AN ABSTRACT OF THE THESIS OF

SUSAN PAULA MARGOLIS for the MASTER OF ARTS

in Mathematics presented on December 17, 1969

Title: A TURING MACHINE PROGRAMMING LANGUAGE

Abstract approved: Professor Harry Goheen

This thesis documents a new language which facilitates the construction of Turing machines. The language translator is written in Compass and has been debugged and is available for use on the CDC 3300.
A Turing Machine Programming Language

by

Susan Paula Margolis

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Arts

June 1970
ACKNOWLEDGEMENTS

I would like to thank Professor H.E. Goheen for his help and encouragement in the development of this thesis.

I would like to thank Charlene Laski for typing the thesis.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II DESCRIPTION OF THE LANGUAGE</td>
<td>3</td>
</tr>
<tr>
<td>III IMPLEMENTATION OF THE LANGUAGE</td>
<td>12</td>
</tr>
<tr>
<td>IV UNIVERSALITY OF THE TURING MACHINE PROGRAMMING LANGUAGE</td>
<td>15</td>
</tr>
<tr>
<td>EPILOGUE</td>
<td>21</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>23</td>
</tr>
<tr>
<td>APPENDIXES</td>
<td>24</td>
</tr>
<tr>
<td>Test case</td>
<td>27</td>
</tr>
<tr>
<td>Flow charts</td>
<td></td>
</tr>
<tr>
<td>Program listing</td>
<td>32</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>A Turing machine which transforms an arithmetic expression into Lukasiewicz Normal Form</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>A listing in Turing machine programming language for transforming an arithmetic expression into Lukasiewicz Normal Form</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>Diagram of source deck structure</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>List of diagnostics</td>
<td>14</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

This thesis develops a symbolic language to facilitate the construction of Turing machines. By now, the importance of Turing machines is well-known. They are recognized as an excellent tool in constructive proof theory, as well as an important source of examples in automata theory. For an introduction to Turing machine theory, see Davis (4), Hermes (5). For important examples of the power of Turing machines see Kleene (6) and Minsky (9).

Existing techniques for constructing Turing machines are inadequate. Construction of symbolic languages for processing automata is an important but neglected area of automatic programming. As Knuth has noted "... most current papers on automata theory have not advanced past the 'machine language' programming stage; authors have been reluctant to simplify their constructions except by using classical mathematical notations which are cumbersome for this purpose.

Symbolic programs are considerably easier to write and to read, and they are less likely to contain careless errors. So it is surprising that these ideas have not
already spread from computer programming to automata pro-
gramming."

The only previous Turing machine simulator language in
the literature belongs to M.W. Curtis (3). Curtis's lan-
guage, written for the IBM 1620, seems to have all the
failings noted by Knuth above. It is a language at the
machine level and employs mnemonics quite divorced from the
structure of Turing machine operations.

The language to be described is a higher level sym-

tonic language, which has been implemented on the CDC 3300.
A program processed in this language yields as output a
Turing machine in a form usable by the TASP\(^2\) compiler (if
desired) as well as a printout of the resulting quintuples.

To test the processor, Watanabe's Universal Turing ma-
chine (12) is encoded in this language, and the resulting
program may be compared with the encoding of a universal
Turing machine by Curtis, who also used Watanabe's version.

The possibility of encoding a Universal Turing machine
gives a proof of the universality of the language in encod-
ing algorithms. This fact is also proved directly in
Chapter 3.

Automata. Journal of the Association of Computing

2. TASP is a Turing Automation Simulation Program.
II. DESCRIPTION

This section consists of an informal description of the programming language, followed by rigorous definition of allowable syntactical structures using the notation of Backus-Naur (2).

In order to describe the language, we shall study it in connection with an example. Consider the problem of devising an algorithm to transform arithmetic expressions into Lukasiewicz Normal Form (8). A Turing machine which will perform such an algorithm is given below:

\[
\begin{array}{cccccccccccc}
H & D & S & ( & X & Y & Z & ) & + & - & * & / \\
1 & 2RH & 1RS & 2RH & 1RX & 1RY & 1RZ & 1RH & 1R+ & 1R- & 1R* & 1R/ \\
2 & 3L) & 2RD & ! & 2R( & 11LX & 11LY & 11LZ & 3L) & 2R+ & 2R- & 2R* & 2R/ \\
3 & 1RS & 3LD & 9RD & & & & & 4RD & 5RD & 6RD & 7RD \\
4 & 4RH & 4RD & 8L+ & 4R( & 4RX & 4RY & 4RZ & 4R) & 4R+ & 4R- & 4R* & 4R/ \\
5 & 5RH & 5RD & 8L- & 5R( & 5RX & 5RY & 5RZ & 5R) & 5R+ & 5R- & 5R* & 5R/ \\
7 & 7RH & 7RD & 8L/ & 7R( & 7RX & 7RY & 7RZ & 7R) & 7R+ & 7R- & 7R* & 7R/ \\
8 & 8LH & 2RD & 8R( & 8RX & 8RY & 3RZ & 8L) & 8R+ & 8L- & 8L* & 8L/ \\
9 & 9RD & & & 13RD & 14RD & 15RD & 10RD & 9R+ & 9R- & & \\
10 & 3L) & 2R* & & 10L( & 10LX & 10LY & 10LZ & 3L) & 2R+ & 2R- & 2R* & 2R/ \\
11 & 9RH & 11LD & 9R( & & & & & 12L+ & 12L- & 9R* & 9R/ \\
12 & 9RH & 12LD & 9R( & & & & & 4RD & 5RD & 6RD & 7RD \\
14 & 14RH & 16LY & 14R( & 14RX & 14RY & 14RZ & 14R) & 14R+ & 14R- & 14R* & 14R/ \\
16 & 16LH & 10RD & 16L( & 16RX & 16LY & 16LZ & 16L) & 16L+ & 16L- & 16L* & 16L/ \\
\end{array}
\]

Figure 1
The above assumes the arithmetic expression is originally given in the standard form with delimiter 'h' at each end. We also follow the convention that the Turing machine is to commence its operation in State 1, looking at the left delimiter "h", and that the expression involves only the three variables x, y, and z; the four operations, viz. multiplication, "*", addition, "+", subtraction, "-", and division, "/", and left parenthesis, "(" , and right parenthesis, ")".

One may describe the Turing machine of Figure 1 in the following terms. For purposes of description consider the program divided into a number of blocks; first the heading block followed by a number of state blocks and finally the terminator block. In the heading block, the first card is the title card which is read by the processor and then printed at the top of each page of output with the time and date of the run. The next card is the Turing machine card. This card is necessary since more than one Turing machine may be processed in the same run.

The third card of the deck is the alphabet card. This card designates to the processor the set of characters which the Turing machine will recognize as its alphabet. The choice of characters for the Turing machine alphabet is limited to 47 of the 48 Standard Keypunch Characters ABC ... XYZ 012...9 +-/* , = ( ) $ the 48th character (space or blank) being reserved as a separator.
Next follows the state blocks. Each of these blocks is equivalent to a Turing machine state as described by its set of quintuples in that state.

The last card of the deck is the termination card END. This card denotes the end of compilation. At this point, if no errors have been detected, the quintuples are printed and the TASP processor is called. If any error is detected an appropriate message is printed along with the line number of the place of error detection and compilation is terminated.

A useful feature of this language is that state blocks correspond in a one-to-one fashion with states in the compiled Turing machine. The faithfulness of the number of states demonstrates the usefulness of this language in researches involving minimal state machines.

In terms of the above discussion the Turing machine of Figure 1 is presented in Figure 2.

Figure 2.

A TRANSFORMER OF ARITHMETIC EXPRESSION TO POLISH STRINGS

\[
\begin{align*}
\text{A LIST IS FORMED TO THE RIGHT OF THE INPUT STRING WHICH WHEN DONE CONTAINS THE POLISH STRING} \\
\text{HEAD BLOCK} \\
\text{TURING MACHINE 1} \\
\text{ALPHABET H D S ( X Y Z ) + - * /}
\end{align*}
\]

(continued)
STATE 1  BEGIN AND CLEAN UP AT THE END
  SEARCH RIGHT FOR H AND D AND S AND )
      H GO RIGHT TO STATE 2
      D CHANGE TO S
      S CHANGE TO H AND GO RIGHT TO STATE 2
      ) CHANGE TO H
STATE 2  LOOK FOR THE FIRST X Y Z OR H
  SEARCH RIGHT FOR H AND S AND X AND Y AND Z AND )
      H CHANGE TO ) AND GO LEFT TO STATE 3
      S GO TO STOP
      X GO LEFT TO 11
      Y GO LEFT TO 11
      Z GO LEFT TO 11
      ) GO LEFT TO 3
STATE 3  LOOK FOR FIRST OPERATOR
  SEARCH RIGHT FOR H ( + - * /
      H CHANGE TO S AND GO R TO 1
      ( CHANGE TO D AND GO R TO 9
      + CHANGE TO D AND GO R TO 4
      - CHANGE TO D AND GO R TO 5
      * CHANGE TO D AND GO R TO 6
      / CHANGE TO D AND GO R TO 7
STATE 4  PUT THE + ON THE OUTPUT SIDE
  SEARCH RIGHT FOR S
      S CHANGE TO + AND GO LEFT TO 8
(continued)
STATE 5  PUT THE - ON THE OUTPUT SIDE
SEARCH RIGHT FOR S
    S CHANGE TO - AND GO LEFT TO 8
STATE 6  PUT THE * ON THE OUTPUT SIDE
SEARCH RIGHT FOR S
    S CHANGE TO * AND GO LEFT TO 8
STATE 7  PUT THE / ON THE OUTPUT SIDE
SEARCH RIGHT FOR S
    S CHANGE TO / AND GO LEFT TO 8
STATE 8  GO BACK TO INPUT STRING
SEARCH LEFT FOR D
    D GO RIGHT TO 2
STATE 9  LOOK FOR THE VARIABLES IN INPUT AND REMOVE THEM
SEARCH RIGHT FOR X Y Z )
    X CHANGE TO D AND GO RIGHT TO 13
    Y CHANGE TO D AND GO RIGHT TO 14
    Z CHANGE TO D AND GO RIGHT TO 15
    ) CHANGE TO D AND GO RIGHT TO 10
STATE 10  * AND / WRITTEN AFTER PRECEDENCE RULES EXECUTED
SEARCH LEFT FOR H ( + - * /
    H CHANGE TO ) AND GO LEFT TO 3
    D CHANGE TO * AND GO RIGHT TO 2
    + GO R TO 2
    - GO R TO 2

(continued)
STATE 11  FIND FIRST OPERATOR AFTER A VARIABLE
SEARCH LEFT FOR H ( + - * /
  H GO R TO 9
  ( GO R TO 9
  + GO L TO 12
  - GO L TO 12
  * GO R TO 9
  / GO R TO 9

STATE 12  PRECEDENCE CHECK ON OPERATORS + AND - WRITTEN
SEARCH LEFT FOR H ( + - * /
  H GO R TO 9
  ( GO R TO 9
  + CHANGE TO D AND GO R TO 4
  - CHANGE TO D AND GO R TO 5
  * CHANGE TO D AND GO R TO 6
  / CHANGE TO D AND GO R TO 7

STATE 13  PUT THE X ON THE OUTPUT SIDE
SEARCH R FOR S
  S CHANGE TO X AND GO LEFT TO 16

STATE 14  PUT THE Y ON THE OUTPUT SIDE
SEARCH R FOR S
  S CHANGE TO Y AND GO LEFT TO 16

(continued)
STATE 15  PUT THE Z ON OUTPUT SIDE
SEARCH R FOR S

S CHANGE TO Z AND GO LEFT TO 16
STATE 16  GO BACK TO INPUT STRING
SEARCH L FOR D

D GO R TO 10
END

Syntactical Construction

1. Basic Symbols
\(<\text{external alphabet character}> ::= \langle\text{letter}\rangle|\langle\text{digit}\rangle|\langle\text{special character}\rangle\)

1.1 \(\langle\text{letter}\rangle ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z\)

1.2 \(\langle\text{digit}\rangle ::= 0|1|2|3|4|5|6|7|8|9\)

1.3 \(\langle\text{special character}\rangle ::= +|-|/|\ast|.|,|\cdot|(|)|\rangle\)

2. \(\langle\text{delimiter}\rangle ::= \langle\text{operator}\rangle|\langle\text{separator}\rangle|\langle\text{bracket}\rangle|\langle\text{declarator}\rangle\)

2.1 \(\langle\text{operator}\rangle ::= \text{search}|\text{go}|\text{change}|\text{changing}\)

2.2 \(\langle\text{separator}\rangle ::= \text{for}|\text{to}|\text{and}|\text{blank}\)

2.3 \(\langle\text{bracket}\rangle ::= \text{state}|\text{Turing machine}|\text{end}\)

2.4 \(\langle\text{declaration}\rangle ::= \text{alphabet}\)

3. \(\langle\text{expression}\rangle ::= \langle\text{search expression}\rangle|\langle\text{replacement expression}\rangle|\langle\text{transfer expression}\rangle\)

3.1 \(\langle\text{search expression}\rangle ::= \text{search}\langle\text{direction}\rangle\text{for}\langle\text{list}\rangle\)

3.1.1 \(\langle\text{direction}\rangle ::= \text{left}|\text{right}|R|L\)
3.1.2 $\langle\text{list}\rangle :: = \langle\text{external alphabet character}\rangle |$
   $\langle\text{list}\rangle\langle\text{blank}\rangle\langle\text{external alphabet character}\rangle$

3.2 $\langle\text{replacement statement}\rangle :: = \text{changing}\n\langle\text{external alphabet character}\rangle \text{ to }\langle\text{external alphabet character}\rangle | \text{changing rest to}\n\langle\text{external alphabet character}\rangle$

$\langle\text{replacement statement 2}\rangle :: = \langle\text{external alphabet character}\rangle \text{ change to }\langle\text{external alphabet character}\rangle$

3.3 $\langle\text{transfer expression}\rangle :: = \text{go}\ \langle\text{direction}\rangle \text{ to}\n\langle\text{label}\rangle | \text{go to }\langle\text{label}\rangle$

3.3.1 $\langle\text{label}\rangle :: = \text{state }\langle\text{number}\rangle|\langle\text{number}\rangle|$
   \text{stop}

$\langle\text{number}\rangle^* :: = \langle\text{digit}\rangle|\langle\text{number}\rangle\langle\text{digit}\rangle$

*restricted to three digits by our TASP processor

4. Syntax

4.1 $\langle\text{basic statement}\rangle :: = \langle\text{search expression}\rangle |$
   $\langle\text{search expression}\rangle\langle\text{replacement expression 1}\rangle$

4.2 $\langle\text{statement}\rangle :: = \langle\text{transfer statement}\rangle |$
   $\langle\text{replacement statement}\rangle$

4.2.1 $\langle\text{transfer statement}\rangle :: = \langle\text{external alphabet character}\rangle\langle\text{transfer expression}\rangle$

4.2.2 $\langle\text{replacement statement}\rangle :: = \langle\text{replacement expression 2}\rangle$
expression 2>|<replacement expression 2>
and <transfer expression>|<replacement expression 2>|<blank>|<transfer expression>

4.3 <machine block> ::= <block head><declaration>
<state block>|<machine block><state block>|<machine block>

4.3.1 <block head> ::= Turing machine <number>
(comment)

4.3.2 <declaration> ::= alphabet <list>

4.3.3 <state block> ::= <block label><basic statement><statement>|<state block>
<statement>

4.3.3.1 <block label> ::= state <number> (comment)

4.4 <program> ::= <machine block> end
III. IMPLEMENTATION OF THE LANGUAGE

Memory Utilization by the Translator.

The function of the translator is to create a disk file of quintuples for the TASP (11) processor from each input source program. Usually, the source input consists of a card deck of program blocks as described in Chapter 1 preceded by control cards in the format for OS3 (10) whose function it is to load the translator into the machine memory. However, by using other control card options, the source may be put in by other input devices.

The translator uses 73 file blocks of core storage, which is then overlayed, by the TASP processor (11), after the compilation of the quintuple list.

Translator-time Error Detection.

The translator prints out diagnostics after each detection of a source language error. Translation continues however, until the terminator card is reached or until 125 errors have been detected, which ever event occurs first. After detection of the first error, the TASP processor call is suppressed.

Translator Outputs.

The translator output is normally given to the standard system output file, the printer. A file for the
TASP processor is also created. The output for both files consists of the quintuples of the Turing machine which has been compiled. The programmer may request the output on a different device by using other control card options.

Only columns 1 to 72 of each card are interpreted by the translator. The other eight columns are available to the user as a label field, comment field, or sequence field. The Turing machine card, alphabet card, and state card must begin in column 1. All other cards have a free format. At translation time, each card is counted and assigned a line count starting with "1" for reference in error messages.

The format of the source deck is given in Figure 3 and the error diagnostic list in Figure 4.

Figure 3. Source Deck.
Figure 4. Diagnostic List.

NO TURING CARD
  missing a Turing machine card

NO ALPHABET
  missing an alphabet card

NO END
  missing an end card

TOO MANY ERRORS
  more than 125 errors encountered

NO DIRECTION
  no direction or an incorrect symbol was found after a search or a go

ILLEGAL STATEMENT AFTER CHANGE
  means as stated

SYNTAX ERROR
  an illegal statement encountered perhaps a misspelling

UNDEFINED TASK
  an illegal expression encountered

NOT IN THE ALPHABET
  alphabet character written which is not listed on alphabet card

ERROR IN SYMBOL LIST
  a word or two character symbol in symbol list
IV. UNIVERSALITY OF THE TURING MACHINE
PROGRAMMING LANGUAGE
PROCESSOR LANGUAGE

In this section the main assertion stated previously is proved. We show that any Turing machine quintuple may be encoded in this language. This will demonstrate that algorithms which are capable of being encoded as Turing machines may be written in this language, and thus any algorithm may be encoded in this language.

Lemma 1.

Let \( S \) be a set of programs which perform algorithms written in language \( L_1 \) and let \( T \) be a set of programs which perform algorithms written in language \( L_2 \). Let \( P \) be a mapping from \( S \) to \( T \) such that if \( m \) is a program of \( S \) and the correspondent in \( T \) is \( Pm \) then

i) The set of inputs to a program \( m \) is the same set as the set of inputs to \( Pm \).

ii) If \( i \) is an input to \( m \) then \( Pm(i) = m(i) \).

Thus the outputs are the same.

If such a function \( P \) does exist then every algorithm encodable in \( L_1 \) as an element of \( S \) is encodable in \( L_2 \) as an element of \( T \).

Proof.

This follows from the definitions. Note that our translator from Turing machine language \( L_1 \) to Turing machine \( L_2 \) is such a function.
THEOREM 1.

Consider a function $P: S \to T$ that satisfies the hypotheses of Lemma 1, and suppose there exists a map $Q: T \to S$ such that $PQ$ is the identity map on $S$. Then the set of algorithms encodable in $L_1$ as elements of $S$ equals the set of algorithms encodable in $L_2$ as elements of $T$.

Proof.

By Lemma 1 it is sufficient to demonstrate that every algorithm encodable in $L_1$ by $S$ is also encodable in $L_2$ by $T$. To show this, it is sufficient to show that $Q$ also satisfies the hypotheses of the lemma.

Let $m$ be a program in $T$ and $i$ be an input to $m$. It is necessary to show that $m(i) = Qm(i)$; that is, the output from $Qm$ when $i$ is the input is the same as the output from $m$.

$$m(i) = (PQm)(i) = [P(Qm)](i) = Qm(i)$$

The last equality holds because $P$ satisfies the hypotheses of the lemma.

QED

THEOREM 2.

Any algorithm may be encoded in the Turing machine processor language.
Proof.

Consider $S$ as the set of all programs which can be written in the Turing machine processor language. Let $T$ be the set of all Turing machines whose alphabet is a subset of the 48 keypunch characters. Basing ourselves on the fact that an algorithm must be performable by some Turing machine which stops, $T$ is the set of all algorithms, since Shannon (11) has shown that the restriction to 48 characters does not restrict the set of algorithms. Let $P: S \rightarrow T$ be the operation which transforms programs written in the Turing machine language into Turing machines. $P$ is the Turing machine processor. The hypotheses of the lemma are satisfied for $P$, because the TASP processor acts as a Turing machine simulator; i.e., the operation of a program $a \in S$ is defined to be the operation $Pa$ as a Turing machine.

So by Theorem 1, it is sufficient to find a map $Q: T \rightarrow S$ which acts as a right inverse for $P$. The remainder of the proof will be the construction of the map $Q$. Suppose $m \in T$ then $m = (s_1, \ldots, s_\alpha)$ where $\alpha$ is the number of states and $s_j$ are the states. Each $s_j = (q_j, \ldots, q_{j\beta})$ and the $q_{jk}$ the quintuples of the $j$th state and $\beta$ is the total number of symbols. Let

3. Actually reference (11) establishes that two symbols are sufficient.
\{\gamma_1, \ldots, \gamma_\beta\} be the set of symbols in the alphabet of \( m \).

Then, \( q_{jk} = (s_j, \gamma_k, \delta_{jk}, \gamma_{jk}, s_{jk}) \) where

- \( s_j \) is the state in which the Turing machine is in
- \( \gamma_k \) is the symbol observed under the Turing head
- \( \delta_{jk} \) is the direction \( \in \{\text{RIGHT, LEFT, PLACE}\} \)
- \( \gamma_{jk} \) is the new state in which the Turing machine goes
- \( s_{jk} \) is the new symbol to be written on the tape

Let \( Q_m = (h, b_1, \ldots, b_k, t) \) where

\[
\begin{align*}
 h &= \{\text{TURING MACHINE NO. } m\} \\
 b_j &= \{\text{ALPHABET } \gamma_1, \ldots, \gamma_\beta\}
\end{align*}
\]

And \( b_j = (l_{j1}, \ldots, l_{jk} + 2) \) where

- \( l_{j1} = \text{STATE } j \)
- \( l_{j2} = \text{SEARCH LEFT FOR } \gamma_1, \ldots, \gamma_\beta \)

For \( k > 2 \)

- \( l_{jk} = \text{CHANGE TO } \gamma_{jk-2} \text{ AND GO } \delta_{jk-2} \text{ TO } s_{jk-2} \)

\( T = \text{END} \)

\( h \) is the heading block

\( b_j \) are the state blocks and

\( t \) is the termination block.

Clearly, \( Q_m \in S \), because \( Q_m \) satisfies the rules of syntax explained in section 1.

To conclude this proof, it would be sufficient to show that \( P \) applied to \( Q_m \) yields \( m \) again. But this can be verified by following the Editor flow chart (see Appendix B). The Editor processes each block independently of the
processing of any other block. If one assumes that the Editor is confronted with $b_j$ then it is sufficient to show that the processor outputs $s_j = (q_{j1}, \ldots, q_{k\beta})$. The program works in the following manner. Given the input $b_j = (l_{j1}, \ldots, l_{j\beta+2})$ where

$$l_{j1} = \text{STATE } j$$

For $k > 2$

$$l_{jk} = \gamma_{k-2} \text{ CHANGE TO } \gamma_{jk-2} \text{ AND CO}$$

$$\delta_{jk-2} \text{ TO } s_{jk-2}$$

it will process each $b_j$ separately.

In the memory of the computer is set up a two dimensional matrix called ALPHABET. This array has four rows: ALPHABET, NEWALPHA, DIRECT, STATE, and $\beta$ columns. The contents of ALPHABET are $\gamma_1, \ldots, \gamma_\beta$. When the program reads a state card $l_{j1}$ it stores $j$ in the memory location called STATENO. This is done in the main program beginning at the location labeled PICKNO. After $l_{j1}$ is processed $l_{j2}$ is read, within the Editor, and then $l_{j2}$ is stored in the 20-word array labeled CARD. The card area is then searched for the word 'SEARCH'. When this is found, the direction following the string 'SEARCH' is put in storage location 'DIRECTION'. Then for $k > 2$, each $l_{jk}$ is successively read into the CARD array. That area is searched for $\gamma_{jk-2}, \delta_{jk-2}, s_{jk-2}$ which are then placed in
the \( j \)th columns of NEWALPHA, DIRECT, and STATE respectively. When a \( b_j \) has been processed the program jumps to the subroutine called CODEGEN. In this subroutine a check is made to see if every column of NEWALPHA for missing quintuples (an empty column indicates a missing quintuple). Then, if no errors have been detected, representing the \( j \)th row, the following is outputted onto the disk file:

Contents of STATENO, contents of the \( j \)th column of ALPHABET, contents of the \( j \)th column of NEWALPHA, contents of the \( j \)th column of DIRECT, contents of the \( j \)th column of STATE respectively.

At this time STATENO contains \( j \), the \( j \)th column of ALPHABET contains \( \gamma_j \), the \( j \)th column of NEWALPHA contains \( \gamma_{j\beta} \), the \( j \)th column of DIRECT contains \( \delta_{j\beta} \), the \( j \)th column of STATE contains \( s_{j\beta} \). This is done for each \( b_j \). Thus the output of the processor consists of \((q_{j1}, \ldots, q_{j\beta})\) where each \( q_{jk} = j, \gamma_j, \delta_{jk}, s_{jk} \).

QED
EPILOGUE

The language as it now stands seems quite satisfactory in clarifying and simplifying the writing of most Turing machines. However, for very long Turing machines there is an addition to the language which would further clarify and reduce the structure of complicated algorithms, a modification which has not as yet been implemented for use on the CDC 3300, namely the patch. The patch would enable one to introduce a subroutine feature into Turing machine programming. The TASP processor does contain a feature whereby a collection of Turing machines may be operated serially, as in the composition of functions. However, this feature, which would allow the use of two short Turing machines in place of one long Turing machine, does not seem to truly reduce the structure of the resulting algorithm. A patch is a method to allow parallel placement of Turing machines instead of sequential placement.

The modification to the description of Chapter 2 necessary for this addition to the language is obtained by replacing paragraph 4.3.3 by the following:

<state block> ::= <block label><basic statement> <statement>|<state block><statement>|<patch block>

<patch statement>

<patch block> ::= <block label> patch <number>

<patch statement> ::= <external alphabet character>
A typical patch block would be:

STATE $j$
PATCH $n$

$A_1$ is $B_1$
$\vdots$
$A_m$ is $B_m$

where $j$ is the state number of the patch state block, $n$ is the Turing machine to be patched to the Turing machine at that stage. $A_1 \ldots A_m$ are the external alphabet characters in the original Turing machine which are to play the role of the external alphabet characters $B_1 \ldots B_m$ in the Turing machine $n$. When the processor encounters a patch block, it starts to execute the Turing machine $n$, substituting characters as described above, until it reaches a STOP command in Turing machine $n$. At that time it returns to the calling Turing machine at the state block following the patch block.


APPENDIX 1
Sample Problem
Input and Output
TURING MACHINE TRANSLATOR 12/19

0001 TURING MACHINE 1
0002 ALPHABET 0 1 A R
0003 STATE 1
0004 SEARCH RIGHT FOR 0 AND 1 AND A AND B
0005 0 CHANGE TO 0 AND GC LEFT TO 2
0006 1 CHANGE TO 0 AND GC LEFT TO 3
0007 A CHANGE TO 0 AND GC LEFT TO 5
0008 R CHANGE TO 1
0009 STATE 2
0010 SEARCH LEFT FOR 0 AND 1 AND * AND A AND B
0011 0 CHANGE TO A
0012 1 CHANGE TO R
0013 * CHANGE TO A AND GC L TO 4
0014 A CHANGE TO 0 AND GC R TO 7
0015 R CHANGE TO 1 AND GC R TO 6
0016 STATE 3
0017 SEARCH LEFT FOR 0 AND 1 AND * AND A AND B
0018 0 CHANGE TO A
0019 1 CHANGE TO R
0020 * CHANGE TO A AND GC L TO 2
0021 A GC R TO 7
0022 R GC R TO 8
0023 STATE 4
0024 SEARCH LEFT FOR 0 AND 1 AND * AND A
0025 0 GC R TO STATE 3
0026 1 CHANGE TO B AND GC R TO 5
0027 * GC R TO 1
0028 A CHANGE TO *
0029 STATE 5
0030 SEARCH RIGHT FOR 0 AND 1 AND * AND A
0031 0 GC R TO 4
0032 1 CHANGE TO R AND GC LEFT TO 6
0033 * CHANGE TO A AND GC R TO 5
0034 A CHANGE TO * AND GC L TO 4
0035 STATE 6
0036 SEARCH LEFT FOR 0 1 * A B
0037 0 CHANGE TO A
0038 1 CHANGE TO R
0039 * CHANGE TO 0 AND GC R TO 7
0040 A CHANGE TO 0 AND GC R TO 2
0041 B CHANGE TO 1 AND GC R TO 2
0042 STATE 7
0043 SEARCH RIGHT FOR 0 1 * A B
0044 0 GC R TO 1
0045 1 GC R TO 1
0046 * CHANGE TO 0 AND GC L TO 6
0047 A CHANGE TO 0
0048 R CHANGE TO 1
0049 STATE 8
0050 SEARCH R FOR 0 1 * A R
0051 0 GC L TO 1
0052 1 GC L TO 1
0053 * CHANGE TO 1 AND GC L TO 6
0054 A CHANGE TO 0
0055 B CHANGE TO 1
0056 END

UNIVERSAL TURING MACHINE
END OF PROGRAM EXECUTION
APPENDIX 2

Flow Charts of

the Turing Machine Processor
APPENDIX 3

Program Listing
null
ENI 3.1
LOA EM
INA.s -1
MUA =0
INA EMS
SWA LOOP

LOOP
LOA #11
STA ERRN.1
IJD LOOP.1
ENA EMS
ENO 11
WRITE1 OUTUNIT
ENA MESS
ENO 5
WRITE1 OUTUNIT
RTJ ERRPRINT
NOP
UJP START

EM BSS 1
FRMS BCD 7;COMPILATION TERMINATED
ERRN BCD 4, NO TURING CARD
EMS BCD 4, NO ALPHABET
ENR BCD 4, NO END
MESS BCD 4, TOO MANY ERRORS
END IDENT ERRPRINT
ENTRY ERRPRINT

ERRPRINT UJP **
STI 31.1
STI 32.2
STI 33 3
LDA ERRCNT
ATJ.EQ ERRPRINT
ENR HD
ENO 4
WRITE I OUTUNIT
ENA HD
ENO 1
WRITE I OUTUNIT
LDA ERRCNT
SWA LOOPEND
ENI 0.1

LOOP
LOP ERLLST.1
ENA 0
ENI 3.3
SHAQ 3
IJD LP.3
STA ERRLINE.2
LOP ERLLST.1
SHAQ 12
ENA 0
SHAQ 12
INA.s -1
MUA =0
INA X1
SWA RL.3
ENI 8.2
PLP LOA #12