	ST2	532;	ST6	536;	STZ	541;	ADD	501;	CMP1	571
16	CMP3	573;	CMP5	575;	CMPA	570;	DEC1	161;	DEC3	163
17	DEC5	165;	DECA	160;	DIV	504;	ENN2	362;	ENN4	364
18										
19	ENN6	366;	ENNX	367;	ENT2	262;	ENT4	264;	ENT6	266
20	ENTX	267;	FCMP	670;	FMUL	603;	HLT	205;	INC1	061
21	INC3	063;	INC5	065;	INCA	060;	IOC	043;	J1NN	351
22	J1NZ	451;	J12	151;	J2NN	352;	J2NZ	452;	J3N	053
23	J3P	253;	J4NP	554;	J5N	053;	J5P	255;	J6NP	556
24										
25	JAN	050;	JAP	250;	J6	647;	JMP	047;	JRED	046
26	JXNN	357;	JXNZ	457;	JXZ	157;	LD1N	521;	LD2N	522
27	LD3N	523;	LD4N	524;	LD5N	525;	LD6N	526;	LDAN	520
28	LDXN	527;	MUL	503;	NUM	005;	SLA	006;	SLC	406
29	SRA	306;	ST1	531;	ST3	533;	ST5	535;	STA	530
30										
31	STX	537;	SUB	502;	J2Z	152;	J3NN	353;	J3NZ	453
32	J3Z	153;	J4NN	354;	J4NZ	454;	J4Z	154;	J5NN	355
33	J5NZ	455;	J5Z	155;	J6NN	356;	J6NZ	456;	J6Z	156
34	JANN	350;	JANZ	450;	JAZ	150;	JE	547;	JGE	747
35	JLE	947;	JNE	847;	JOV	247;	JSJ	147		
36	(ALF		CON		END		EQU		ORIG)	

8.1 Table of error messages.

error number:	description:
ER 01	instruction definitions not o.k.
ER 03	pseudo instruction definitions not o.k.
ER 10	. required.
ER 11	value of number too large.
ER 21	atomic expression required.
ER 22	undefined local backward in expression.
ER 23	undefined symbol in expression.
ER 31	overflow by arithmetic operation.
ER 41	illegal field.
ER 42	) required.
ER 51	symbol required.
ER 52	illegal label.
ER 53	(local) label used twice.
ER 54	unknown instruction.
ER 55	separator required.
ER 61	= required.
ER 62	) required.
ER 63	value of a part not o.k.
ER 64	value of i part not o.k.
ER 65	value of f part not o.k.
ER 71	wrong address.
ER 72	address already used.
ER 81	value for future reference too large.
ER 91	wrong start address.
ER 92	program incorrect.
ER 98	end of input.
ER 99	overflow symbol table.

9. A loading program.

Before discussing the loading program, we have to treat a feature of the MIX simulator, concerning the start of the computer. With respect to the MIXAL system, consisting of simulator and assembler (see introduction), there is no question of a start of the simulator from the point that no relevant information is in the memory; the proper MIX simulator, without assembler, has the problem of how to load the first instructions into memory. For this purpose, the proper simulator has been provided with a simulated "start button", being "pressed" at the beginning of the execution of the MIX simulator. Pressing this start button, the MIX instruction "IN 0(16)" is executed, which causes the activation of the card reader; after one card has been read, and the information of it has been stored into the locations zero upwards, the normal cycle of the computer is started at location zero.

Preparing this first card by hand, we thus can get the first simple loading program, consisting of at most 16 instructions, into the machine. Each position on this card corresponds to one byte; this byte is filled with the internal code of the character on that position (this is the usual effect of an IN instruction).

Unfortunately, this function on the collection of characters into the numbers $\{n \mid 0 \leq n < 100\}$ is not a surjective one; e.g. the instruction codes of the IN and JBUS instructions (44 and 42, resp.) do not correspond to any MIX character. As these two instructions are indispensable in a loading program, we have to find another way to get these instructions into the memory. An easy solution is: constructing these instructions programmatically (e.g. by means of the instructions: "ENT2 40; INC2 4; ST2 X(5:5)", when an IN instruction has to be loaded at location X). Till so far, we have discussed the start of this special MIX simulator, and noticed some difficulties and possibilities of it.

The loading program of this section consists of three cards, the proper loader being contained in the second and third card. The first card serves to construct the instructions for reading the next two cards, and, after these cards have been read, to construct the IN and JBUS instructions of the proper loader. Next, control is passed on to the proper loader, and the

locations 0 - 15 are used as an input buffer by the loader. Much attention had to be paid to arranging the instructions of the proper loader in such a way, that the value of the bytes of all instructions did correspond to an existing MIX character.

Eventually, the code of the information to be loaded is given. The input of the loader consists of a number of blocks of information, each block consisting of a number of information cards, preceded by one traject card. On the traject card, two numbers have been punched: a startaddress (columns 1 - 5), and a quantity indicating the number of consecutive locations (beginning at the startaddress) to be loaded (columns 11 - 15). Each information card contains the information to be loaded into (the next) five consecutive locations. Column 3 contains the sign (+ or -) of the first location, columns 6 - 15 contain the numerical value for this location in decimal notation (two columns for each byte); columns 18 and 21 - 30 contain the information for the second location, etc..

The last of the information cards contains as much information as is necessary to complete this particular block.

A traject card, which contains a zero as second number, indicates the final block. In this case, the startaddress on the traject card is interpreted as the location at which the loaded program has to be started.

This kind of code is produced by the routine called "punch traject" of the MIXAL version of the assembler. This routine punches one block of information, including the traject card. The startaddress is given by the value of index register 2, and the number of consecutive locations to be punched out, is given by the negative value of index register 3.

Next follows the MIXAL text of the entire loading program (some instructions have been added, from location 100 upwards, to punch the three initial cards, mentioned before, which contain the loading system), in the way it has been printed by the MIXAL version of the MIXAL assembler.

04

290173-126	0 2166Z.1 GERTENVELDEN	53	
0000	+ 1640000261	1	INBUF:
0001	+ 0002010262	2	ENT1
0002	+ 0004000061	3	ENT2
0003	004531	4	INC:
0004	004531	5	ST1
0005	004531	6	ST1
0006	004532	7	ST2
0007	+ 0016000000	8	NOP
0008	+ 0006000000	9	NOP
0009	+ 0032000000	10	NOP
0010	+ 0007004531	11	ST1
0011	+ 0008000000	12	ST2
0012	+ 0008004532	13	JMP
0013	000047	14	READINFCARDS (4:5)
0014		15	WAITINFCARDS (4:5)
0015	+ 0000000000	16	READTRAJECT (4:5)
0016	+ 0000000038	17	WAITINFCARDS (4:5)
0017	+ 0003000063	18	READTRAJECT (4:5)
0018	000453	19	WAITTRAJECT (4:5)
0019	+ 0015000063	20	READTRAJECT
0020	+ 0000000000	21	
0021	+ 0020004531	22	
0022	+ 0021000000	23	
0023	+ 0021004532	24	
0024	+ 0017003311	25	
0025	+ 0022000453	26	
0026	+ 0016030510	27	
0027	+ 00170030517	28	
0028	+ 0000000005	29	
0029	+ 0016000571	30	
0030	000647	31	
0031	+ 0015000530	32	
0032	+ 0015000520	33	
0033	+ 0000020530	34	
0034	+ 0030000847	35	
0035	+ 0001000062	36	
0036	+ 0017000452	37	
0037	+ 0000000000	38	
0038	+ 0033000047	39	
0039	+ 0033004531	40	
0040	+ 0034000000	41	
0041	+ 0034004532	42	
0042	+ 0000000260	43	
0043	+ 0002000517	44	
0044	+ 0000000005	45	
0045	+ 0045000230	46	
0046	+ 0000020560	47	
0047	+ 0000000152	48	
0048	+ 0030000030	49	
0049	+ 0019000047	50	

2901/3-126	0 2166Z.1 GERTENVELDEN	54	ORIG	100
0100		51	STJ	9F
0101	000240	52	OUT	0 (17)
0102	+ 0000001745	53	OUT	16 (17)
0103	+ 0016001745	54	OUT	32 (17)
0104	+ 0032001745	55	JBUS	* (17)
0105	+ 0104001742	56	JMP	*
0106	+ 0102000147	57		
0107	+ 0105000240	58	END	100

2901/3-126	0 2166Z.1 GERTENVELDEN	55	AUX	17
	+ 0000000015	1	INBUF	1
	+ 0000000000	7	INCARD1	7
	+ 0000000006	8	INCARD2	8
	+ 0000000007	9	JBUS	9
	+ 0000000008	10	MINUS	10
	+ 0000000016	22	READINFCARDS	22
	+ 0000000020	35	READTRAJECT	35
	+ 0000000033	32	STORE	32
	+ 0000000030	23	WAITINFCARDS	23
	+ 0000000021	36	WAITTRAJECT	36
	+ 0000000034			

References.

- [1]. Knuth, D.E., The Art of Computer Programming, Vol. 1
Fundamental Algorithms.
Reading (Mass.), Addison-Wesley, 1968.

- [2]. van de Riet, R.P.,
Inleiding in de Informatica,
Syllabus bij het Oriënterend Colloquium Informatica,
CR 21/70 december, Mathematisch Centrum.

- [3]. Grune, D., Handleiding Milli-systeem voor de EL X8,
LR. 1.1, Mathematisch Centrum.