IS-T-637

The META-T Compiler-Compiler

Anthony Peter Lucido

Ph.D. Thesis Submitted to Iowa State University May 1974

Ames Laboratory, USAEC
Iowa State University
Ames, Iowa 50010

Date Transmitted: July 1974

PREPARED FOR THE U. S. ATOMIC ENERGY COMMISSION
DIVISION OF RESEARCH UNDER CONTRACT NO. W-7405-eng-82

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Available from: National Technical Information Service
Department A
Springfield, VA 22151

Price: Microfiche **\$1.45**
Printed Copy \$5.45

The META-T Compiler-Compiler

by

Anthony Peter Lucido

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Roy Keller
In Charge of Major Work

For the Major Department

For the Graduate College

Iowa State University
Ames, Iowa

1974

TABLE OF CONTENTS

	Page
ABSTRACT	v
INTRODUCTION	1
THE META-T MACHINE	12
THE META-T PROGRAMMING LANGUAGE	66
THE META-T COMPILER	89
CONCLUSIONS AND FUTURE WORK	95
BIBLIOGRAPHY	98
ACKNOWLEDGMENTS	100
APPENDIX	101

The META-T Compiler-Compiler

Anthony Peter Lucido

Under the supervision of Roy F. Keller
From the Department of Computer Science
Iowa State University

The need for mechanical aids to both the translator definition and implementation phases of translator generation has been known for many years. Many products have been devised and have enjoyed various degrees of success. The META-II (12) compiler-compiler is one of the first to use a metalanguage resembling BNF (8) with semantics capability.

META-PI (9) is the first practical version of META-II, and provides in its working environment a production system for generating translators. Without modifications, it is limited to use on either the RCA SPECTRA/70 or the IBM 360/370 series machines. Because of certain limitations inherent in its design, META-PI cannot be used for defining and implementing translators that produce arbitrary object texts. Also, it is difficult using META-PI to specify multipass translation routines.

In this dissertation, the META-T compiler-compiler is developed. It provides users with a powerful definitional facility, the META-T Programming Language (MTPL), which may be used for the implementation of translators after definition.

A translator may also be implemented by hand-coding the definition using some programming language.

A host machine has been defined for MTPL. This host machine, the META-T Machine (M-TM), is an adaptation of desirable features from existing machines and the addition of other features, as needed, to provide a powerful structure that may be implemented on existing machines using an interpreter. The META-T Compiler (M-TC) is developed to translate text written in MTPL to the equivalent text in M-TM code. These three parts comprise the META-T compiler-compiler.

INTRODUCTION

The META compilation system to be described allows one to specify a program for translating a source text to some resultant text. The system is called META-T. META-T is oriented more towards a laboratory environment than to a production environment. Thus, design considerations were geared towards providing a versatile expression tool.

The basis for the META-T system came from the META-series of compiler-compilers consisting of systems such as META-II (12), META-3 (11) and META-PI (9). Also, non-META-series systems were considered. Cogent ideas from all areas, including personal experience, led to the concepts embodied in META-T.

META-T is an aid to the translation process. The need for aids to the translation process is evident to anyone who has attempted the generation of a translator. There is no suitable language in which to specify the needed transformations. For example, the generation of a compiler using only an assembly language which is not enhanced by a macro processor is not a desirable task.

A language or system which is suited to the task of translator generation eases the task of generating a complex translation routine. Many operations that are tedious to perform can be automated, and it is helpful to have a syntax

that allows language definitions to be easily stated. META-T is an attempt to provide a good compiler generation aid.

There are many approaches that can be taken for writing a translator. The program can be written in an assembly language. This is not a satisfactory solution unless a large, helpful, subroutine library and/or macro library/facility exists. The problem is mainly one of bulk--an assembly language program for any moderately large translator will require many instructions, and the writing and debugging of such code will tax any programmer.

Another approach is to use one of the available higher-level languages. Most higher-level languages, such as FORTRAN (3), PL/I (5), etc., are not really well suited to symbol manipulation. PL/I includes string processing facilities, but most of the tasks which are needed are still cumbersome unless support programs are devised. Using FORTRAN is absurd without a rather extensive set of string processing routines. Thus, one is again faced with many difficulties.

Over the years, many automatic compiler generators have been devised. These have enjoyed varying degrees of success. Some have been only feasibility studies, in the sense that they have been developed to test concepts. Some have been production systems devised for everyday use.

One of the automatic compiler generators (hereafter also termed a compiler-compiler) qualifying as a feasibility study

is META-II (12). Whereas Backus-Naur Form (BNF) (8) is used only to specify the syntax of a set of languages, META-II can be used to specify the syntax and post-syntax (anything done after the syntax testing phase proves successful) of a language. It incorporates a top-down, left-to-right, parsing strategy. This compiler-compiler severely limits the complexity of the language that a user is able to define, however, it does show a place from which to start.

META-PI (9) is a production level compiler-compiler. It also incorporates a top-down, left-to-right, parsing strategy and a back-tracking mechanism. Languages previously described using META-PI are relatively easy to modify.

META-PI includes a translate-time stack onto which character strings, intermediate text and other information acquired during translation may be placed. META-II does not include this facility. In fact, META-II is very limited in its code generation ability. META-PI greatly reduces the restrictions on how complex a language may be described, as it is able to store type information for names, to retain intermediate information on a translate-time stack, and to perform at least partial backup.

Two other compiler-compilers not in the META-series include FSL (1) and Brooker and Morris's table driven compiler generator (10). These two systems are quite different in appearance. FSL has two parts to a single "program". The first

part, a syntax description and action invoking specification, has an abstract appearance. The second part resembles an algorithmic language, with conditional execution, data manipulation/definition, and code generation facilities.

The Brooker and Morris system has a rather different format, resembling a cross between BNF and a macro language processor. It is based on the use of format templates to specify the form of a source statement. An associated format routine specifies the code to be generated.

Another system to consider is the Vienna Definitional Language (6), called VDL. VDL is designed to handle lists and trees as basic data items. The main import of VDL is that it is used for the specification of interpreters for languages, rather than simply translators for the languages. In specifying the meaning of the language being defined, one sees the ramifications of each construct.

The method of use of all META-series compiler-compilers is the same, and is:

- (a) The translator writer expresses the syntax and semantics of the language in terms of the meta-language of the compiler-compiler.
- (b) The compiler-compiler accepts the results of (a) and generates a compiler for the host machine.
- (c) With the source text to be translated comprising the initial input string, the results of (b) are

executed on the host machine yielding whatever post-syntactic text that is defined.

To use a compiler-compiler merely for translator definition, only (a) above is performed.

Features needed by a translator writer include:

- (a) Data definition and management methods;
- (b) Provisions for the testing for classes of terminal symbols (primitive elements/items);
- (c) The ability to generate the required post-syntactic text;
- (d) Some ability to manage the input string;
- (e) The ability to generate auxiliary input strings.

In the following discussion each of the above points will be considered in turn.

In META-T there are five data types:

- (a) NUMERIC - a numeric integer type which allows one to establish counters and switches for use in the translation process;
- (b) STRING - a character string type useful for keeping intermediate character information generated during the translation process;
- (c) STACK - the conventional pushdown stack, useful for storing intermediate information from the translation process;

- (d) HASH - a table data type able to store/retrieve simple strings;
- (e) VECTOR - a one-dimensional array having entries of character or numeric integer types, intermixed as required. Using HASH and VECTOR data areas, one can create symbol tables as desired.

The user may declare as many of the above types, as required, on a totally dynamic basis and destroy them as they become obsolete. No size restrictions are imposed on the number or the length of entries. META-PI, in contrast, has one translate-time stack and one translate-time symbol table (with a very rigid set of flag/data areas). META-II does not even include a stack.

As with the rest of the META-series compiler-compilers, META-T has the ability to test for the presence of terminal or primitive items in the input string. If, at the current point in the parsing procedure, one requires a test for the occurrence of an identifier, for example, then this may be done using a primitive function (.ID in the META-series compiler-compilers). An internal flag is set to indicate whether the test was successful or not and this test is used for flow control of the translation program. Also, if the test is successful, the primitive item is instated as the current item, replacing any previously such found item.

META-PI includes a primitive to test for real (floating

point) numbers. The test is for an optionally signed decimal number, with decimal point and an optional exponent part. The exponent part is in FORTRAN's E or D format. Along with the primitive that recognizes floating point numbers is a support routine that converts the floating point number so recognized (still in character form) to its internal, RCA SPECTRA/70 representation. If another floating point representation is desired, both the productions that describe the form are required, and also another support module must be provided to accept this new format and convert it to the machine form.

As META-T is not tied to one specific object text, the META-PI support routines become unnecessary. M-TM (and MTPL) include the instructions (primitives) to test for not only optionally signed decimal integers, but also unsigned decimal integers and single digits. Using these and other facilities, any desired floating point representation may be defined and used.

As another point of comparison between META-PI and META-T, all tests for syntactic items done using META-PI are blank insensitive. Thus, any blanks between the current position in the input string and the next non-blank character are ignored. This limits one to treating only free format text (as in ALGOL, PL/I, etc.). When fixed format text is encountered using META-PI, a text editor external to META-PI is

employed to reformat the original input text and test for correct field usage. There is, using META-T, the ability to treat both fixed and free format text directly. Syntactic testing may be either blank insensitive, as in META-PI, or blank sensitive, as required when treating fixed format text.

META-T and META-PI differ in some other areas. META-PI incorporates some object code optimization facilities, whereas META-T provides a rich language for use in generating optimization routines. META-T is able to describe multipass translators, but META-PI cannot. Also, META-PI has a symbol table that is only able to handle a FORTRAN environment. Using META-T, one is able to generate whatever symbol table format that is desired.

The area of text generation requires some careful consideration if machine independence is to be achieved. The first "aid" to eliminate is the automatic assembler present in META-PI. One must return to a simpler output routine, as in META-II. Once one requires an internal assembly routine, machine dependence is imposed.

All output for a language is representable by characters taken from an alphabet. Thus only the ability to output character strings is needed. In META-T the output function will only transmit its character argument to the output file. No automatic assembly routine is present, so the output is free of any intrinsic structure of format requirements. When

direct machine code needs to be produced, META-T is able to be used, since an object deck is nothing more than a character string.

To generalize post-syntactic text generation is one thing, but one must first successfully parse (recognize) a sentence. Here, generality is again required. To allow back-up, one must be able to traverse the input string in both directions. META-PI has an adequate algorithm for allowing backup, and this algorithm is adopted essentially intact into META-T. One feature added to the basic algorithm is that an internal flag, accessible by the translator writer, was created to signal when the right-hand end of the string has been encountered. This end-of-string condition provides another method of logic control.

To generate multipass translators, a technique must be established to allow rescanning of the most recent input string, or some version of it. In contrast to many compiler-compilers, one may do this using META-T. In order to generate a new input string, a user executes whatever transformations are required and outputs the results of these transformations to the "new input" string area. Then, when the user is finished with the current input string and requires the newly generated one, it is simply a matter of executing the function .RESCAN to get it. This procedure has one characteristic that should be noted. When a new input string is gen-

erated and instated, the previous input string is destroyed. What happens, in effect, is that the new input string replaces the current input string, and the new input string is cleared. If one merely wishes to rescan the original input string, then no new input string is produced. Upon executing .RESCAN, the current input string is kept as the new input string, and all needed pointers are reset. (It should be noted that after a .RESCAN, the input string position pointer is reset to the beginning of the string.) The concept of a multipass translator is not possible with the approach that META-PI takes. When using META-PI under the BTSS (Basic Time Sharing System) system, there are pre-processor features present which one may use to do any initial massaging of the input string; but, META-PI does not itself have a structure allowing this.

One complaint often heard about compiler-compilers is that the object code produced is inefficient, it is unoptimized. Among others, O'Neil recognized this problem, and because of this, META-PI has several code optimization functions. As META-PI operates with one object code type, it is easy to locally optimize the code and a more complete optimization facility may be implemented.

META-T consists of three sections. These are:

- (a) META-T Machine - This is a pseudo-machine, the architecture of which has been specifically

tailored to the requirements of the META-T Programming Language.

- (b) META-T Programming Language (MTPL) - This is a high level programming language which the user employs to express the logic of his translator. It corresponds to the metalanguage mentioned earlier.
- (c) META-T Compiler - This is a program written for the META-T Machine which accepts an MTPL program as input and outputs the corresponding object code for the META-T Machine.

The design of the META-T Machine incorporates many generalizations from conventional machine architecture. Included are facilities for the dynamic allocation/deallocation of data areas, a generalized move instruction, indirect data referencing, and in short, everything needed to mirror the power of MTPL, but in a form which may be implemented as an interpreter. The use of a pseudo-machine instead of a commercially available machine allows portability.

MTPL incorporates all of the features just discussed in a form which permits one to state the transformations required for the translator being defined.

The META-T Compiler is a translation routine to generate a META-T Machine program from a MTPL program. It is written in META-T Machine Language and executes on an interpreter for the META-T Machine.

THE META-T MACHINE

The intent of the META-T Compiler-Compiler is to provide a facility for the production of machine independent translators; i.e., translators that will execute on many different computers. As the introduction indicated, the approach used is to divide the META-T Compiler-Compiler into three parts. This chapter will detail both the architecture and instructions of the META-T Machine (M-TM). M-TM is a hypothetical machine, an amalgamation of ideas drawn from many different computers, ideas that have been extended and modified as required to create an architecture and an instruction set that may be implemented in a fairly straightforward manner on the present mix of computers.

The storage for M-TM is divided into two independent parts, the Program and Data memories. Only instructions may reside in the Program memory, and only data may reside in the Data memory. All data manipulation instructions reference the Data memory. Thus, no instruction modification may take place at execution time, ensuring a reentrant and recursive environment for the program. Both memories are managed using a virtual storage methodology.

The Data memory is used for both user allocated and pre-defined (internally maintained) data area storage. When a user allocates an instance of a user data area, the storage

is allocated from Data memory. When freed, the corresponding storage is released for further use.

A user's program is assumed to be present in memory before the start of execution. Thus, the number, n , of instructions in a program is known before execution. The Program memory is addressed from zero on, but a user's program starts in location one. Location zero contains a permanent instruction (the Scan and Execute) which is used for program management. Each instruction, in symbolic form, is assumed to occupy one location of the program memory.

The form of an M-TM program is a set of procedures, the set of which is terminated by a PROGEND instruction (see the listing of instructions below). Each procedure is of the form of a HEAD instruction, including a required, globally unique label; a set of instructions other than HEAD, PROGEND or END; and finally an END instruction.

An M-TM instruction is of the form:

[label] : opcode [operand(s)] ;

The form [item] means that the item may not be present in some cases, or is optional in others. A label, indicated by [label], is optional. If present, it must be unique within the procedure containing it. The form of a label is, in BNF:

```

<LABEL> ::= <FIRST CHARACTER> | <LABEL> <CHARACTER>
<FIRST CHARACTER> ::= # | A | B | ... | Z
<CHARACTER> ::= <FIRST CHARACTER> | 0 | 1 | ... | 9 | _

```

The term opcode is the operation code, the instruction's name.

It is always required and must be from the set of instructions to be given below. The term [operand(s)] is used to note that a list of operands (parameters to the instruction) must be present if required by the instruction, or must be absent if not so required. A blank must separate the opcode from any operands. The characters ":" and ";" are separation characters and are always required.

As was mentioned earlier, there are two types of data areas, predefined and user allocatable. The predefined data areas are:

- a) Current Input String - This area contains the string to be parsed. There is no maximum length string, thus there is no maximum Current Input String length.
- b) Current Input String Pointer - This area holds a value corresponding to the current character position in the input string.
- c) New Input String - This string results from the issuing of NEWIN instructions. It comprises a potential input string and may replace the current input string upon command.
- d) New Input String Pointer - This area holds a value corresponding to the current character position in the New Input String.
- e) Output Buffer - This area holds intermediate and final values of character expressions. The contents

of the Output Buffer may be processed further, stored, or transmitted to the output device.

- f) Output Buffer Pointer - This area holds a value corresponding to the current character position in the Output Buffer.
- g) True/False Indicator (TF flag or TF indicator) - This flag is set either explicitly or as the result of syntactic tests.
- h) EOS Flag - This is the End-of-String flag that indicates that the last operation treating the Current Input String positioned the Current Input String Pointer to the right of the last character stored in the Current Input String.
- i) EOF Flag - This is the End-of-File flag. It is used to signify that the last READ instruction was attempted and no more records were available.
- j) Computation Stack - This stack contains numeric values that are used in data referencing or as results (or intermediate results) in arithmetic computations.
- k) Computation Stack Pointer - This area holds a value that points to the top entry on the Computation Stack.
- l) Instruction Pointer - This area is used to indicate the address in the Program memory of the current instruction being executed.

- m) Next Label Counter (see the GN instruction) - This is a counter which is incremented when a new internally generated labelling string is required. These labelling strings are used in the production of object text by the user.
- n) Control Stack - This stack holds entries generated from the execution of CALL and LATCH instructions. The stack entries have two forms, depending on which instruction generates them. For a CALL instruction the form is:

CALL/LATCH flag is set to CALL
Return Point

For a LATCH instruction the form is:

CALL/LATCH flag is set to LATCH
Return Point
Current Input String Pointer value at the time of the LATCH

The Return Point value is the address, in the Program memory, of the CALL (LATCH) instruction creating the entry.

- o) Control Stack Pointer - This area holds a pointer referencing the top entry on the Control Stack.
- p) Labels Stack - This stack contains entries generated by the execution of a HEAD instruction. The form of

an entry is:

Label Table Reference
Labels List Pointer

The Label Table Reference field is a pointer into the Program Label Table (explanation below). The Labels List Pointer field references the list of internally generated labels (via the GN instruction) for the current activation of this procedure.

- q) Labels Stack Pointer - This area contains a pointer to the top entry on the Labels Stack.
- r) Program Label Table - This table contains both HEAD instruction labels and the labels used in each procedure. The table is in the form of a set of back-chained entries, as:

entry	λ
entry	•
•	
•	
•	
entry	•
entry	•

(λ - empty or grounded node)

Each entry is of the form:

Backchain pointer
HEAD instruction label
Location of the HEAD instruction, minus 1
Number of instruction labels for this procedure
Label
Location of the labelled instruction, minus 1
⋮
Label
Location of the labelled instruction, minus 1

Each entry is for one procedure. The "Number of instruction labels for this procedure" field gives a count of the number of (label; location of the label, minus 1) pairs following this field. The instruction labels are the labels on instructions (other than the HEAD instruction) for the procedure. For an entry, all labels kept as elements must be unique, although instruction labels may be repeated in other Program Label Table entries. Note that the address values stored are one less in value than the location of the instruction being referenced. This compensates

for the instruction execution method (to be covered shortly).

- s) Program Label Table Pointer - This area contains a pointer to the head of the last entry in the Program Label Table.
- t) Pre-execution Error Flag - This flag (indicator) is set if an error condition is encountered during the execution of the Scan and Execute instruction.
- u) User Data Area Symbol Tables - There are five separate tables, one for each user data type. Since user data areas are dynamic, these tables are maintained as doubly linked lists. Each table is composed of a pointer to the first entry on its list, and a set of (doubly linked) entries. An entry is of the form:

Back Pointer
Front Pointer
Name
Pointer to data area

The Back and Front Pointers are used to effect the double linking. The Name field is the data area name. The "Pointer to data area" field contains a pointer to the allocated data area.

There are five user data types. Users may freely allocate and deallocate instances of these data areas, and as

shall be seen, the referencing ability is versatile enough to allow that five different data areas, one of each type, may all have the same name.

The five data types and their storage structures are:

- a) NUMERIC - optionally signed decimal integer of unlimited precision. The form of a NUMERIC variable is:

Numeric data

- b) STRING - character string of unlimited length. The form of a STRING variable is:

String data

- c) VECTOR - one-dimensional array with an unlimited number of elements. Each element may be of numeric or character mode. The storage structure for this is:

Maximum index used so far

Pointer to the first element

Pointer to the last element

An element holds the data value itself. The structure of an element is:

NUMERIC/STRING type flag

Index of the element

Data

The NUMERIC/STRING type flag indicates whether the data is of numeric or character type.

- d) STACK - pushdown list with an unlimited number of entries able to be entered. The storage structure is:

Pointer to the top stack element

A stack element is of the form:

Pointer to the preceding stack element
--

Character string

When the stack is empty, the pointer to the top stack element area has the value "null". The first stack element also has the "Pointer to the preceding stack element" field set to "null".

- e) HASH - table that stores simple, unlimited length, character strings. The table management routine is based on a hashing technique (2). The structure of a HASH area is:

Predecessor link
Successor link
Flag field
Pointer 0
<div style="text-align: center;"> . . . </div>
Pointer 9

The Predecessor and Successor links allow one HASH area to be linked to other HASH areas (see the FIND instruction). The Flag field is present so the user may insert information about this table to be accessed, by the user, in later processing. The entries, Pointer 0 through Pointer 9, are pointers to forward linked lists. These ten lists have elements of the form:

Pointer to the next element
Character string

When a character string is entered into a HASH area (also called a HASH table, or simply a table), it is first used as the argument of a hash value generation algorithm. This algorithm returns a value *i* between zero and nine. Then, while a count of the number of list elements encountered is maintained, the new entry is compared against the contents of the character string field for each element on the list referenced by the respective pointer. If this sequential search results in a match (hit), then a new entry is not made. If no match is made, then a new element is linked to the end of the list, and the value of the character string field of this new element is set to the value of the new entry. The

total number of elements on this newly augmented list is termed the J value. The pair of values (I,J) is sufficient to later recover the value of this newly created element.

In respect to the HASH areas, it should be remembered that only a simple character string is retained as data. Thus, one may ask how it is possible to construct a symbol table. Since the position of the stored item is returned to the user, the (I,J) pair can be used to generate a unique subscript for use with an accompanying VECTOR area (or as many VECTOR areas as required) in keeping other needed data. The J value is in the range $[0, 1, \dots]$, and the I value is in the range $[0, 9]$. Using the formula $10 \cdot J + I$ gives a unique subscript which may be used to index a VECTOR. All other needed flags and data items may be stored using the needed number of VECTOR areas.

The above data areas have initial values set as:

- a) NUMERIC - Upon allocation, the area is set to zero.
- b) STRING - Upon allocation, the area is set to empty.
- c) VECTOR - When an element is first referenced as an operand for a source field (operand of a load operation, for example) and has yet to be stored into, the element is initialized as for a) or b) above, depending on the type of the element.
- d) STACK - The area is initialized to empty.

- e) HASH - Each list is set to the empty value for the pointers, and the link and flag areas are set to "null".

As one name may be used in the allocation of five separate and distinct user data areas, some method is needed for differentiating between them. The instruction being performed will indicate whether a STACK or a HASH area is required. For a character or numeric operation, a method must exist to differentiate between a scalar and a vector argument. Subscripting information is used in this case. The subscripting information is placed on the Computation Stack by instructions preceding the one needing such information. For a positive subscript value, the reference is to a VECTOR data type, and the value is the index of the desired vector element. Otherwise, a scalar reference is made. When the subscript value has been used and is no longer needed, it is popped from the Computation Stack. An exception to this method is taken for indirect CALL, LATCH and branch references, as shall be seen shortly. Although ignored for STACK and HASH area references, subscripting information must still be present on the Computation Stack. It is merely popped from the Computation Stack and ignored.

The instruction execution cycle of M-TM is:

- a) Increment the Instruction Pointer by one.
- b) Check the operation code (opcode) for one of the

M-TM opcodes. If the opcode is not correct, issue an error message and terminate execution. Otherwise, if operands are required, check to see that they are present and of the required type for the instruction. If any error is encountered, issue an error message and terminate execution.

c) Execute the instruction.

This cycle is repeated until an error condition occurs, an instruction terminates further processing, or the value of the Instruction Pointer exceeds n (the number of user supplied instructions) and the cycle has not ended.

The execution of an M-TM program proceeds as follows:

a) Initialize all predefined data areas to:

1. Empty for strings;
2. "Null" for pointers;
3. Empty for stacks;
4. "Off", or false, for flags/indicators.

b) Set the Instruction Pointer to -1.

c) Initiate the instruction execution cycle.

Thus, execution begins with the (always present) instruction in location zero of the Program memory.

A very important consideration in the writing and execution of an M-TM program is the scope of variables. For user data areas the scope is global, in that when an area is allocated by a user in one procedure, it is automatically able to

be referenced by every other procedure in the program. Pre-defined data areas also have a global scope.

Labels have a more restricted scope. Labels on HEAD instructions are global, but may only be referenced by CALL or LATCH instructions. Labels on other instructions are known only within the procedure in which they occur. One procedure may not access another's non-HEAD instruction labels. This convention is assured by the method of maintaining the Program Label Table (PLT) linkages. A HEAD instruction creates an entry on the Labels Stack that points to its entry in the PLT. Then, for a branch, the target label is used as a search argument against those instruction labels present in the indicated PLT entry. If present, the transfer of control is made; otherwise, an error condition occurs. Thus, there is no chaining of PLT entries as would be done for a block structured format.

Normally, references to data and labels are made directly. However, there are instances when it is desirable to perform indirect references to both data and labels. There is a mechanism for doing this using M-TM. Indirect references are made by using a STRING area to hold the actual name to be used. All subscripting information is applied to this actual (resolved or final) name. To denote an indirect reference, the STRING variable name used as an operand is preceded by an "at" sign(@). Nearly all data area referenc-

ing instructions may employ indirect addressing. The only non-data manipulation instructions which may use indirect referencing are the CALL, LATCH, and branching instructions. In these cases, the STRING variable used as the operand would have no subscripting information present for it on the Computation Stack.

The eleven different operand types are:

- 1 - integer literal in the form:

=length.integer

where the length is the number of characters, including sign, that comprise the integer. The equal sign (=) and the decimal point (.) are required delimiters.

- 2 - direct or indirect reference to a NUMERIC area.
- 3 - character literal in the form:

=length.string

where the length is the number of characters in the string. The equal sign (=) and the decimal point (.) are required delimiters.

- 4 - direct or indirect reference to a STRING area.
- 5 - direct or indirect reference to a STACK area.
- 6 - direct or indirect reference to a VECTOR area.
- 7 - direct or indirect reference to a HASH area.
- 8 - type 1 or 2.
- 9 - type 3 or 4.

10 - direct or indirect reference to an instruction label.

11 - direct or indirect reference to a HEAD instruction label.

During execution, a type error causes an error message to be issued and execution to be terminated.

We are now ready to list the instructions and give the action for each instruction. This list will be broken up into four groups. Unless otherwise stated, only those actions given are performed. Thus, for example, unless explicitly stated, the TF indicator (true/false indicator) is not modified. The groups are:

a) Control

1. SE - no operands. This is the Scan and Execute instruction, which is a permanent instruction kept in location zero of the Program memory. Given control first, it steps sequentially through memory, constructing the Program Label Table (PLT) and checking the M-TM program format. The action is:

i) Upon encountering a HEAD instruction,

a. If no label exists, then an error message is issued, the Pre-execution error flag is set to true ("on"), and a null length label

value is assumed. Continue with step b.

- b. The label for the HEAD instruction is checked against existing HEAD instruction label entries already present in the PLT. If the PLT is empty, then the search fails and the process continues with step c. If a duplicate is found, an error message is issued, the Pre-execution error flag is set to true, and the process continues to step c.
- c. An entry is backchained onto the PLT, with the PLT pointer being adjusted to point to this new entry. The HEAD instruction label is entered into the appropriate field of this newly created entry; the location, minus one, of the HEAD instruction is stored in the respective field; and the "Number of instruction labels for this procedure" field is set to zero.

- ii) If an instruction is encountered that has a label and is not a HEAD or PROGEND instruction, then the action is as follows:
 - a. If the PLT is empty, issue an error message, set the Pre-execution error flag to true, and go on to the next instruction. Otherwise, continue with step b.
 - b. Test the instruction's label against the set of instruction labels already present in the PLT entry referenced by the PLT pointer.
 - c. If a duplicate is found, issue an error message, set the Pre-execution error flag to true, and go on to the next instruction. Otherwise, continue with d.
 - d. Increment the "Number of instruction labels" field for this procedure (hereafter termed the NL field) value by one, add an element to this PLT entry for

the label and its location
(minus one), and copy in those
field values.

- iii) Except for a PROGEND instruction, unlabelled non-HEAD instructions do not require any treatment. They are skipped over and the next instruction is treated as required.

The above action is repeated until either a PROGEND instruction is encountered or the number, n , of M-TM program instructions (not including the SE instruction in location zero) is exceeded. If a PROGEND is encountered, the action is:

- i) Ignore any label that may be present.
- ii) If the PROGEND is not in location n of the Program memory (and thus is not the last instruction for this M-TM program), issue an error message and set the Pre-execution error flag to true.
- iii) If the Pre-execution error flag has been set because of an error in any instruction, issue an error message and terminate further program execu-

tion.

- iv) As it may now be assumed that no errors are present in the syntax of the M-TM program, the SE instruction terminates. Because of the action of the instruction execution cycle, we now begin with the first instruction of the user's program

If the number, n , of instructions is exceeded, an error message is issued and execution terminates. The SE instruction also checks the overall M-TM program format. The procedure format must be kept, or an error message is issued and the Pre-execution error flag is set. Thus, if any error condition is encountered, the program will not be allowed to proceed into execution.

- 2. HEAD - no operands, but requires a label to be present. This label must not be repeated as the label on any other HEAD instruction in the program. The action is:

- i) Push a new entry onto the top of the Labels Stack and adjust the Labels Stack Pointer to reference this new entry.

- ii) Set the Labels List Pointer field to "null", indicating an empty list.
- iii) Using the label on the HEAD instruction as the search argument, locate the PLT entry for this procedure. The Label Table Reference field of the current Labels Stack entry is set to reference the PLT entry just found.

The above found PLT entry is needed to acquire the environment for this procedure. When a branch is performed (not a CALL or LATCH), the operand is used as a search argument against the instruction labels stored for this procedure only. Thus, locality of branching is ensured.

- 3. END - no operands. If the Labels Stack is empty, issue an error message and terminate execution. Otherwise, free storage for any Labels List entries and pop the Labels Stack by one entry.
- 4. CALL - operand: type 11. If the operand is an indirect reference, then the reference is resolved to the actual operand. The resolved operand value is used as a search argument against the set of HEAD instruction labels in

the PLT. If no match is obtained, an error message is issued and execution is terminated. Otherwise, a new entry is pushed onto the Control Stack and the Control Stack Pointer is adjusted to reference this new top entry. The CALL/LATCH flag of the entry is set to indicate a CALL type invocation, and the Return point field value is set to the current value of the Instruction Pointer (IP). The entry point address of the procedure to be invoked, given by the field "Location of the HEAD instruction, minus one", replaces the contents of the IP.

5. LATCH - operand: type 11. If the operand is an indirect reference, then the reference is resolved to the actual operand. The resolved operand is used as the search argument against the set of HEAD instruction labels in the PLT. If no match occurs, an error message is issued and execution terminates. Otherwise, a new entry is pushed onto the Control Stack and the Control Stack Pointer is adjusted to reference this new top entry. The CALL/LATCH flag is set to indicate a LATCH type invocation, the Return point field value is set to the current value of the IP, and the current value of the Current In-

put String Pointer is entered into the third field of this entry. The entry point address of the procedure to be invoked, obtained from the PLT entry for that procedure, is copied to the IP.

6. EXIT - no operands. If either the Labels Stack or the Control Stack is empty, issue an error message and terminate execution. Otherwise, free the storage for any Labels List entries, pop the top entry off the Labels Stack, replace the IP value by the contents of the Return point field of the top Control Stack entry, and pop the Control Stack by one entry.
7. PROGEND - no operands. During the pre-execution scan, the PROGEND is used to delimit the M-TM program's set of instructions. During execution, it causes an error message to be issued and execution to be terminated if an attempt is made to execute it.
8. STOP - no operands. Execution of the program is terminated.
9. EQU - no operands. This is a "no operation" instruction. When an M-TM program is generated by a compiler, it is sometimes advantageous to insert labels using this instruction.

10. BT - operand: type 10. If the TF indicator is set to false, then set it to true. Otherwise, a branch is performed. The branch process is:
If the operand is an indirect reference, then it is resolved to the actual label value. The actual label value is used as the search argument against the set of instruction labels in the PLT entry referenced by the PLT pointer field of the top Labels Stack entry. If a match is not found, an error message is issued and execution is terminated. Otherwise, the IP value corresponding to the found label entry replaces the current contents of the IP.
11. BF - operand: type 10. If the TF indicator is set to true, then no action occurs. Otherwise, the TF indicator is set to true and the branch process is performed (see BT).
12. BER - no operands. If the TF indicator is set to true, then no action occurs. Else, one of two actions occur. If this operation occurs in an uncanceled LATCH (as indicated by the top Control Stack entry), then set the TF indicator to false, set the Current Input String Pointer

value to the value contained in the third field of the top Control Stack entry, and perform the actions specified by the EXIT instruction above. Else, output an error message and terminate execution of the program.

13. ERR - no operands. A message is transmitted to the output device indicating that an error condition has arisen at or about the position in the Current Input String indicated by the Current Input String Pointer value. Also, the string in the Output Buffer is copied and transmitted to the output device. The output Buffer Pointer is then reset to indicate an empty buffer condition.
14. CANCEL - no operands. The Control Stack entries are searched, starting from the top entry, up to the first encountered LATCH entry. The contents of the CALL/LATCH flag field are set to indicate a cancelled LATCH entry.
15. UNLATCH - no operands. The complete Control Stack is searched for LATCH entries. When found, the respective CALL/LATCH flag field contents are set to indicate a cancelled LATCH entry.
16. TRUE - no operands. Set the TF indicator to

true.

17. FALSE - no operands. Set the TF indicator to false.
18. BEOS - operand: type 10. If the EOS flag is not set, then no action occurs. Otherwise, the EOS flag is reset to false and the branch process is performed.
19. BNEOS - operand: type 10. If the EOS flag is set to true, then reset it to false. Otherwise, perform the branch process.
- 20a. Bc - operand: type 10. This is a conditional branch. The relation is between the value of the top entry on the Computation Stack and the value zero. If the Computation Stack is empty, then an error message is issued and program execution is terminated. The relations (conditions, or values of c) are:
 - LT - less than,
 - LE - less than or equal to,
 - EQ - equal to,
 - NE - not equal to,
 - GE - greater than or equal to, and
 - GT - greater than.

If the condition is met, the branch process is performed. Otherwise, there is no action.

- 20b. BcP - operand: type 10. The action is as for Bc, but the top Computation Stack entry is popped off after the test is made.
- 21. BCE - operands: type 4 and type 10. A test is performed between the contents of the Output Buffer and the contents of the area referenced by the first operand. If the strings are not equal, the Output Buffer Pointer is reset to indicate an empty buffer. Otherwise, the Output Buffer Pointer is reset to indicate an empty buffer, and the branch process is performed using the second operand value.
- 22. BCNE - operands: type 4 and type 10. A test is performed between the contents of the Output Buffer and the contents of the area referenced by the first operand. If the strings are equal, the Output Buffer Pointer is reset to indicate an empty buffer. Otherwise, the Output Buffer Pointer is reset to indicate an empty buffer and the branch process is performed using the second operand value.
- 23. BEOF - operand: type 10. If the EOF indicator is not set, then no action occurs. Otherwise, the EOF indicator is reset to "off" and the branch process is performed.

24. BNEOF - operand: type 10. If the EOF indicator is set, reset it. Otherwise, perform the branch process.
25. BMTY - operands: type 5 and type 10. If the STACK area referenced by the first operand is not empty, then no action occurs. Otherwise, the branch process is performed using the second operand value.
26. BNMTY - operands: type 5 and type 10. If the STACK area referenced by the first operand is empty, then no action occurs. Otherwise, the branch process is performed using the second operand value.
27. BCH - operands: type 6 and type 10. Here, as for BNCH, BNUM and BNUM to follow, the first operand references a VECTOR area, and the element referenced is indicated by the value of the top entry on the Computation Stack (the value must be greater than zero). If the Computation Stack is empty, an error message is issued and program execution terminated. Otherwise, the respective test is performed. After the test is performed, the top entry of the Computation Stack is popped. For BCH, the action is to perform the branch process, using

the value of the second operand, if the referenced element is of type STRING (a character string data element).

28. BNCH - operands: type 6 and type 10. Perform the branch process, using the second operand value, if the referenced element is not of type STRING.
29. BNUM - operands: type 6 and type 10. Perform the branch process, using the second operand value, if the referenced element is of type NUMERIC.
30. BNNUM - operands: type 6 and type 10. Perform the branch process if the referenced element is not of type NUMERIC.
31. B - operand: type 10. If necessary, the operand is resolved to its actual value. The branch process is then performed.
32. TO - no operands. The contents of the Output Buffer specify the search argument. The action is as follows:
 - i) If the EOS indicator is set to true, set the TF indicator to false, and end the instruction. Else, go to ii).
 - ii) Retain the current value of the Cur-

rent Input String Pointer.

- iii) Starting from the position indicated by the Current Input String Pointer, search sequentially to the right for the first occurrence of the search argument in the Current Input String.
- iv) If the end of the Current Input String is encountered before a successful match is found, set the TF indicator to false and copy the retained Current Input String Pointer value back to that area.
- v) If the search argument is longer than the remaining portion of the Current Input String (the current number of characters in the Current Input String is known), then set the TF indicator to false and restore the retained value of the Current Input String Pointer.
- vi) Otherwise, since a match has occurred, set the TF indicator to true, and set the value of the Current Input String Pointer to reference

the first character of the located string in the Current Input String.

- vii) In any outcome above, reset the Output Buffer Pointer value to indicate an empty buffer.

33. READ - no operands. The input data is assumed to consist of zero or more blocks of data. A block of data may be a single line from a time-sharing terminal, a deck of cards, or whatever the input device supports. The process is to read the next available block of data, as a character string, and copy it to the Current Input String. The previous contents of the Current Input String are destroyed. If the EOF indicator was set to true before the READ instruction was executed, no read is attempted. Also, if there are no more blocks of data available, the EOF indicator is set to true and no read is performed. In these last two cases, the Current Input String Pointer is set to indicate an empty Current Input String.

- b) Syntax - The set of syntax instructions tests for certain classes of terminal strings being the next item in the Current Input String. The test starts

from the character referenced by the Current Input String Pointer. If successful, the TF indicator is set to true, the item found is made the "current item", and the Current Input String Pointer is set to reference the character immediately to the right of the item (if in doing this the end of the Current Input String is passed, the EOS indicator is set to true). Otherwise, the TF indicator is set to false, and the Current Input String Pointer value is reset to its value before the test. The "current item" is the most recently found terminal string. Until replaced by the next terminal string to be found, it is available for use in the production of object text.

Except for the NEXT instruction, all of the syntax instructions skip over leading blanks and then test for the indicated terminal class being the next item. In the case of the NEXT instruction, leading blanks are not skipped. The instructions are:

1. ID - no operands. The test is for an identifier of the form:

$$\langle \text{IDENTIFIER} \rangle ::= \langle \text{FIRST} \rangle \mid \langle \text{IDENTIFIER} \rangle \langle \text{CHAR} \rangle$$

$$\langle \text{FIRST} \rangle ::= A \mid B \mid \dots \mid Z$$

$$\langle \text{CHAR} \rangle ::= \langle \text{FIRST} \rangle \mid 0 \mid 1 \mid \dots \mid 9 \mid _$$

2. UNINT - no operands. The test is for an unsigned decimal integer. The form is:

$\langle \text{UNSIGNED INTEGER} \rangle ::= \langle \text{DIGIT} \rangle \mid$
 $\langle \text{UNSIGNED INTEGER} \rangle \langle \text{DIGIT} \rangle$

$\langle \text{DIGIT} \rangle ::= 0 \mid 1 \mid \dots \mid 9$

3. INT - no operands. The test is for an optionally signed decimal integer. The form is:

$\langle \text{INTEGER} \rangle ::= +\langle \text{UNSIGNED INTEGER} \rangle \mid$
 $-\langle \text{UNSIGNED INTEGER} \rangle \mid$
 $\langle \text{UNSIGNED INTEGER} \rangle$

4. SR - no operands. The test is for a quoted string. The form is:

$\langle \text{QUOTED STRING} \rangle ::= ' \mid$
 $\langle \text{CHARACTER STRING} \rangle '$

$\langle \text{CHARACTER STRING} \rangle ::=$ a possibly null length string of characters (the only imposed restriction is that no single quote may be included)

The form "'''" is used to signify a string consisting of one single quote.

5. LETTER - no operands. The test is for one alphabetic character, as:

$\langle \text{LETTER} \rangle ::= A \mid B \mid \dots \mid Z$

6. DIGIT - no operands. The test is for one decimal digit, as:

$\langle \text{DIGIT} \rangle ::= 0 \mid 1 \mid \dots \mid 9$

7. TST - operand: type 3. The character literal

string is the terminal symbol to use in the test.

8. NEXT - the operand may take three forms. The first form is one of the set {A, Q, L, D, U}. The terminal class to be tested for is indicated as follows:

A - any single character,

Q - one single quote,

L - one alphabetic character (as for LETTER, above),

D - one decimal digit, or

U - one underscore (_).

The second form is a type 3 operand. In this case, the terminal to be tested for is the specified literal. The third case is a type 4 or type 6 operand. For this case, a question mark (?) precedes the operand to denote this form. The resolved character string value is used as for the second case. Note that for this case, subscripting information will be present as the top entry on the Computation Stack. If the stack is empty, an error message is issued and program execution is terminated. Otherwise, the Computation Stack has its top entry popped off after the reference is done.

c) Data Manipulation - These instructions allocate, deallocate, and manage data areas. In the case of copying data to a VECTOR area, if the index, as specified by the Computation Stack's top entry value, references an element that is already allocated, then deallocate the element. Whether the element was present or not, allocate a new element and set the pointer fields as required to chain it onto the VECTOR area. Then, set the index value as specified by the top Computation Stack entry and the data type field as required. The data is then copied to the data field of this new element. The data manipulation instructions are:

1. STA - operand: type 2 or type 6. If there are fewer than two entries on the Computation Stack, issue an error message and terminate program execution. The top entry of the Computation Stack is the data to be stored. The next value down is the subscripting information. If the reference is to a VECTOR area, then the element is created as outlined above, and the data value is copied to its data field. Otherwise, for a NUMERIC area, the data value is copied to the data area referenced. In both cases, the top two entries are popped off the Computation

Stack after the data is stored.

2. STB - operand: type 4 or type 6. If the Computation Stack is empty, issue an error message and terminate program execution. The top entry on the Computation Stack is the subscripting information. If this is a VECTOR reference, then create the element as outlined above. The contents of the Output Buffer are copied to the data field of this entry. For a STRING reference, the Output Buffer contents are copied to the data area. In both cases, the Computation Stack's top entry is popped off, and the Output Buffer Pointer is reset to indicate an empty buffer.
3. PUSH - operand: type 5. If the Computation Stack is not empty, the top entry is popped off (this is ignored subscripting information). If the Computation Stack is empty, an error message is issued and program execution terminates. The contents of the Output Buffer are pushed onto the STACK area referenced by the operand. The Output Buffer Pointer is reset to indicate an empty buffer.
4. SWAP - operand: type 5. If the Computation Stack is empty, issue an error message and ter-

minate program execution. Otherwise, pop the top entry from the Computation Stack. If the referenced STACK area has fewer than two entries present, then no action occurs. Otherwise, interchange the top two entries.

5. PART - operand: type 4. There are four other operand values present. Three are the top three Computation Stack entries, and the fourth is the current contents of the Output Buffer. If there are fewer than three entries on the Computation Stack, issue an error message and terminate program execution. Otherwise, the top Computation Stack entry indicates the substring length. The next entry indicates the starting character number (from the left) into the string in the Output Buffer. If the starting character number is to the right of the string currently in the Output Buffer, or if the substring length value is less than one, then a zero length string is taken as the result of the operation. Otherwise, the substring, indicated by the starting character number and consisting of the number of characters given by the substring length, is extracted from the string in the Output Buffer (if the requested

substring length is greater than the number of characters from the indicated starting position to the end of the string, only that substring starting at the starting character number and going to the end of the string is extracted). The third Computation Stack entry is the subscript value. The extracted substring is stored into the area referenced by the given operand.

6. ENTER - operands: type 7 (HASH area name),
 type 2 (I value),
 type 2 (J value), and
 type 2 (error code).

Note that for code production ease from the META-T Compiler, subscript values present on the Computation Stack are in reverse order, but are popped off and ignored. If any operands are indirect references, they are resolved to their actual operand values. There is one more operand present--the contents of the Output Buffer. The character string in the Output Buffer is taken as the entry argument. An attempt is made to enter the entry argument into the HASH area. If the entry argument was entered previously, the attempt to enter this occurrence is indicated. This condition is

termed a hit. If no hit occurs, then the new entry is made into the HASH area.

The error code is set to zero if no hit occurs; otherwise, it is set to one. Also, the I and J values are set to the location of the new entry (for no hit) or the previous entry (for a hit). The Output Buffer Pointer is reset to indicate an empty buffer, and the Computation Stack is popped by four entries.

7. FIND - operands: type 4 (contains a reference to the HASH area to be used),
type 4 (string area argument),
type 2 (I value),
type 2 (J value), and
type 2 (option code).

Subscripting information is, as for ENTER, kept in reverse order on the Computation Stack. If fewer than five entries are present on the Computation Stack, issue an error message and terminate program execution.

There are two methods of retrieving strings entered by the ENTER instruction. These are specified by the value of the option code. If the option code is negative, then the (I,J) ele-

ment of the HASH area referenced by the first operand is copied to the string area argument. If the value is not present, the option code is set to one and the string area argument is set to a null length. Otherwise, the option code is set to zero, and the data is copied to the string area argument.

The other retrieval method is requested by having a non-negative option code value at the invocation of the instruction. In this case, the string area argument is used as the search argument into the referenced HASH area. If the search is successful, the (I,J) pair is stored and the option code is set to zero. If the search is not successful and the Predecessor field of the referenced HASH area does not specify an allocated HASH area (other than the current one), then the option code is set to one. However, if a currently activated HASH area is specified, then the contents of the Predecessor field for the current HASH area are copied to the STRING area referenced by the first operand and the FIND instruction is re-executed. In this manner, a chained HASH area search is performed automatically.

8. SLINK - operands: one of S - successor field,
 P - predecessor
 field, or
 F - flag field; and
 type 7 (HASH reference).

If the HASH reference is indirect, then it is resolved to the actual operand value (subscripting information must be present, but it is not used). The contents of the Output Buffer are copied to the specified field of the referenced HASH area. The top entry of the Computation Stack is popped off, and the Output Buffer Pointer is reset to indicate an empty buffer after the data is moved.

9. GLINK - operands: one of S - successor field,
 P - predecessor
 field, or
 F - flag field;
 type 7 (HASH reference); and
 type 4 (string receiving
 field).

The type 4 and type 7 operands are resolved, if indirect, to their actual references. If the HASH reference is to a non-existent area, an error message is issued and program execution

is terminated. Otherwise, the contents of the specified field of the indicated HASH area are copied to the string receiving field. Then, the top Computation Stack entry is popped off.

10. RESCAN - no operands. If the Current Input String and the New Input String are both empty, then no action occurs. Otherwise, if the New Input String is not empty, its contents replace the contents of the Current Input String, the Current Input String Pointer is set to the first character, and the New Input String Pointer is set to indicate an empty string. However, if the New Input String is empty, the Current Input String Pointer is merely reset to the beginning of the current contents of the Current Input String. In each case, the EOS indicator is set to false.

11. RESET - no operands. The Output Buffer Pointer is reset to indicate an empty buffer.

In the following arithmetic instructions, if the required number of entries are not present on the Computation Stack, an error message is issued and program execution is terminated. In the binary operations, let the top Computation Stack entry be called b, and the next entry be a. Then, the operation is:

a op b

for op being the binary operation to be performed.

12. NEG - no operands. The top Computation Stack entry has its value changed in sign.
13. ABS - no operands. The absolute value of the top Computation Stack entry replaces this entry.
14. ADD - no operands. The top two Computation Stack entries are popped off and their sum is pushed onto the Computation Stack.
15. SUB - no operands. The top two Computation Stack entries are popped off and their difference is pushed onto the Computation Stack.
16. MPY - no operands. The top two Computation Stack entries are popped off and their product is pushed onto the Computation Stack.
17. DIV - no operands. The top two Computation Stack entries are popped off. If the top entry is not equal to zero, then the integral value of the quotient of the two entries is pushed onto the Computation Stack. Otherwise, an error message is issued and program execution is terminated.
18. REM - no operands. The top two Computation Stack entries are popped off. If the top entry

is equal to zero, then an error message is issued and program execution is terminated.

Otherwise, the remainder after dividing the value of the top entry into the value of the next entry is pushed onto the Computation Stack.

19. ACNVT - no operands. If the contents of the Output Buffer are in the character form of an optionally signed, decimal integer, then the value is converted to numeric form, pushed onto the Computation Stack, and the Output Buffer Pointer is reset to indicate an empty buffer. Otherwise, the value zero is pushed onto the Computation Stack, and the Output Buffer Pointer is reset to indicate an empty buffer.
20. LEN - no operands. The length attribute of the current contents of the Output Buffer is pushed onto the Computation Stack, and the Output Buffer Pointer is reset to indicate an empty buffer.
21. DIM - operand: type 6. If the referenced VECTOR area contains no elements, then the value zero is pushed onto the Computation Stack. Otherwise, the maximum index value used is pushed onto the Computation Stack. Before the value is pushed onto the Computation Stack, the

stack has the top entry popped off to remove the operand's subscripting information.

22. LD - operand: type 8. If the operand is a literal, the value is pushed onto the Computation Stack. Otherwise, the operand is resolved to the actual operand, the subscript information is popped off the Computation Stack, and the referenced value is pushed onto the Computation Stack.
23. POP - operand: type 5. If the referenced STACK area is empty, then no action results. Otherwise, starting at the current character position in the Output Buffer, the contents of the top entry of the referenced STACK area are copied to the Output Buffer. This entry is then popped from the stack, the Output Buffer Pointer is adjusted to reflect the new length, and the Computation Stack is popped by one entry.
24. CCNVT - no operands. If the Computation Stack is empty, a zero value is assumed. Otherwise, the value of the top Computation Stack entry is used. This value is converted to character form and copied to the Output Buffer starting at the position indicated by the Output Buffer Pointer. If not empty, the Computation Stack's top entry

is popped off. Also, the Output Buffer Pointer is adjusted to reflect the new length.

25. LDB - operand: type 9. If the operand is a literal, then this value is copied to the Output Buffer starting at the position indicated by the Output Buffer Pointer. The Output Buffer Pointer is then adjusted to reflect the new length. Otherwise, the actual operand value is obtained and the referenced string is copied to the Output Buffer and the Output Buffer Pointer is adjusted. In this latter case, the Computation Stack has its top entry popped off.

26. Dc - no operands. The set of values for c are:
 - N - NUMERIC,
 - C - STRING,
 - V - VECTOR,
 - S - STACK, and
 - H - HASH.

If the Output Buffer is empty, then there is no action. Otherwise, the contents in the Output Buffer constitute a name to be allocated. This new name is used as a search argument against the name table contents for the indicated type. If the name is present, no allocation is made. Otherwise, a new entry is created with this new

name in the Name field, and the rest of the data area is allocated and initialized as specified earlier. Finally, the Output Buffer Pointer is reset to indicate an empty buffer.

27. Fc - no operands. The set of values for c is the same as for Dc, above. If the Output Buffer is empty, then no action results. Otherwise, the string contained there is used as a name. If this name is present as one of the entries of the name table indicated by the given type, all storage for this name is released and the name table entry is deallocated. If the name is not present, then no action results. Finally, the Output Buffer Pointer is reset to indicate an empty buffer.

d) Post-Syntactic

1. OUT - no operands. The contents of the Output Buffer are transmitted to the output device, and the Output Buffer Pointer is reset to indicate an empty buffer.
2. NEWIN - no operands. The contents of the Output Buffer are copied to the New Input String starting at the location specified by the New Input String Pointer. The New Input String Pointer is adjusted to reflect the inclusion of the Output

Buffer string. The Output Buffer Pointer is reset to indicate an empty buffer.

3. LDC - no operands. The current item is copied to the Output Buffer starting at the position specified by the Output Buffer Pointer. This pointer is then updated to reflect the inclusion of the current item.
4. GN - operand: an unsigned integer (this is an exception to the set of operand types). Sometimes it becomes necessary to generate labelling strings during the production of object text. The GN instruction is used to produce such strings. The form for the strings is:

```

<LABEL STRING> ::= #<DIGIT STRING>
<DIGIT STRING> ::= <DIGIT> |
                  <DIGIT STRING> <DIGIT>
<DIGIT> ::= 0 | 1 | ... | 9

```

Thus, one may produce acceptable M-TM instruction labels. Within an activation of a procedure, the strings are constant. Over the program's execution, they are globally unique. As many different strings are available in a procedure as required, since the operand is unrestricted in magnitude. Within an activation of a procedure, a list is maintained, the Labels List, which has entries of the form:

Pointer to the next element
Index
Value

The index is the value of the operand used to create this entry. The value is generated by converting the current Next Label Counter value to character form and preceding this by the "#" character. Then, the Next Label Counter is incremented by one.

When a GN instruction is executed, the list for this procedure is searched for an entry having an index value identical to the given operand. If the search is successful, then the corresponding value is copied to the Output Buffer starting at the point indicated by the Output Buffer Pointer. The pointer is then adjusted to reflect the inclusion of this value. If the search is not successful, then a new entry is chained onto the end of the Labels List with an index value indicated by the operand, and a new label value is generated as described. This new label value is copied to the value field of the new entry and to the Output Buffer starting at the point indicated by the Output Buffer Pointer.

The pointer is then adjusted to reflect this new inclusion.

We will end with two examples. The first example is a common one. It is a (simplified) parser for arithmetic assignment statements. The productions describing the assignment statement format are given using BNF. The productions are:

```

<ARITH> ::= <ID = <EXPR>

<ID> ::= M-TM identifier

<EXPR> ::= -<TERM> | <TERM> | <EXPR> + <TERM> |
          <EXPR> - <TERM>

<TERM> ::= <ID> | <INT>

<INT> ::= optionally signed decimal integer.

```

A corresponding M-TM program is:

```

RUN : HEAD ;
    L1 : READ ;
        : BEOS L2 ;
        : CALL ARITH ;
        : BT L1 ;
        : LDB =15.IMPROPER FORMAT ;
        : OUT ;
        : B L1 ;
    L2 : STOP ;
        : END ;
ARITH : HEAD ;
        : ID ;
        : BT L1 ;      L2: FALSE ;      : EXIT ;
    L1 : TST 1.= ;
        : BF L2 ;
        : CALL EXPR ;
        : EXIT ;
        : END ;
EXPR  : HEAD ;
        : TST 1.- ;
        : CALL TERM ;
        : BF L3 ;
    L1 : TST 1.+ ;

```

```

      : BT L2 ;
      : TST 1.- ;
      : BF L4 ;
L2 : CALL TERM ;      : BT L1 ;
L3 : FALSE ;      L4 : EXIT ;
      : END ;
TERM : HEAD ;
      : ID ;      : BT L1 ;
      : INT ;
L1 : EXIT ;
      : END ;
      : PROGEND ;

```

It should be noted from the above example that the instructions are written in a free format.

The next example reads a string of characters (a block) and counts the number of alphabetic characters that are present. After exhausting the string, the character counts are printed. Another process is demonstrated. When the CHECK procedure completes, it passes control directly to the OUTPUT procedure. The OUTPUT procedure prints the counts and then returns control to the START procedure. This is possible even though it was the CHECK procedure which was invoked by START. The M-TM program is:

```

START : HEAD ;
      : CALL FILL_ER_UP ;      : TRUE ;
      : CALL CHECK ;
      : STOP ;
      : END ;
FILL_ER_UP : HEAD ;
      : LDB =26.ABCDEFGHIJKLMNOPQRSTUVWXYZ ;
      : NEWIN ;      : RESCAN ;
      : LDB =1.I ;
      : DN ;
      : LDB =7.LETTERS ;
      : DV ;
      : LD =1.0 ;
      : LD =1.1 ;
      : STA I ;

```

```

L1 : LD =1.0 ;
    : LD I ;
    : LD =2.26 ;
    : SUB ;
    : BGTP L2 ;
    : LD =1.0 ;
    : LD I ;
    : NEXT A ;
    : LDC ;
    : STB LETTERS ;
    : LD =1.0 ;
    : LD =1.0 ;
    : LD I ;
    : LD =1.1 ;
    : ADD ;
    : STA I ;
    : B L1 ;
L2 : LDB =5.COUNT ;
    : DC ;
    : READ ;      : EXIT ;
    : END ;
INDEX : HEAD ;
      : LD =1.0 ;
      : LD =1.1 ;
      : STA I ;
L1 : LD =1.0 ;
    : LD I ;
    : LD =2.26 ;
    : SUB ;
    : BGTP L2 ;
    : LD =1.0 ;
    : LD I ;
    : NEXT ?LETTERS ;
    : BT L3 ;
    : LD =1.0 ;      : LD =1.0 ;
    : LD I ;      : LD =1.1 ;      : ADD ;
    : STA I ;
    : B L1 ;
L2 : FALSE ;
L3 : EXIT ;
    : END ;
CHECK : HEAD ;
L1 : BEOS L2 ;
    : CALL INDEX ;
    : BF L1 ;
    : LD =1.0 ;      : LD =1.0 ;      : LD I ;
    : LD I ;
    : LD COUNT ;      : LD =1.1 ;      : ADD ;
    : STA COUNT ;
    : B L1 ;

```



```

      L2 : END ;
OUTPUT : HEAD ;
      : LDB =15.THE COUNTS ARE: ;
      : OUT ;
      : LD =1.0 ;      : LD =1.1 ;
      : STA I ;
L1 : LD =1.0 ;
      : LD I ;      : LD =2.26 ;      : SUB ;
      : BGTP L2 ;
      : LD =1.0 ;      : LD I ;      : LD COUNT ;
      : CCNVT ;      : OUT ;
      : LD =1.0 ;      : LD =1.1 ;      : LD =1.0 ;
      : LD I ;
      : ADD ;      : STA I ;
      : B L1 ;
L2 : EXIT ;
      : END ;
      : PROGEND ;

```

From the above examples it should be obvious that to write programs directly in M-TM code is a tedious task. M-TM is designed to serve as a versatile machine upon which to base mechanical translators, and not for ease of direct use in programming. Thus, judgment as to the worth of M-TM should not be based on the examples, but rather on its ability to serve its intended purpose.

THE META-T PROGRAMMING LANGUAGE

After trying to generate a translator using just the M-TM instructions directly, one begins to feel that there must be something better, something on a higher level. What is needed is to create a metalanguage that at least partially mirrors the power of M-TM, but does so with a syntax that is both easier to use and not as verbose. This need was the force behind the META-T Programming Language (MTPL).

A metalanguage used for the more convenient expression of transformations may take the form of a decision table, an algorithmic language, a macro language, or a BNF derivative. META-PI's metalanguage falls into this last category. Since the author became familiar with compiling through the vehicles of META-II and META-PI and is comfortable with the syntax of these translator writing systems, the syntax of MTPL will also be a BNF derivative.

Strong parallels exist between the META-PI metalanguage format and that of MTPL. A production in META-PI is in the form

$$l := e;$$

whereas in MTPL the form of a production is

$$l = e;$$

A META-PI "program", or set of productions, merely lists all the productions and assumes that the initial production is to

be executed first. MTPL uses this same philosophy, except that the set of productions (statements) comprising a program are delimited by a keyword (.END).

As a comparison between BNF, META-PI and MTPL, consider the following sets of productions:

BNF:

```

<EXPR> ::= -<TERM> | +<TERM> | <TERM> |
          <EXPR> + <TERM> | <EXPR> - <TERM>
<TERM>  ::= <FACTOR> | <TERM> * <FACTOR> |
          <TERM> / <FACTOR>
<FACTOR> ::= <IDENTIFIER> | <INTEGER> | (<EXPR>)

```

META-PI:

```

EXPR := ('-'TERM | ('+' | .EMPTY) TERM)
       $('+'TERM | '-'TERM);
TERM := FACTOR $('*'FACTOR | '/'FACTOR);
FACTOR := .ID | .INT | '(' EXPR ')';

```

MTPL:

```

EXPR = ('-'TERM | ('+' | .TRUE) TERM)
       $('+'TERM | '-'TERM);
TERM = FACTOR $('*'FACTOR | '/'FACTOR);
FACTOR = .ID | .INT | '(' EXPR ')';
.END;

```

On the metalanguage level, the differences between META-PI and MTPL are not very great.

A production in MTPL must not have occurrences of left recursion, as:

```
A = A...;
```

This production will never terminate, because of recursive calls to the production A. Also, the use of:

```

A = B...;
B = A;

```

does not eliminate the problem, as the production A attempts to produce a B, which attempts to produce an A, etc. Although the above examples are trivially obvious, their essence is important in generating productions. Both to make productions more compact and to reduce or eliminate the need for backup, all productions should be factored. Thus, the production

$$A = B C \mid B D ;$$

becomes

$$A = B (C \mid D) ;$$

The factored form is more compact and exhibits the meaning of the production in a clearer manner. Also, the unfactored form would require use of the backup facility of MTPL for proper operation, since, if a parse were to require the alternative "B D", the alternative "B C" would be used first. Failure in attempting to recognize an occurrence of a "C" would cause failure of the production, even though the parse would have been successful using the "B D" alternative.

Certain data areas are needed in the execution of a translator. Some of the data areas are managed for the user. These data areas are those of the TF indicator, the EOS flag, and so on, mentioned in the last chapter. Many are directly reflected in MTPL.

Guiding of the execution of a production is done using the TF indicator; alternatives are chosen by the success or

failure of preceding alternatives. Functions exist that allow the direct setting of the TF indicator, and some functions affect the value of the TF indicator by the result of a syntax test.

The form of an MTPL production was given as:

$$l=e;$$

The left part, l , is the name of the production (read "sub-routine") and constitutes a non-terminal symbol. The right part, e , is an expression (also termed a meta-expression) of the form:

$$e_1 \mid \dots \mid e_n \quad (n > 0)$$

where the e_i terms are alternative expressions and the character " $\mid$ " is used to separate the alternatives. Each term consists of primitives, terminals and non-terminals, just as for BNF or META-PI. The terminal symbols are character literals; the non-terminals are calls upon other productions; and the primitives are calls upon the built-in functions of MTPL.

The listing of primitives will be done, as much as is possible, by relating a primitive from MTPL to its equivalent code in terms of M-TM instructions. Thus, for the primitive that tests for the occurrence of an identifier we have:

.ID maps to ID.

Some primitives do not require operands, and some do. Some primitives that require operands allow lists of operands (where each list entry is separated from its neighbor by

a comma, and each list element must separately satisfy the type requirements of the primitive. In MTPL, a name is also termed a reference, or REF, and is of the form (in BNF):

$$\langle \text{REF} \rangle ::= \langle \text{IDENTIFIER} \rangle (\langle \text{EXPRESSION} \rangle) \mid \langle \text{IDENTIFIER} \rangle$$

The non-terminal $\langle \text{IDENTIFIER} \rangle$ in the above production, is of the form required by the ID instruction from M-TM, and the non-terminal $\langle \text{EXPRESSION} \rangle$ is an arithmetic expression (see the first example and the REF production for further details).

The (data) type of the identifier recognized by $\langle \text{IDENTIFIER} \rangle$ will depend upon the primitive in use. A type error occurs if the required type of the identifier is not met (i.e., an arithmetic type used where a character type is required). In most cases, the required type will be obvious from the mapping. Where there may be confusion, a more complete explanation will be given. (Note: the types of operands will be referenced using the numeric codes from the last chapter, which were:

- 1 - integer literal,
- 2 - direct or indirect reference to a NUMERIC area,
- 3 - character literal,
- 4 - direct or indirect reference to a STRING area,
- 5 - direct or indirect reference to a STACK area,
- 6 - direct or indirect reference to a VECTOR area,

- 7 - direct or indirect reference to a HASH area,
- 8 - type 1 or 2,
- 9 - type 3 or 4,
- 10 - direct or indirect reference to an instruction label, and
- 11 - direct or indirect reference to a HEAD instruction label (the name of a production).

The syntax primitives are:

Primitive	M-TM Equivalent
.ID	: ID ;
.INT	: INT ;
.UNINT	: UNINT ;
.SR	: SR ;
.NEXT(a) where a is: one of {A,Q,D,L,U},	: NEXT a ;
or, a is a literal, i.e., a string of characters enclosed in single quotes, or three single quotes (giving a string of one single quote),	: NEXT 1.a ; where 1 is the length of the literal (not including enclosing quotes) and a is the literal (not including quotes)
or, a is a reference ?REF where the REF is a type 4 or type 6 reference	: NEXT ?ref ;
.LETTER	: LETTER ;
.DIGIT	: DIGIT ;
.TRUE	: TRUE ;
.FALSE	: FALSE ;

Primitive

M-TM Equivalent

.DCL(arg)
 The argument (arg, above) is in the form:
 <ARG>::=<ARG>,<ENTRY>|
 <ENTRY>
 <ENTRY>::=<TYPE>(<one
 or more character
 expressions in a
 list form>)
 <TYPE>::=.NUMERIC |
 .STRING |
 .VECTOR |
 .STACK |
 .HASH

.FREE(arg)
 The argument (arg, above) is in the form as for .DCL(...), above.

.LATCH(r)
 where r is a type 11 operand

.CANCEL

.UNLATCH

.EOF(meta-expression)
 where a meta-expression is the e term of a production

**.TYPE(REF,meta-expression_1,
 meta-expression_2,
 meta-expression_3)**
 The REF is to a VECTOR area and may be indirect.

For each ENTRY, the action is to create the code:
 text to evaluate the character expression

 : Dc ;

Here, the character expression is from the list occurring after the TYPE in the ENTRY production. And, the c is one of:

 N - .NUMERIC,
 C - .STRING,
 V - .VECTOR,
 S - .STACK, or
 H - .HASH,

as determined by the TYPE production.

As for .DCL(...), above, except that

 : Dc ;

is replaced by

 : Fc ;

 : LATCH r ;

 : CANCEL ;

 : UNLATCH ;

 : BNEOF label ;

text for the meta-expression

label : EQU ;

 : BCH ref,label_1 ;

 : BNUM ref,label_2 ;

text for meta-expression_1

 : B label_3 ;

label_1 : EQU ;

text for meta-expression_2

 : B label_3 ;

label_2 : EQU ;

text for meta-expression_3

label_3 : EQU ;

Primitive

M-TM Equivalent

**.EMPTY(REF,meta-expression_1,
meta-expression_2)**
The REF is to a STACK area.
Any available subscripting
is ignored.

```
: BNMTY ref,label_1 ;
text for meta-expression_1
: B label_2 ;
label_1 : EQU ;
text for meta-expression_2
label_2 : EQU ;
```

.SET(REF,expression)
If the expression is
arithmetic, then the
REF must be to a NUMERIC
area (or an element of
a VECTOR, in which case
the referenced element
will be made to be of
numeric type).

```
text to evaluate the arith-
metic expression
: STA ref ;
```

If the expression is
character, then the
REF must be to a char-
acter area (STRING or
a STRING element of a
VECTOR area).

```
text to evaluate the charac-
ter expression
: STB ref ;
```

**.PUSH(REF,character
expression)**

```
text to evaluate the charac-
ter expression
: PUSH ref ;
```

**.SWAP(REF) or
.SWAP(list of REF's)**

```
For each list entry:
: SWAP ref ;
```

.READ

```
: READ ;
```

**.PART(REF,arex1,arex2,
character expression)**
where arex means
arithmetic expression

```
text to evaluate the charac-
ter and arithmetic
expressions
: PART ref ;
```

**.ENTER(REF1,character
expression,REF2,
REF3,REF4)**

```
text to evaluate the charac-
ter expression
: ENTER ref1,ref2,ref3,ref4 ;
```

**.FIND(REF1,REF2,REF3,
REF4,REF5)**

```
: FIND ref1,ref2,ref3,ref4,
ref5 ;
```

Primitive	M-TM Equivalent
.SLINKc(REF,character expression) where c is one of: S - successor link, P - predecessor link, or F - flag field	text to evaluate the character expression : SLINK c,ref ;
.GLINKc(REF1,REF2) where c is one of: S - successor link, P - predecessor link, or F - flag field	: GLINK c,ref1,ref2 ;

The control functions are:

Primitive	M-TM Equivalent
.EOS(meta-expression)	: BNEOS label ; text for meta-expression label : EQU ;
.RESCAN	: RESCAN ;
.AIF(arex,meta-expression1, meta-expression2, meta-expression3)	text to evaluate the arithmetic expression (arex) : BGE label_1 ; text for meta-expression1 : B label_3 ; label_1 : BNEP label_2 ; text for meta-expression2 : B label_3 ; label_2 : EQU ; text for meta-expression3 label_3 : EQU ;
.CIF(character expression, REF,meta-expression1, meta-expression2)	text to evaluate the character expression : BCNE ref,label_1 ; text for meta-expression1 : B label_2 ; label_1 : EQU ; text for meta-expression2 label_2 : EQU ;

Primitive	M-TM Equivalent
.STOP	: STOP ;
.EXIT	: EXIT ;
.EXITT	: TRUE ; : EXIT ;
.EXITF	: FALSE ; : EXIT ;
.TO(character expression)	text to evaluate the character expression : TO ;

The miscellaneous primitives are:

Primitive	M-TM Equivalent
.OUT(list of character expressions)	For each character expression in the argument list: text to evaluate the character expression : OUT ;
.NEWIN(list of character expressions)	For each character expression in the argument list: text to evaluate the character expression : NEWIN ;
.IGN(list of character expressions)	For each character expression in the argument list: text to evaluate the character expression : RESET ;
.REM(arex1, arex2)	text to evaluate the arithmetic expressions : REM ;
.ACNVT(character expression)	text to evaluate the character expression : ACNVT ;

Primitive	M-TM Equivalent
.LENGTH(character expression)	text to evaluate the character expression : LEN ;
.DIM(REF)	: DIM ref ;
.POP(REF)	: POP ref ;
.CCNVT(arex)	text to evaluate the arithmetic expression : CCNVT ;
.ERR(...) There are four forms:	
.ERR()	: BT label_1 ; label_1 : EQU ;
.ERR(character expression)	: BT label_1 ; text to evaluate the character expression : ERR ; label_1 : EQU ;
.ERR(,meta-expression)	: BT label_1 ; text for the meta-expression label_1 : EQU ;
.ERR(character expression, meta-expression)	: BT label_1 ; text to evaluate the character expression : ERR ; text for the meta-expression label_1 : EQU ;

To illustrate the versatility and use of MTPL we will present some examples. Before proceeding with the examples, however, some ideas must be given for those not familiar with META-PI or META-II. The most important point is that the operator "\$", to be seen in the examples, is used to specify

zero or more replications of the term immediately following it. Thus,

```
A = 'B' '$' 'C' 'D';
```

specifies strings such as 'BD', 'BCD', 'BCCD', etc.; and

```
AA = 'B' $('C' 'D') 'E';
```

specifies strings such as 'BE', 'BCDE', 'BCDCDE', etc. The parentheses are used for grouping, as seen in the latter production. The flow of control in MTPL begins with the first production. This first production determines which other productions are to be used.

We now turn to the first example, which is the syntax of MTPL expressed using MTPL. The productions are:

```
RUN = .DCL(.NUMERIC('LC','END'), .STRING('C1','C2','REF'),
  .STACK('REFS')) .SET(END,0) .READ $COMMENT META_STMT ';'
  $.AIF(END, .EXITF, RUN_STMT .TRUE,
    .OUT('; END ;') .FALSE);
```

```
RUN_STMT = COMMENT | .LATCH(META_STMT) ';' |
  '.END' (';' | .TRUE) .SET(END,1) | OUT_SKIP;
```

```
COMMENT = '#c' .TO('c#') 'c#';
```

```
OUT_SKIP = .SET(LC,0) $(.AIF(LC-1, .NEXT(Q) .SET(LC,1) |
  .NEXT(';') .EXITT | .NEXT(A),
  .NEXT(Q) .SET(LC,3) | .NEXT(A) .SET(LC,2),
  .AIF(LC-2, .STOP, .NEXT(Q) .SET(LC,0) | .NEXT(A),
  .NEXT(';') .EXITT | .NEXT(A) .SET(LC,0))));
```

```
META_STMT = .ID '=' META_EXPR;
```

```
META_EXPR = META_TERM $('|' META_TERM);
```

```
META_TERM = $COMMENT META_FACTOR $(COMMENT | META_FACTOR);
```

```
META_FACTOR = META_PRIMARY (ERROR | TRUE);
```

```
ERROR = '.ERR' '(' (CITEM | .TRUE) (',' META_EXPR | .TRUE)
  ')';
```

```
META_PRIMARY = META_FUNCTION | .ID | '@' .ID | .SR |
                '(' META_EXPR ')' | '$' META_EXPR;
```

```
META_FUNCTION = PRIMITIVE_FUNCTION | DATA_FUNCTION |
                CONTROL_FUNCTION | MISC_FUNCTION;
```

```
PRIMITIVE_FUNCTION = '.ID' | '.SR' | '.UNINT' | '.INT' |
                    '.LETTER' | '.DIGIT' | '.TRUE' | '.FALSE' |
                    '.NEXT' '(' ('A' | 'Q' | 'L' | 'D' | 'U') | '.SR' |
                    '? REF ')';
```

```
DATA_FUNCTION = '.DCL' '(' DCLENTRY $(',' DCLENTRY) ')' |
                '.FREE' '(' FREETYPE $(',' FREETYPE) ')' |
                '.LATCH' '(' (.ID | '@' .ID) ')' | '.CANCEL' |
                '.UNLATCH' | '.EOF' '(' META_EXPR ')' |
                '.TYPE' '(' REF ', ' META_EXPR ', ' META_EXPR ', ' META_EXPR
                ')', |
                '.EMPTY' '(' REF ', ' META_EXPR ', ' META_EXPR ')' |
                '.SET' '(' REF ', ' (CITEM | AREX) ')' |
                '.PUSH' '(' REF ', ' CITEM ')' |
                '.SWAP' '(' REF $(',' REF) ')' |
                '.READ' | '.PART' '(' REF ', ' AREX ', ' AREX ', ' CITEM
                ')', |
                '.ENTER' '(' REF ', ' CITEM ', ' REF ', ' REF ', ' REF ')' |
                '.FIND' '(' REF ', ' REF ', ' REF ', ' REF ', ' REF ')' |
                '.SLINK' ('S' | 'P' | 'F') '(' REF ', ' CITEM ')' |
                '.GLINK' ('S' | 'P' | 'F') '(' REF ', ' REF ')';
```

```
CONTROL_FUNCTION = '.EOS' '(' META_EXPR ')' | '.RESCAN' |
                  '.AIF' '(' AREX ', ' META_EXPR ', ' META_EXPR ', '
                  META_EXPR ')' |
                  '.CIF' '(' CITEM ', ' REF ', ' META_EXPR ', ' META_EXPR
                  ')', |
                  '.STOP' | '.EXIT' ('T' | 'F' | .TRUE) |
                  '.TO' '(' CITEM ')';
```

```
MISC_FUNCTION = '.OUT' '(' CITEM $(',' CITEM) ')' |
                '.NEWIN' '(' CITEM $(',' CITEM) ')' |
                '.IGN' '(' CITEM $(',' CITEM) ')';
```

```
AREX = ('-' TERM | ('+' | .TRUE) TERM) $('+' TERM | '-' TERM);
```

```
TERM = FACTOR $('*' FACTOR | '/' FACTOR);
```

```
FACTOR = '.REM' '(' AREX ', ' AREX ')' |
          '.ACNVT' '(' CITEM ')' | '.LENGTH' '(' CITEM ')' |
          '.DIM' '(' REF ')' | REF | '(' AREX ')' |
          '| AREX' | '.INT';
```

```
REF = REFTERM | '@' REFTERM;
```

```

REFTERM = .ID ((' AREX ') | .TRUE);
CITEM = $(' .SWAP' ((' REF $(' , ' REF ) ') |
    '*' (.UNINT | .TRUE) | ' .POP' ((' REF ') |
    ' .CCNVT' ((' AREX ') | REF | .SR;
DCLENTY = TYPE ((' CITEM $(' , ' CITEM ) ') ;
FREETYPE = TYPE ((' CITEM $(' , ' CITEM ) ') ;
TYPE = ' .NUMERIC' | ' .STRING' | ' .VECTOR' | ' .STACK' |
    ' .HASH';
.END;

```

In the CITEM production, the part:

```

    '*' (.UNINT | .TRUE)

```

may cause confusion. If this production is used to recognize the terminal "*", then this is interpreted as a reference to the current item. If, however, one has a terminal of the form "*i", where i is an unsigned integer, one has encountered a reference to the ith internally generated labelling string.

A labelling string is a non-null string of characters from the set {#, 0, 1, ..., 9}. Each new occurrence of an "*i" is unique within a program execution. And, within an activation of a production, an "*i" is constant. As many different i's may be generated as needed.

To illustrate the situation consider the following productions:

```

A = B .OUT(*1) $(C .OUT(*2)) .OUT(*1);
B = C .OUT(*1 '_' *2);
C = .OUT(*1 '.' *2);

```

.END;

Although these are absurd productions, they demonstrate the action quite well. In the A production, two separate, globally unique labels exist. They are the ones generated by the first occurrence of *1 and *2, respectively. Each use, in A, of *1 or *2, thereafter, references these strings.

In the B production, the occurrences of *1 and *2 create completely new unique strings, different from the *1 and *2 in the A or C productions. Each invocation of the C production generates new, globally unique strings for *1 and *2.

The former productions constitute the reference for the syntax of MTPL. The semantics of MTPL is specified partially by the correspondence between primitives and their M-TM equivalents. A compiler for MTPL is included in the next chapter and gives the complete correspondence.

The task performed by an interpreter for postfix expressions is a translation task that maps the postfix string into a member of the resultant text, be this numbers, strings, or whatever. The next example demonstrates the basic technique in producing an interpreter for postfix strings that generate a number as a result. The postfix string will take the form of:

```
INPUT = STRING ';' $(STRING ';' ) ;
STRING = TERM $(', ' TERM);
TERM = OPERAND | OPERATOR;
```



```

OPERAND = .INT;

OPERATOR = '+' | '-' | '*' | '/';

.END;

```

The semicolon serves to delimit each single string. Thus, the input string consists of as many postfix expressions as desired, but each is separated from its neighbor by a semicolon.

The productions are:

```

RUN = .READ .DCL(.NUMERIC('SP'), .VECTOR('STK')) .SET(SP,0)
      $(RUNS .OUT(.CCNVT(STK(1))) | .FALSE);

RUNS = TERM $(',' TERM) ';' ;

TERM = OPERAND | OPERATOR;

OPERAND = .INT .SET(SP,SP+1) .SET(STK(SP), .ACNVT(*));

OPERATOR = ('+' ADD | '-' SUB | '*' MPY | '/' DIV) .TRUE;

ADD = TWO_OPNDS .SET(STK(SP), STK(SP)+STK(SP+1));

SUB = TWO_OPNDS .SET(STK(SP), STK(SP)-STK(SP+1));

MPY = TWO_OPNDS .SET(STK(SP), STK(SP)*STK(SP+1));

DIV = TWO_OPNDS .AIF(STK(SP+1), .TRUE, .EXIT, .TRUE)
      .SET(STK(SP), STK(SP)/STK(SP+1));

TWO_OPNDS = .AIF(SP-2, .OUT('**INVALID STACK CONTENTS**')
      .EXITF, .SET(SP,SP-1) .EXITT, .SET(SP,SP-1) .EXITT);

.END;

```

A partial trace of the postfix string

1, 2, 3, +, 4, *, +;

(or $1+4*(2+3)$) is:

Production	Current Item	STACK Contents at the Start of Execution of this Production
RUN	-	-
RUNS	-	-
TERM	1	-
RUNS	,	1
TERM	2	1
RUNS	,	1 2
TERM	3	1 2
RUNS	,	1 2 3
TERM	+	1 2 3
RUNS	,	1 5
TERM	4	1 5
RUNS	,	1 5 4
TERM	*	1 5 4
RUNS	,	1 20
TERM	+	1 20
RUNS	;	21

In the above, only the main production names have been noted.

The next example is rather significant, as it demonstrates the manner in which locally optimized code may be generated for arithmetic expressions. The object code to be produced is IBM 360 assembler text. The arithmetic is to be done in floating point, double precision. The four floating point registers (0, 2, 4, 6) will be used as the top four

cells of a stack. Also, we will use a section of memory as the rest of the stack.

The optimization to be done is somewhat limited, but will replace the code:

```
LD      0,A
LD      2,B
ADR     0,2
```

with:

```
LD      0,A
AD      0,B
```

for the expression $A+B$.

To be able to understand the code that will be produced, one must be somewhat familiar with the IBM 360's instruction format. We will use the two definitions:

addr - An address, noted either by a name, or a register-displacement pair in the format of disp(register). The disp term is an unsigned decimal integer, in the range $[0,4097]$, and the register is either an element of the set $\{0,1,2,\dots,15\}$ or an identifier that is (externally) related to one from the above set.

item - Either an addr or a numeric literal. A numeric literal is of the form:

=E'E-number'

where an E-number is in the form specified by CONSTANT in the productions to follow.

Also, an $F_reg\#$ is one of the set $\{0,2,4,6\}$, denoting one of the floating point registers.

The instructions to be used, along with their required operand forms, are:

```

LDR      F_reg#1,F_reg#2:  contents(F_reg#1)=
                           contents(F_reg#2)

LNDR     F_reg#1,F_reg#2:  contents(F_reg#1)=
                           - contents(F_reg#2)

LPDR     F_reg#1,F_reg#2:  contents(F_reg#1)=
                           |contents(F_reg#2)|

LD       F_reg#,addr:  contents(F_reg#)=
                           contents(memory(addr))

STD      F_reg#,addr:  contents(memory(addr))=
                           contents(F_reg#)

ADR      F_reg#1,F_reg#2:  contents(F_reg#1)=
                           contents(F_reg#1)
                           +contents(F_reg#2)

SDR      F_reg#1,F_reg#2:  contents(F_reg#1)=
                           contents(F_reg#1)
                           -contents(F_reg#2)

MDR      F_reg#1,F_reg#2:  contents(F_reg#1)=
                           contents(F_reg#1)
                           *contents(F_reg#2)

DDR      F_reg#1,F_reg#2:  contents(F_reg#1)=
                           contents(F_reg#1)
                           /contents(F_reg#2)

AD       F_reg#,addr:  contents(F_reg#)=contents(F_reg#)
                           +contents(memory(addr))

SD       F_reg#,addr:  contents(F_reg#)=contents(F_reg#)
                           -contents(memory(addr))

MD       F_reg#,addr:  contents(F_reg#)=contents(F_reg#)
                           *contents(memory(addr))

DD       F_reg#,addr:  contents(F_reg#)=contents(F_reg#)
                           /contents(memory(addr))

```

where contents(argument) specifies that the contents of the area, either a memory location (specified by memory(addr)) or a floating point register (F_reg#) is used as an argument. All operations are on double precision data.

With the above points in mind, we are ready to proceed to the productions. The productions are not an entity unto themselves, but are a part of a larger set of productions comprising a complete language definition. The productions are:

```
(initially, .SET(LOADFLAG, 'NO') .SET(RP,-2) .SET(TP,-8))

AREX = ('-' TERM .OUT(' LNDR ' .CCNVT(RP)', ' .CCNVT(RP)) |
        ('+' | .TRUE) TERM)
        $('+' TERM .SET(OPCODE, ' ADR' ) BINOP | '-' TERM
          .SET(OPCODE, ' SDR' ) BINOP);

TERM = FACTOR $('*' FACTOR .SET(OPCODE, ' MDR' ) BINOP |
               '/' FACTOR .SET(OPCODE, ' DDR' ) BINOP);

FACTOR = .ID .SET(VAL,*) LOAD | '(' AREX ')' |
          '|' AREX '|' .OUT(' LPDR ' .CCNVT(RP)', ' .CCNVT(RP)) |
          .INT .SET(VAL, '=E' ' ' * ' ') LOAD;

LOAD = .CIF('YES', LOADFLAG, LOAD_1, LOAD_2);

LOAD_1 = .AIF(RP-5, .TRUE, .STOP,
              .OUT(' STD 0, ' .CCNVT(TP+8) '(TREG)',
                  ' STD 2, ' .CCNVT(TP+16) '(TREG)',
                  ' STD 4, ' .CCNVT(TP+24) '(TREG)',
                  ' LDR 0,6') .SET(TP,TP+24) .SET(RP,0))
              .SET(RP,RP+2) .OUT(' LD ' .CCNVT(RP)', ' ITEM) LOAD_2;

LOAD_2 = .SET(LOADFLAG, 'YES') .SET(ITEM,VAL);

BINOP = .CIF('NO', LOADFLAG, BINOP-1,
             .PART(OPCODE, 1, .LENGTH(OPCODE)-1, OPCODE)
             .OUT(OPCODE ' ' .CCNVT(RP)', ' ITEM)
             .SET(LOADFLAG, 'NO'));

BINOP_1 = .AIF(RP, .STOP, GINOP-2, .TRUE) .SET(RP,RP-2)
          .OUT(OPCODE ' ' .CCNVT(RP)', ' .CCNVT(RP+2));
```

```

BINOP_2 = .AIF(TP, .STOP, .OUT(' LDR 2,0', ' LD 0,0(TREG)')
          .SET(RP,2) .SET(TP,-8),
          .OUT(' LDR 4,0',
              ' LD 2,' .CCNVT(TP) '(TREG)'
              ' LD 0,' .CCNVT(TP-8) '(TREG)')
          .SET(TP,TP-16) .SET(RP,4));

.END;

```

Let us translate some arithmetic expressions to see what object text is produced by our translator segment. Consider the expression $A*B+C*D$. Ideally, the code would be:

```

LD      0,A
MD      0,B
LD      2,C
MD      2,D
ADR     0,2

```

The code produced by the translator is:

```

LD      0,A
MD      0,B
LD      2,C
MD      2,D
ADR     0,2

```

which is exactly the code produced by the "ideal" translator.

Now, consider the expression $A+B*(C+D*(E+F))$. Ideally, the code would be:

```

LD      0,E
AD      0,F
LD      2,D
MDR     2,0
LD      0,C
ADR     0,2
LD      2,B
MDR     2,0
LD      0,A
ADR     0,2

```

This code assumes that commutativity is not allowed, but that register usage is free. From the defined translator the code

is:

```

LD      0,A
LD      2,B
LD      4,C
LD      6,D
STD     0,0(TREG)
STD     2,8(TREG)
STD     4,16(TREG)
LDR     0,6
LD      2,E
AD      2,F
MDR     0,2
LDR     4,0
LD      2,16(TREG)
LD      0,8(TREG)
ADR     2,4
MD      0,2
LDR     2,0
LD      0,0(TREG)
ADR     0,2

```

As can be seen, the optimization method defined by the translator segment does not approach the code possible from a hand translation for some expressions, but it does eliminate some redundant load instructions for other expressions.

An interesting problem in the area of symbol manipulation is the task of symbolic differentiation. A complete differentiation. A complete differentiation routine is beyond the scope of being an example, as it is a complex task. The example to follow will only perform differentiation on polynomials, and will not perform simplifications of any sort on the derivative.

Also to be demonstrated in this example is the method of handling "item at a time" processing, where each equation is a complete block of input (item). Thus, the process will be

to read an equation, differentiate it, output the derivative, then repeat the process until there are no more input blocks.

The productions for this are:

```

RUN = .DCL(.NUMERIC('C','E'))
      $(.READ .EOF(.STOP) EQUATION);

EQUATION = FIRST_TERM $(('+' .SET(C,1) | '-' .SET(C,-1))
      TERM);

FIRST_TERM = .INT .SET(C, .ACNVT(*)) ('X' .INT .SET(E,
      .ACNVT(*)) | .SET(C,C*E)
      .AIF(|C|, .STOP, .TRUE, .OUT(.CCNVT(C) 'X'
      .CCNVT(E-1))));

TERM = .UNINT .SET(C,C*.ACNVT(*))
      ('X' .INT .SET(E, .ACNVT(*)) |
      .SET(E,0)) .SET(C,C*E) .AIF(C, .OUT(.CCNVT(C) 'X'
      .CCNVT(E-1)), .TRUE,
      .OUT('+' .CCNVT(C) 'X' .CCNVT(E-1)));

.END;
```

An example of the action described would be:

input: 5 + 8X10 - 7X1 + 10X7 + 1X0

output: 80X9 - 7X0 + 70X6.

Note that 5 and 5X0 are equivalent, and that coefficients and exponents must be present as 7X1 rather than 7X. Also, the input of 8 (or 8X0) would produce a null output.

THE META-T COMPILER

A program written using MTPL may be termed a compiler. As a mapping may be specified taking MTPL programs to M-TM code, the equivalent program in M-TM code also specifies a compiler. To effect this mapping (translation), another compiler (a compiler-compiler) is required. This is the task assumed by the META-T Compiler (M-TC).

The META-T Compiler is initially described using MTPL. To use the compiler, it must first be translated to the equivalent M-TM code. Two main approaches may be used in this bootstrapping situation. Either the MTPL program may be hand translated to its equivalent M-TM code, or a compiler may be generated either using one of the existing mechanical translators available (META-PI, for example) or some programming language other than M-TM. No recommendation will be made for one approach over another, except to say that even though an available mechanical translator may not totally provide the translation function, if only minor hand alteration is required, it may yet be easier to use the mechanical translator rather than to perform a hand translation. It should be noted that the semantics action of the META-T Compiler inserts the pair of instructions STOP and END as the last instructions of the first production, and the pair of instructions EXIT and END as the last instructions of all

subsequent procedures. Also, the PROGEND instruction is emitted automatically. The above actions ensure that the M-TM program will be able to execute properly.

The compiler is best described by reading the MTPL productions. The following should aid in their reading.

1. Data areas:

END - a NUMERIC type area used to signal that the
.END for the MTPL program has been encountered.

LC - a NUMERIC type area used for control purposes
in the OUTSKIP production.

C1,C2 - two STRING type areas used for temporary
storage during the translation process.

REF - a STRING type area that holds the results of a
successful parse of a name by the REF produc-
tion.

REFS - a STACK area used in the REF production.

2. OUTSKIP production:

When the parse of a MTPL statement fails, the begin-
ning of the next statement must be found. The
OUTSKIP production performs this function. It uses
a state transition method to guide the processing.
The variable LC is used for holding the "level-
control", or last state specification. The transi-
tions are (LC is initially set to 0):

LC=0 If the next character is a single quote,

set LC to 1 and continue. If the next character is a semicolon, exit with TF set to true. Otherwise, skip over the next character and continue with LC=0.

LC=1 If the next character is a single quote, set LC to 3 and continue. Otherwise, skip over the next character and continue with LC set to 2.

LC=2 If the next character is a single quote, set LC to 0 and continue. Otherwise, skip over the next character and continue with LC unchanged.

LC=3 If the next character is a semicolon, exit with TF set to true. Otherwise, skip over the next character and continue with LC set to 0.

The productions defining M-TC are:

```

RUN = .DCL(.NUMERIC('LC','END'), .STRING('C1','C2','REF'),
  .STACK('REFS')) .SET(END,0) .READ $COMMENT META_STMT ';'
  .OUT(': STOP;', ':END;')
  $.AIF(END, .EXITF, RUN_STMT .TRUE .OUT(':EXIT;', ':END;')
    .OUT(':PROGEND;') .FALSE);

RUN_STMT = COMMENT | .LATCH(META_STMT) ';' |
  '.END' (';' | .TRUE) .SET(END,1) | OUT_SKIP;

COMMENT = '#¢' .TO('¢#') '¢#';

```

```

OUT_SKIP = .SET(LC,0) $(.AIF(LC-1,
    .NEXT(Q) .SET(LC,1) | .NEXT(';') .EXITT | .NEXT(A),
    .NEXT(Q) .SET(LC,3) | .NEXT(A) .SET(LC,2),
    .AIF(LC-2, .STOP, .NEXT(Q) .SET(LC,0) | .NEXT(A),
    .NEXT(';') .EXITT | .NEXT(A) .SET(LC,0)));

META_STMT = .ID .SET(C1,*) '=' .OUT(C1 ': HEAD ;') META_EXPR;

META_EXPR = META_TERM $('| ' .OUT(': BT ' *1 ';') META_TERM)
    .OUT(*1 ': EQU ;') ;

META_TERM = $COMMENT META_FACTOR .OUT(': BF ' *1 ';')
    $(COMMENT | META_FACTOR .OUT(': BER ;'))
    .OUT(*1 ': EQU ;') ;

META_FACTOR = META_PRIMARY (ERROR | .TRUE);

ERROR = '.ERR' '(' .OUT(': BT ' *1 ';')
    (CITEM .OUT(': ERR ;') | .TRUE) (', ' META_EXPR | .TRUE)
    ')' .OUT(*1 ': EQU ;') ;

META_PRIMARY = META_FUNCTION | .ID .OUT(': CALL ' * ';') |
    '@' .ID .OUT(': CALL @' * ';') |
    .SR .OUT(': TST ' .CCNVT(.LENGTH(*)) '. ' * ';') |
    '(' META_EXPR ')' |
    '$' .OUT(*1 ': EQU ;') META_EXPR .OUT(': BT ' *1 ';') ;

META_FUNCTION = PRIMITIVE_FUNCTION | DATA_FUNCTION |
    CONTROL_FUNCTION | MISC_FUNCTION ;

PRIMITIVE_FUNCTION = '.ID' .OUT(': ID ;') |
    '.SR' .OUT(': SR ;') | '.UNINT' .OUT(': UNINT ;') |
    '.INT' .OUT(': INT ;') | '.LETTER' .OUT(': LETTER ;') |
    '.DIGIT' .OUT(': DIGIT ;') | '.TRUE' .OUT(': TRUE ;') |
    '.FALSE' .OUT(': FALSE ;') |
    '.NEXT' '(' (('A' | 'Q' | 'L' | 'D' | 'U')
        .OUT(': NEXT ' * ';') |
    '?' REF .OUT(': NEXT ?' REF ';') |
    .SR .OUT(': NEXT ' .CCNVT(.LENGTH(*)) '. ' * ';')) ')' ;

DATA_FUNCTION = '.DCL' '(' DCLENTRY $(',' DCLENTRY) ')' |
    '.FREE' '(' FREETYPE $(',' FREETYPE) ')' |
    '.LATCH' '(' .ID .OUT(': LATCH ' * ';')
        '@' .ID .OUT(': LATCH @' * ';') ')' |
    '.CANCEL' .OUT(': CANCEL ;') |
    '.UNLATCH' .OUT(': UNLATCH ;') |
    '.EOF' '(' .OUT(': BEOF ' *1 ';') META_EXPR ')'
    .OUT(*1 ': EQU ;') |

```

```

'.TYPE' '(' REF ',' .OUT(': BCH ' REF ',' *1 ';',
      ': BNUM ' REF ',' *2 ';') META_EXPR ','
      .OUT(': B ' *3 ';', *1 ': EQU ;') META_EXPR ','
      .OUT(': B ' *3 ';', *2 ': EQU ;') META_EXPR ')'
      .OUT(*3 ': EQU ;') |
'.EMPTY' '(' REF ',' .OUT(': BNMTY ' REF ',' *1 ';',
      META_EXPR ',' .OUT(': B ' *2 ';', *1 ': EQU ;')
      META_EXPR ')'.OUT(*2 ': EQU ;') |
'.SET' '(' REF ',' .SET(C2,REF ';')
      (AREX .OUT(': STA ' C2) | CITEM .OUT(': STB ' C2))
      ') |
'.PUSH' '(' REF ',' .SET(C2,REF ';') CITEM ')'.
      .OUT(': PUSH ' C2) |
'.SWAP' '(' REF .OUT(': SWAP ' REF ';')
      $(' ' REF .OUT(': SWAP ' REF ';')) ')' |
'.READ' .OUT(': READ ;') |
'.PART' '(' REF ',' AREX ',' AREX ',' CITEM ')'.
      .OUT(': PART ' C2) |
'.ENTER' '(' REF ',' .SET(C2,REF ';') CITEM ',' REF ','
      .SET(C2,C2 REF ';') REF ',' .SET(C2,C2 REF ';') REF
      ')'.OUT(': ENTER ' C2 REF ';') |
'.FIND' '(' REF ',' .SET(C2, REF ';') REF ','
      .SET(C2, C2 REF ';') REF ',' .SET(C2,C2 REF ';')
      REF ',' .SET(C2,C2 REF ';') REF ')'.
      .OUT(': FIND ' C2 REF ';') |
'.SLINK' ('S' | 'P' | 'F') .SET(C2,*) '(' REF ','
      CITEM ')'.OUT(': SLINK ' C2 ',' REF ';') |
'.GLINK' ('S' | 'P' | 'F') .SET(C2,*) '(' REF ','
      .SET(C2,C2 ',' REF ';') REF ')'.
      .OUT(': GLINK ' C2 REF ';') ;

CONTROL_FUNCTION = '.EOS' '(' .OUT(': BNEOS ' *1 ';')
      META_EXPR ')'.OUT(*1 ': EQU ;') |
      '.RESCAN' .OUT(': RESCAN ;') |
      '.AIF' '(' AREX ',' .OUT(': BGE ' *1 ';') META_EXPR ','
      .OUT(': B ' *3 ';', *1 ': BNEP ' *2 ';') META_EXPR ','
      .OUT(': B ' *3 ';', *2 ': EQU ;') META_EXPR ')'.
      .OUT(*3 ': EQU ;') |
      '.CIF' '(' CITEM ',' REF ','
      .OUT(': BCNE ' REF ',' *1 ';') META_EXPR ','
      .OUT(': B ' *2 ';', *1 ': EQU ;') META_EXPR ')'.
      .OUT(*2 ': EQU ;') |
      '.STOP' .OUT(': STOP ;') |
      '.EXIT' ('T' .OUT(': TRUE ;') | 'F' .OUT(': FALSE ;') |
      .TRUE) .OUT(': EXIT ;') |
      '.TO' '(' CITEM ')'.OUT(': TO ;') ;

```

```

MISC_FUNCTION = '.OUT' '(' CITEM .OUT(': OUT ;')
                $(',' CITEM .OUT(': OUT;')) ')' |
                '.NEWIN' '(' CITEM .OUT(': NEWIN ;')
                $(',' CITEM .OUT(': NEWIN ;')) ')' |
                '.IGN' '(' CITEM .OUT(': RESET ;')
                $(',' CITEM .OUT(': RESET ;')) ')' ;

AREX = ('-' TERM .OUT(': NEG ;') | ('+' | .TRUE) TERM)
        $('+' TERM .OUT(': ADD ;') | '-' TERM .OUT(': SUB ;')) ;

TERM = FACTOR $('*' FACTOR .OUT(': MPY ;') |
                '/' FACTOR .OUT(': DIV ;')) ;

FACTOR = '.REM' '(' AREX ',' AREX ')' .OUT(': REM ;') |
          '.ACNVT' '(' CITEM ')' .OUT(': ACNVT ;') |
          '.LENGTH' '(' CITEM ')' .OUT(': LEN ;') |
          '.DIM' '(' REF ')' .OUT(': DIM ' REF ;') |
          REF .OUT(': LD ' REF ;') | '(' AREX ')' |
          '|' AREX '|' .OUT(': ABS ;') |
          '.INT .OUT(': LD =' .CCNVT(.LENGTH(*)) '.' * ;') ;

REF = REFTERM | '@' REFTERM .SET(REF, '@' REF);

REFTERM = .ID .PUSH(REFS, *) ((' AREX ')' | .OUT(': LD =1.0'))
          .SET(REF, .POP(REFS));

CITEM = $('.SWAP' '(' REF .OUT(': SWAP ' REF ;')
          $(',' REF .OUT(': SWAP ' REF ;')) ')' |
          '*' (.UNINT .OUT(': GN ' * ;') | .OUT(': LDC ;')) |
          '.POP' '(' REF ')' .OUT(': POP ' REF ;') |
          '.CCNVT' '(' AREX ')' .OUT(': CCNVT ;') |
          REF .OUT(': LDB ' REF ;') |
          '.SR .OUT(': LDB =' .CCNVT(.LENGTH(*)) '.' * ;')) ;

DCLENTY = TYPE '(' CITEM .OUT(': D' C1 ;')
              $(',' CITEM .OUT(': D' C1 ;')) ')' ;

FREETYPE = TYPE '(' CITEM .OUT(': F' C1 ;')
               $(',' CITEM .OUT(': F' C1 ;')) ')' ;

TYPE = '.NUMERIC' .SET(C1, 'N') | '.STRING' .SET(C1, 'C') |
        '.VECTOR' .SET(C1, 'V') | '.STACK' .SET(C1, 'S') |
        '.HASH' .SET(C1, 'H');

.END;

```

CONCLUSIONS AND FUTURE WORK

The preceding chapters are mainly slanted towards the design and implementation of language translators, this being the main thrust of the META-series compiler-compilers. But the areas of application of META-T are broader than just producing a translator to emit object text for some language.

META-T's main area of application is in providing a research environment whereby users may experiment with the structure of translators. Textbook theories may be tested and evaluated, without the problem of designing the supporting utility routines. Another area of impact is in the design of interpreters for languages. Because of the general programming facilities provided by MTPL (and ultimately, M-TM), one may design an interpreter for, say, APL.

Some areas of research the author wants to look into are those of computer description languages, microprogramming languages, and statistics gathering systems. Computer description languages, to be useful, must be implemented and used. META-T will be useful in providing translators and interpreters for these languages.

There are two reasons for looking into microprogramming languages. The first is the ability to provide translators for these languages. Secondly, as M-TM has been designed to be easily interpreted, it should also be equally possible to

develop a mapping from M-TM instructions to microcode and get a hardware (firmware) implementation of M-TM.

In the area of statistics gathering, one may easily gather information about a language. Instead of producing machine code for a language, a translator may be developed for, say, FORTRAN, that would tally the number of different operations (assignment, I/O, comparison, etc.) and produce a report. Also, for each statement in the language, the operations dictated by the statement may be reported. Thus, information may be acquired for use in tailoring instruction sets for computers.

Another research area is that of the hardware or firmware implementation of high level machines, such as APL machines, BASIC machines, etc. One approach to doing such a task is to define the translator and associated interpreter for the language using MTPL. Then, using M-TC, the translator and interpreter in M-TM code would result. If the M-TM instructions and facilities used by the translator and interpreter were implemented in either hardware or firmware and the M-TM program were put into read-only memory, the desired high level machine would exist.

A very important research area is in optimizing the M-TM structure. It should be obvious that the M-TM structure produces tediously long programs, mainly as a result of the subscripting method. A judicious revision of this area is

indicated. Also, minor alterations to the set of M-TM instructions would result in an even more versatile instruction set.

BIBLIOGRAPHY

1. Feldman, J. A. "A Formal Semantics for Computer Languages and Its Application in a Compiler-Compiler." Communications of the Association for Computing Machinery, 9 (January 1966), 3-9.
2. Gries, David. Compiler Construction for Digital Computers. New York: John Wiley and Sons, Inc., 1971.
3. IBM System/360 and System/370 FORTRAN IV Language. File No. S360-25, Order No. GC28-6515-8.
4. IBM System/360 Operating System Job Control Language Reference. File No. S360-36, Order No. GC28-6704-2.
5. IBM System/360 Operating System PL/I (F) Language Reference Manual. File No. S360-29, Order No. GC28-8201-3.
6. Lucas, P.; Lauer, P.; and Stigleitner, H. "Method and Notation for the Formal Definition of Programming Languages." Technical Report TR 25.087, July 1, 1970. Austria: IBM Laboratory Vienna.
7. Lucido, Anthony P. "META-T and the Implementation of Special Application Languages." Paper presented at the Seventh Annual Conference on the Interface: Statistics and Computer Science, October 18-19, 1973.
8. Naur, Peter (Ed.) "Revised Report on the Algorithmic Language ALGOL 60." Programming Systems and Languages. Edited by Saul Rosen. New York: McGraw-Hill Book Co., 1967.
9. O'Neil, John T. "META-PI: An On-Line Interactive Compiler-Compiler." Proceedings of the Fall Joint Computer Conference, 33, Part I (1968), 201-218.
10. Rosen, Saul. "A Computer-Building System Developed by Brooker and Morris." Programming Systems and Languages. Edited by Saul Rosen. New York: McGraw-Hill Book Co., 1967.

11. Schneider, F. W., and Johnson, G. D. "META-3: A Syntax-Directed Compiler Writing Compiler to Generate Efficient Code." Proceedings of the ACM 19th Annual Conference, D1.5-1, 1964.
12. Schorre, D. V. "META-II: A Syntax-Oriented Compiler Writing Language." Proceedings of the ACM 19th Annual Conference, D1.3-1, 1964.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major professor Dr. Roy F. Keller. Without his guidance and patience, neither my research nor abilities would have come to fruition. Thanks are also due to my other committee members, Drs. R. J. Lambert, S. M. Silverston, A. V. Pohm, and T. A. Smay.

Finally, I would like to thank my wife Barbara who has done an excellent job at typing and at being there when needed.

APPENDIX. LIST OF M-TM INSTRUCTIONS AND MTPL
PRIMITIVES WITH PAGE REFERENCES

M-TM instruction	Page	M-TM instruction	Page
ABS	55	GN	60
ACNVT	56	HEAD	32
ADD	55	ID	44
B	41	INT	45
Bc	38	LATCH	34
BCE	39	LD	57
BCH	40	LDB	58
BCNE	39	LDC	60
BcP	39	LEN	56
BEOF	39	LETTER	45
BEOS	38	MPY	55
BER	36	NEG	55
BF	36	NEWIN	59
BMTY	40	NEXT	46
BNCH	41	OUT	59
BNEOF	40	PART	49
BNEOS	38	POP	57
BNMTY	40	PROGEND	35
BNNUM	41	PUSH	48
BNUM	41	READ	43
BT	36	REM	55
CALL	33	RESCAN	54
CANCEL	37	RESET	54
CCNVT	57	SE	28
Dc	58	SLINK	53
DIGIT	45	SR	45
DIM	56	STA	47
DIV	55	STB	48
END	33	STOP	35
ENTER	50	SUB	55
EQU	35	SWAP	48
ERR	37	TO	41
EXIT	35	TRUE	37
FALSE	38	TST	45
Fc	59	UNINT	45
FIND	51	UNLATCH	37
GLINK	53		

MTPL primitive	Page	MTPL primitive	Page
.ACNVT	75	.LATCH	72
.AIF	74	.LENGTH	76
.CANCEL	72	.LETTER	71
.CCNVT	76	.NEWIN	75
.CIF	74	.NEXT	71
.DCL	72	.OUT	75
.DIGIT	71	.PART	73
.DIM	76	.POP	76
.EMPTY	73	.PUSH	73
.ENTER	73	.READ	73
.EOF	72	.REM	75
.EOS	74	.RESCAN	74
.ERR	76	.SET	73
.EXIT	75	.SLINK	74
.EXITF	75	.SR	71
.EXITT	75	.STOP	75
.FALSE	71	.SWAP	73
.FIND	73	.TO	75
.FREE	72	.TRUE	71
.GLINK	74	.TYPE	72
.ID	71	.UNINT	71
.IGN	75	.UNLATCH	72
.INT	71		