TTM: AN EXPERIMENTAL INTERPRETIVE LANGUAGE

By

Stephen H. Caine E. Kent Gordon

Willis H. Booth Computing Center

Programming Report No. 7 July 1968

California Institute of Technology

Note: This document has been merged with the following ttm document:

TTM: A Macro Language for Batch Processing, by Stephen H. Caine E. Kent Gordon, Willis H. Booth Computing Center, Programming Report No. 8 May, 1969, California Institute of Technology.

This is because the two documents are mostly the same. Text from the batch processing document are marked where it occurs with the tag [Batch]. Text that is in the interactive version only is marked with the tag [Interactive].

TABLE OF CONTENTS

[TTM: AN EXPERIMENTAL INTERPRETIVE LANGUAGE 1](#_Toc337390339)

[1. Introduction 6](#_Toc337390340)

[2. Language Structure 6](#_Toc337390341)

[2.1 Characters and Strings 6](#_Toc337390342)

[2.2 Functions 6](#_Toc337390343)

[2.3 Special Delimiting 7](#_Toc337390344)

[3. The Processing Algorithm 7](#_Toc337390345)

[4. TTM Functions 13](#_Toc337390346)

[4.1 Dictionary Operations 13](#_Toc337390347)

[4.1.1 Define a String 13](#_Toc337390348)

[4.1.2 Append to a String 13](#_Toc337390349)

[4.1.3 Erase Strings 14](#_Toc337390350)

[4.1.4 Copy a Function 14](#_Toc337390351)

[4.1.5 Segment a String 14](#_Toc337390352)

[4.1.6 Segment and Count 14](#_Toc337390353)

[4.1.7 Mark for Creation 14](#_Toc337390354)

[4.2 String Selection 15](#_Toc337390355)

[4.2.1 Call One Character 15](#_Toc337390356)

[4.2.2 Call N Characters 15](#_Toc337390357)

[4.2.3 Skip N Characters [Batch] 15](#_Toc337390358)

[4.2.4 Initial Character Scan 15](#_Toc337390359)

[4.2.5 Character Scan 15](#_Toc337390360)

[4.2.6 Call Parameter 15](#_Toc337390361)

[4.2.7 Call Segment 16](#_Toc337390362)

[4.2.8 Reset Residual Pointer 16](#_Toc337390363)

[4.3 String Scanning Operations 16](#_Toc337390364)

[4.3.1 Give N Characters 16](#_Toc337390365)

[4.3.2 Zero-Level Commas 16](#_Toc337390366)

[4.3.3 Zero Level Commas and Parentheses 16](#_Toc337390367)

[4.3.4 Flip a String (FLIP) [Batch] 16](#_Toc337390368)

[4.3.5 Translate to Lower Case (TRL) [Batch] 17](#_Toc337390369)

[4.3.6 Translate Hexadecimal to Decimal (THD) [Batch] 17](#_Toc337390370)

[4.4 Character Class Operations 17](#_Toc337390371)

[4.4.1 Define a Class 17](#_Toc337390372)

[4.4.2 Define a Negative Class 17](#_Toc337390373)

[4.4.3 Erase a Class 17](#_Toc337390374)

[4.4.4 Call Class 17](#_Toc337390375)

[4.4.5 Skip Class 17](#_Toc337390376)

[4.4.6 Test Class 18](#_Toc337390377)

[4.4.7 Arithmetic Operations 18](#_Toc337390378)

[4.4.8 Add 18](#_Toc337390379)

[4.4.9 Subtract 18](#_Toc337390380)

[4.4.10 Multiply 18](#_Toc337390381)

[4.4.11 Divide and Give Quotient 18](#_Toc337390382)

[4.4.12 Divide and Give Remainder 18](#_Toc337390383)

[4.4.13 Obtain Absolute Value 19](#_Toc337390384)

[4.5 Numeric Comparisons 19](#_Toc337390385)

[4.5.1 Compare Numeric “Equal” 19](#_Toc337390386)

[4.5.2 Compare Numeric "Greater Than" 19](#_Toc337390387)

[4.5.3 Compare Numeric "Less Than" 19](#_Toc337390388)

[4.6 Logical Comparisons 19](#_Toc337390389)

[4.6.1 Compare Logical “Equal" 19](#_Toc337390390)

[4.6.2 Compare Logical "Greater Than” 20](#_Toc337390391)

[4.6.3 Compare logical "Less Than” 20](#_Toc337390392)

[4.7 Peripheral Input/Output Operations [Interactive] 20](#_Toc337390393)

[4.7.1 Read a String 20](#_Toc337390394)

[4.7.2 Set to Read from Cards 20](#_Toc337390395)

[4.7.3 Print a String 20](#_Toc337390396)

[4.8 Print String and Read 21](#_Toc337390397)

[4.8.1 Change Meta Character 21](#_Toc337390398)

[4.9 Formatted Output Operations 21](#_Toc337390399)

[4.9.1 Format a Line or Card 21](#_Toc337390400)

[4.9.2 Declare Tab Positions 21](#_Toc337390401)

[4.9.3 Set Continuation Convention 22](#_Toc337390402)

[4.9.4 Insert a Control Character 22](#_Toc337390403)

[4.9.5 Output the Buffer 22](#_Toc337390404)

[4.10 Library Operations 23](#_Toc337390405)

[4.10.1 Store a Program 23](#_Toc337390406)

[4.10.2 Delete a Program 23](#_Toc337390407)

[4.10.3 Copy a Program 23](#_Toc337390408)

[4.10.4 Show Program Names 23](#_Toc337390409)

[4.10.5 Obtain String Names 24](#_Toc337390410)

[4.10.6 Determine if a Program is in the Library [Batch] 24](#_Toc337390411)

[4.10.7 Declare Standard Qualifiers [Batch] 24](#_Toc337390412)

[4.11 Utility Operations 24](#_Toc337390413)

[4.11.1 Determine if a Name is Defined 24](#_Toc337390414)

[4.11.2 Obtain the Norm of a String 24](#_Toc337390415)

[4.11.3 Program Break 24](#_Toc337390416)

[4.11.4 Obtain Execution Time [Interactive] 25](#_Toc337390417)

[4.11.5 Turn Trace On 25](#_Toc337390418)

[4.11.6 Turn Trace Off 25](#_Toc337390419)

[4.11.7 Return from TTM [Interactive] 25](#_Toc337390420)

[4.11.8 Test for End of String [Batch] 25](#_Toc337390421)

[5. Functions for the Batch Version [Batch] 25](#_Toc337390422)

[5.1 Monitor Control Functions 26](#_Toc337390423)

[5.1.1 Control Output of Input Monitor 26](#_Toc337390424)

[5.1.2 Control Handling of TTM Programs 26](#_Toc337390425)

[5.2 Input Functions 26](#_Toc337390426)

[5.2.1 Input One Card 26](#_Toc337390427)

[5.2.2 Control CD Input 26](#_Toc337390428)

[5.2.3 Input Next Complete Fortran Statement 26](#_Toc337390429)

[5.2.4 Control FOR input 27](#_Toc337390430)

[5.2.5 Look Ahead One Card 27](#_Toc337390431)

[5.2.6 Control PK Input 27](#_Toc337390432)

[5.2.7 Output Functions Print a String 27](#_Toc337390433)

[5.2.8 Specify Length of Page 27](#_Toc337390434)

[5.2.9 Space Before Printing 27](#_Toc337390435)

[5.3 Formated Output Operations 28](#_Toc337390436)

[5.3.1 Format a Line or Card 28](#_Toc337390437)

[5.3.2 Declare Tab Positions 28](#_Toc337390438)

[5.3.3 Set Continuation Convention 28](#_Toc337390439)

[5.3.4 Control FM Output 28](#_Toc337390440)

[5.4 Utility Functions 29](#_Toc337390441)

[5.4.1 Obtain Time of Day 29](#_Toc337390442)

[5.4.2 Obtain Execution Time 29](#_Toc337390443)

[5.4.3 Define Error String 29](#_Toc337390444)

[6. Examples 29](#_Toc337390445)

[6.1 Def 29](#_Toc337390446)

[6.1.1 Defer 30](#_Toc337390447)

[6.1.2 Basic Examples 30](#_Toc337390448)

[6.1.3 Keyword Parameters 32](#_Toc337390449)

[6.1.4 A TTM Operating System 35](#_Toc337390450)

[6.1.5 Expression Evaluation 36](#_Toc337390451)

[7. Using the Conversational System 38](#_Toc337390452)

[7.1 Initiating a Session 38](#_Toc337390453)

[7.2 Calling for TTM 38](#_Toc337390454)

[7.3 Terminating Session 38](#_Toc337390455)

[7.4 Upper and Lower Case 38](#_Toc337390456)

[7.5 General Error Correction 39](#_Toc337390457)

[7.6 Entering a TTM Program 39](#_Toc337390458)

[7.7 The Attention (ATTN) Key 39](#_Toc337390459)

[7.8 System Commands 39](#_Toc337390460)

[7.9 Color Shift 40](#_Toc337390461)

[8. Using the Batch Version [Batch] 41](#_Toc337390462)

[9. Appendix A Error Messages 42](#_Toc337390463)

[10. Appendix B Function Attributes 44](#_Toc337390464)

[11. Appendix C Storage Overflow 46](#_Toc337390465)

[12. Bibliography 47](#_Toc337390466)

# Introduction

TTM Is a recursive, interpretive language designed primarily for string manipulation, text editing, macro definition and expansion, and other applications generally classified as systems programming. It is derived, primarily, from GAP [12] and GPM [11].

Initially, TIM was planned as the macro processing portion of an assembler for the IBM System/360 and, as such, was designed to overcome the restrictions and inconsistencies which existed in the standard assemblers for that system [9,10], In addition, it was designed to have all of the power possessed by earlier general macro assemblers but with the unfortunate syntactic and semantic difficulties removed [4,5,6,7,8].

During the development of TIM, it became apparent that applications other than assembler macro processing were possible. These include data editing, text manipulation, expression compiling, and macro processing for language processors other than assemblers.

The initial version of TTM was implemented to run in a conversational manner under the Caltech Basic Time Sharing System for the IBM System/360-50 [13]. This manual describes that conversational version. Other versions have been, or will be, written to run in the batch processing environment of OS/360 and to operate in front of or in conjunction with various language processors. These will be described in other publications.

# Language Structure

## Characters and Strings

The basic unit of information in TTM is the character. The characters acceptable to TIM are:

* The lower case letters, a – z
* The upper case letters, A – Z
* The decimal digits, 0-9
* The special characters, < > ( ) # + - \* / , & | ~ @ $ ; % ' ? : = " . ! \_ blank
* An internal character representing end-of-line and known as the carriage return (CR) character

In certain contexts the special characters < > # ; @ CR act as control characters. The contexts under which this special treatment occurs are discussed below.

For ease of processing, characters are normally collected into units known as strings. A string is a sequence (possibly empty) of characters which is usually manipulated as a single entity. Strings may be placed Into the TTM dictionary (memory), in which case they must be named. The name of a string consists of an arbitrary number (including zero) of characters.

## Functions

The contents of a string may be arbitrary text of some kind or it may represent a program which is to be executed by TTM. In general, program and text (data) are indistinguishable except by the context in which they are used.

When a string is fetched out of memory and executed, text may be substituted for previously declared substrings or bound variables. Thus, a string to be executed may be considered, in some sense, as a function. In addition to such programmer-defined functions, TTM contains a number of built-in functions \*

To invoke a function, the following notation is used:

#<sl;s2;s3;…;sn>

where the si are strings and n > 1. The first string, s1, is taken to be the name of the function and the remaining strings to be its parameters.

This form is known as an active call. A passive form also exists:

##<s1;s2;s3;...;sn>

The differences between these two forms will be discussed in the section on the processing algorithm.

The pound sign (#) is recognized as a control character only when it appears in the context #< or ##<.In all other cases, it is simply considered to be part of a string.

The number of parameters supplied to a function is usually dependent upon the use of the function and the number of bound variables it is defined or declared to have. If fewer than the indicated number of parameters are supplied, programmer-defined functions will consider the remaining ones as null. However, some of the primitives cannot operate with fewer than a certain number of parameters. If more than the indicated number of parameters are provided, the excess ones will be ignored by both programmer-defined and primitive functions.

The result of Invoking a function is to produce a value and, for some functions, to produce side effects. A function value is always a string although it may be a null string. The value can, in some sense, be thought of as replacing the text of the function call.

## Special Delimiting

As any parameter, including the function name, is being scanned prior to calling the function, the detection of a semicolon will terminate that parameter. Also, the detection of a #< or ##< will cause the process to recurse so that the inner function will be executed while collecting the parameter of the outer function. However, this action is not always desired. Therefore, if a < is detected, no further searching for a semicolon or a function call will be made until the matching > is encountered. This outer pair of brackets will be deleted and the text between them will be considered, verbatim, as part of the parameter being collected.

This special treatment would normally mean that strings could not contain unbalanced brackets. To allow this, and to circumvent certain other problems of this type, a control character @, is introduced. This is known as the escape character. When encountered, the escape character causes the following character to be included in the scanned parameter without any examination.

The escape character itself is deleted unless it occurs inside of a bracketed string.

The carriage return character is deleted when encountered unless it occurs inside of a bracketed string.

# The Processing Algorithm

A description of the actual processing algorithm will prove useful in learning to write programs for TTM. With the processing algorithm in mind it is possible to completely predict the results of passing any sequence of characters to TTM for processing.

In general, TTM can be thought of as a computer with a somewhat unusual organization. To cause this "computer" to operate, a "program" in the language of TTM is placed in an area known as the program string. The TTM processor starts scanning at the beginning of the program string, executing function calls when encountered. Some of these may cause additional characters to be inserted into the program string. When the end of the program string is reached, TTM terminates processing. Depending upon the particular version of TTM used, the initial program may be entered from a conversational terminal, the input stream of a batch processor, or as an argument to a call on TTM from some other routine.

The actual processing consists of the three steps shown below. In this discussion, the symbols {, [, and ^ are used strictly as metalinguistic symbols. They are introduced solely as a notational convenience in explaining the algorithm and never occur in actual practice.

The processing algorithm can be expressed as:

1. Scan the program string, replacing #< with {, ##< with [, and ; with ^. If a carriage return is encountered, it is deleted. If a @ is detected, it is deleted and the following character is skipped without any examination for special significance. If a < not preceded by # or ## is detected, it is deleted and a scan is made for the matching >, which is also deleted. During this "balancing bracket" scan, only the two brackets and the @ are of significance. When encountered, the @ and the following character are passed over. The constructions #< and ##< as well as the ; and the carriage return are simply passed over. In any case, if the end of the program string is reached, processing is terminated.

2. When a > is encountered outside of a bracketed string scan, it signifies the end of the function call which begins with the immediately preceding { or [ and whose parameters are separated by a ^ The first parameter is taken as the function name, the function is invoked, and the characters from the { or [ to the matching > inclusive, are deleted.

3. If the call was initiated by a { the function value is placed to the right of the current scan position so that it will be encountered when scanning is continued. If the call was initiated by a [ the function value is placed to the left of the current scan position. In either case, scanning is continued at step one above.

Several examples may clarify the operation of the algorithm. In these, an upward-pointing arrow will be used to indicate the position of the scanner.

#### Example 1

The Print String function is written as

#<PS;string>

which has the side effect of causing string to be printed on an output device and the result of returning a null value. Consider the processing of

#<PS;123>

↑

which becomes

{PS;123>

↑

then

{PS^123>

↑

and finally prints 123 and results In a null program string.

#### Example 2

Given:

#<PS;<1;<2;3>;4>>

↑

Yields:

{ < < > >>

{PS;<1;<2;3>;4>>

↑

{PS^1;<2;3>;4>

↑

Result: Print 1;<2;3>;4

#### Example 3

Given:

#<PS;@>>

↑

Yields:

{PS^@ >>

↑

{PS^@ >>

↑

Result: Print >

#### Example 4

Given:

#<PS;<@<>>

↑

Yields:

{PS^<@<>>

↑

{PS^@<>>

↑

{PS^@<>>

↑

{PS^@<>

↑

Result: Print @<

#### Example 5

The add function is written as

#<ad;n1;n2>

and results In a value which is the sum of the numbers n1 and n2.

Given:

#<PS; #<ad;1;2>>

↑

Yields:

{PS^#<ad;l;2>>

↑{PS^{ad^l^2>>

↑{PS^3>

↑{PS^3>

↑

Result: Print 3

#### Example 6

Assume that the string named X contains the characters

1;2

Given:

#<PS;#<X>>

↑

Yields:

#<PS^#<X>>

↑

#<PS^{X>>

↑

#<PS^1;2>

↑

#<PS^1^2>

↑

Result: Prints 1

because PS expects only one parameter, thus causing the second parameter to be ignored. However, the same example but using a passive call on X gives:

Given:

#<PS;#<X>>

↑

Yields:

#<PS^#<X>>

↑

#<PS^{X>>

↑

#<PS^1;2>

↑

Result: Prints 1;2

#### Example 7

Assume that the string named Z has the value

PS

then,

Given

#<##<Z>;123>

↑

Yields:

{[Z>;123>

↑

{PS;123>

↑

{PS^123>

↑

Result: Prints 123

#### Example 8

Assume that the string named X has the value

##<ad;6;4>

then,

Given:

#<PS;##<X >>

↑

Yields:

{PS^##<X >>

↑

#<PS^[X>>

↑

#<PS^##<ad;6;4>>

↑Result: Print ##<ad;6;4>

However,

Given:

#<PS;#<X >>

↑

Yields:

{PS^#<X >>

↑

#<PS^{X>>

↑

#<PS^##<ad;6;4>>

↑

#<PS^{ad^6^4>>

↑

#<PS^10>

↑

Result: Prints 10

# TTM Functions

The built-in TTM functions are described in this section. They are divided into groups of functions with similar format or operation and each group is prefaced by an introductory description.

A standard form is used to describe each operation:

|  |  |
| --- | --- |
| Data: | name n1,n2,r |
|  | where name is the function name as used in the call; |
|  | n1 is the minimum number of parameters required; |
|  | n2 is the maximum number of parameters used (an asterisk is used to denote an arbitrary number); |
|  | r is either V or S or VS to indicate that the function can have a non-null value, produces side effects, or both has a non-null value and produces side effects. |
| Call: | Format of a call on the function |

These are followed by a description of the parameters, values, and side effects.

## Dictionary Operations

These functions are designed to define and erase named strings and to scan such strings for occurrences of specified substrings. A string so scanned is said to be segmented and the specified substrings will be replaced by segment marks.

When a named string is called out of the dictionary, the parameters specified in the call are positionally substituted for the corresponding segment marks. Missing parameters are taken as null and excess parameters are ignored.

In addition to segmenting, a string may also be scanned for substrings which are to be replaced by unique numbers each time the string is invoked. These numbers are useful in generating created symbols.

Within a single string, the number of segment marks may not exceed 62. Segmentation is cumulative. That is, if a string is segmented on AA and BB in one operation and then later segmented on CC, it is equivalent to having segmented on AA, B15, and CC in a single operation.

The search for substring matches is performed on a character by character basis while scanning the string from left to right. Each substring is searched for separately with as many passes being made over the string as there are substrings.

Special treatment is not given to any character in the string and it is, therefore, possible to search for characters such as the brackets, the semicolon, or the end-of-line character.

If the characters of one substring are contained in another substring, the larger of the two should be searched for first. Thus, it is possible to scan for both THIS and IS if THIS is searched for first.

Associated with each string are two pointers: the initial pointer and the residual pointer. The initial pointer locates the true start of the string. When a string is assigned a value, the residual pointer is set equal to the initial pointer. Whenever TTM examines a string, it begins at the point specified by the residual pointer. As the various functions in the string selection group work on a string, the residual pointer is moved down the string, thus truncating it from the left. At any time, the programmer may cause the residual pointer to be reset to the value of the initial pointer.

### Define a String

DS 2,2,S #<DS;name;text>

Side Effects: the string text is placed in the dictionary and assigned the name of name.

Note: If name has been previously defined, it will be redefined with the new text. Built-in functions can be redefined in this manner.

### Append to a String

AP 2,2,S #<AP;name;text>

Side Effects: the string text is added to end of the named string and the residual pointer is positioned to the new end of the string.

Note: Append allows successive additions to be made to a string without copying the whole string each time an addition Is made. The residual pointer will always be left at the end so it is necessary to reset it before using the string. However, for purposes of efficiency, the residual pointer should be left at the end as long as additional calls on AP are expected. If name is not defined, AP will be treated as DS.

### Erase Strings

ES 1,\*,S #<ES;name1,name1,…>

Parameter: the namei are the names of strings.

Side Effects: The given strings and their names will be deleted from the dictionary.

Note: Strings which are no longer needed should be erased as the space occupied by them is thereby recovered. ES can be applied to built-in functions in which case the name of the function will be deleted.

### Copy a Function

CF 2,2,S # <CF; new-name; old-name>

Parameters: Two strings to be used as names.

Side Effects: New-name is assigned a value which is a copy of the value of old-name starting at the residual pointer.

Note: Segment marks and created symbol marks are copied. If old name is the name of a builtin function, then new name will be defined as if it were also that builtin function.

### Segment a String

SS 2,63,S #<SS;name;S1;S2;…;Sn>

Parameters: name is the name of a string. The Si, 1 < i < 62, are strings.

Side Effects: All occurrences of the Si in string name are replaced by segment marks. Scanning begins at the position marked by the residual pointer.

### Segment and Count

SC 2,63,VS #<SS;name;S1;S1;…;Sn>

Parameters: name Is the name of a string. The Si, 1 < i < 62, are strings.

Side Effects: All occurrences of the Si in string name are replaced by segment marks. Scanning begins at the position marked by the residual printer.

Value: An unsigned decimal integer representing the number of segment marks placed into the string by this call.

### Mark for Creation

CR 2,2,S #<CR;name;S>

Parameters: name is the name of a string and S is a string.

Side Effects: All occurrences of S in string name are replaced by a special creation mark. Scanning begins at the position marked by the residual pointer.

Notes: Substring matching operates in the same way as SS operates. However, no matter how many different substrings are marked in a given string they will each be replaced by the same quantity when the string is invoked. This quantity will be a four-digit decimal number and will be different for each invocation.

## String Selection

These functions provide various ways of selecting portions of named strings for processing.

### Call One Character

CC 1,1,VS #<CC;name>

Value: the next character from the position in string name marked by the residual pointer.

Side Effect: The residual pointer is advanced by one character.

### Call N Characters

CN 2,2,VS #<CN;num;name>

Parameters: num is a positive decimal integer, name is the name of a string.

Value: the next num characters from the position in string name marked by the residual pointer.

Side Effect: The residual pointer is advanced by num characters.

### Skip N Characters [Batch]

SN 2,2,VS #<SN;num;name>

Parameters: num is a positive decimal integer, name is the name of a string.

Side Effect: the residual pointer of the string is advanced by num characters.

### Initial Character Scan

ISC 4,4,VS #<SCN;string1;name;string2;string3>

Value: string2 if the characters of string1 match the characters of the string name which immediately follow the residual pointer; string3 if there is not a match.

Side Effect: if the characters do match, the residual pointer is advanced past these characters.

### Character Scan

SCN 3,3,VS #<SCN;string1;name;string2>

Value: If the string string1 is a substring of the string name starting at the residual pointer, then the characters from the residual pointer to the beginning of the first occurrence of this substring are returned. Otherwise string2 is returned.

Side Effect: If string1 was a substring of string name starting at the residual pointer, than the residual pointer is moved past the first occurrence of this substring.

### Call Parameter

CP 1,1,VS #<CP;name>

Value: all of the characters from the position in string name marked by the residual pointer up to, but not including the next zero-level semicolon.

Side Effect: The residual pointer is spaced past the zero level semicolon.

Note: A zero-level semicolon is one not enclosed in brackets. In searching for a zero-level semicolon, TTM will process, but not delete, escape characters. Therefore, an escape character may be used to protect a bracket or a semicolon.

### Call Segment

CS 1,1,VS #<CS;name>

Value: all of the characters from the position in string name marked by the residual pointer up to, but not including, the next segment mark.

Side Effect: The residual pointer is spaced past the segment mark.

### Reset Residual Pointer

RRP 1,1,S #<RRP;name>

Side Effect: The residual pointer of string name is returned to the beginning of the string.

## **String** Scanning Operations

These functions allow various portions of a string to be selected for processing or allow a string to be reformated without requiring the string to be defined as a named string.

### Give N Characters

GN 2,2,V #<GN;num;string>

Parameters: num is a signed or unsigned decimal integer and string is an arbitrary string.

Value: If num is positive, the value is the first num characters of string. If num is negative, the value is all but the first num characters in string. If num is zero, the value is null.

### Zero-Level Commas

ZLC 1,1,V #<ZLC;string>

Value: The value is string with each zero-level comma replaced by a semicolon.

Notes: Zero-level commas are those not enclosed in parentheses.

Brackets are ignored. The escape character can be used to protect parentheses and commas.

### Zero Level Commas and Parentheses

ZLCP 1,1, V #<ZLCP;string>

Value: The value is string with all zero-level commas and parentheses replaced by semicolons.

Note: Zero-level parentheses and commas are those not enclosed in other parentheses. In replacing commas and parentheses by semicolons, care is taken not to change the number of delimited units In the string. Thus, A(B) will give A;B and (A) , (B) ,C will give A;B;C.

### Flip a String (FLIP) [Batch]

FLIP 1,1,V #<FLIP;string>

Value: string with its characters in reverse order.

### Translate to Lower Case (TRL) [Batch]

TRL 1,1,V #<TRL;string>

Value: string with all upper case alphabetic characters changed to lower case.

### Translate Hexadecimal to Decimal (THD) [Batch]

THD 1,1,V #<THD;string>

Parameter: string is up to eight hexadecimal characters.

Value: signed decimal integer with leading zeros suppressed.

## Character Class Operations

These operations allow string selection to be performed on the basis of the class to which the characters in the string belong.

The character classes have names but are quite different from strings and cannot be used as strings. It is possible to have a character class and a string with the same name.

### Define a Class

DCL 2,2,S # <DCL;clname;chars>

Side Effects: The class clname is defined to consist of the characters chars

Notes If the class clname has been previously defined, it will be redefined.

### Define a Negative Class

DNCL 2,2,S # <DNCL;clname;chars>

Side Effects: The class clname is defined to contain all characters except those in chars.

Note: If chars is null the class of all characters will be defined.

### Erase a Class

ECL 1,\*,S #<ECL;clname1;clname2;…>

Side Effect: The classes named by the clnamei will be deleted.

### Call Class

CCL 2,2,VS #<CCL;clname;name>

Value: All of the characters from the position in string name marked by the residual pointer up to but not including the first character which is not a member of the class clname.

Side Effect: The residual pointer is advanced past the characters returned as the value.

### Skip Class

SCL 2,2,S #<SCL;clname;name>

Side Effect: The residual pointer of name is advanced until it reaches a character which does not belong to the class clname.

### Test Class

TCL 4,4,V #<TCL;clname;name;string1;string2>

Value: If the character at the position in string name marked by the residual pointer belongs to the class clname, the value is string1, otherwise it is string2.

Note: If name is null the value will be string2.

### Arithmetic Operations

The arithmetic functions perform fixed-point decimal arithmetic. The addition and subtraction functions accept fifteen digit operands and produce fifteen digit results. The multiplication function accepts fifteen digit operands and produces a thirty digit result. Division accepts a thirty digit dividend and a fifteen digit divisor and produces a fifteen digit result.

The operands for these functions may be signed or unsigned and leading zeros need not be supplied. Null operands are treated as zero. Results will be signed if negative and will have leading zeros suppressed. A zero result will consist of a single zero digit.

### Add

AD 2,2,V #<AD;num1;num2>

Value: the sum of integers num1 and num2.

Notes: Overflow will produce a number modulo 1015.

### Subtract

SU 2,2,V #<SU;num1;num2>

Value: the value of integers num1 minus num2.

Notes: Overflow will produce a number modulo 1015.

### Multiply

MU 2,2,V #<MU;num1;num2>

Value: the product of integers num1 and num2.

Notes: Overflow will produce a number modulo 1015.

### Divide and Give Quotient

DV 2,2,V #<DV;num1;num2>

Value: the value of integers num1 divided by num2.

Notes: Overflow will produce a number modulo 1015.

A quotient of more than 15 digits will terminate processing and produce an error message.

### Divide and Give Remainder

DVR 2,2,V #<DVR;num1;num2>

Value: the remainder of integers num1 divided by num2

Notes: Overflow will produce a number modulo 1015.

If the operands are such that the quotient would exceed 15 digits, processing is terminated and an error message is produced.

### Obtain Absolute Value

ABS 1,1,V #<ABS;num>

Value: the absolute value of the integer num.

## Numeric Comparisons

The functions in this group provide for the comparison of two signed or unsigned decimal integers, each a maximum of 15 digits in length. The result of such a comparison is one of two strings also provided as parameters to the function. In practice, these strings will often contain function calls to be executed depending on the result of the comparison.

### Compare Numeric “Equal”

EQ 4,4,V #<EQ;num1;num2; string1; string2>

Value: If num1 equals num2 the result is string1; otherwise, it is string2.

### Compare Numeric "Greater Than"

GT 4,4,V #<GT;num1;num2; string1; string2>

Value: If num1 is greater than num2 the result is string1; otherwise, it is string2.

### Compare Numeric "Less Than"

LT 4,4,V #<LT;num1;num2; string1; string2>

Value: If num1 is less than num2 the result is string1; otherwise, it is string2.

## Logical Comparisons

The functions in this group provide for the comparison of two arbitrary strings. Comparison begins from the left and proceeds character by character. If the strings are of unequal length, the shorter string is treated as if it were filled-out with characters which collate low to all other characters.

The EBCDIC collating sequence is used (special characters low to lower case alphabet which is low to upper case alphabet which is low to the decimal digits).

The result of a logical comparison is one of two additional strings which are also provided as parameters. In practice, these strings will often contain function calls which will be executed depending on the result of the comparison.

### Compare Logical “Equal"

EQ? 4,4,V #<EQ?;string1;string2;string3;string4>

Value: If string1 is equal to string2, the result is string3; otherwise, it is string4.

### Compare Logical "Greater Than”

GT? 4,4,V #<GT?;string1;string2;string3;string4>

Value: If string1 is greater than string2, the result is string3; otherwise, it is string4.

### Compare logical "Less Than”

LT? 4,4,V #<LT?;string1;string2;string3;string4>

Value: If string1 is less than string2, the result is string3; otherwise, it is string4.

## Peripheral Input/Output Operations [Interactive]

TTM can read a string from the controlling remote terminal or from the card reader and can type a string on the remote terminal.

The various input functions will read characters, including carriage returns, until a certain special character is detected. The string will then be considered complete. The special character, known as the meta character, can be redefined by the programmer at any time. In the absence of a programmer provided definition, the single quotation mark (') will be used as the meta character.

### Read a String

RS 0,0,V #<RS>

Value: All characters preceding the first occurrence of the current meta character will be returned as the value of this function. Input is normally from the remote terminal, but can be switched to the card reader by means of the RCD function.

Note: By using a passive call on RS, characters will be input without examination. For example, strings containing unbalanced brackets can be entered in this manner.

### Set to Read from Cards

RCD 0,2,S #<RCD; lower; type>

Parameters: If lower is non-null, any upper case alphabetic characters inputed will be translated to lower case. If type is non-null, the incoming cards will be typed on the remote terminal as they are read.

Side Effect: Future calls on the RS function will cause input to come from the card reader. This action will hold until a card with \*EOF in columns 1-4 is read or until the physical end-of-file is detected in the card reader. Note: The end of card will be interpreted as a carriage return. Normal meta character usage applies.

### Print a String

PS 1,1,S #<PS;string>

Side Effects: The string, of any length, will be typed on the remote terminal. If the string is longer than the maximum line length, multiple lines will be typed.

Note: Carriage return characters can be used to force multiple lines.

## Print String and Read

PSR 1,1,VS #<PSR;string>

Side Effects: The string will be typed on the remote terminal in the same manner as that typed by the PS function. However, the carriage will not be returned at the end. The remote terminal will then be placed in the input mode.

Value: All characters typed preceding the first occurrence of the current meta character.

Note: Treated as if it were PS if input has been switched to the card reader.

### Change Meta Character

CM 1,1,S #<CM;string>

Parameter: a string, usually only one character in length.

Side Effect: The first character of the string becomes the new meta character.

Note: the carriage return character can be used as the meta character. When entered, it must be protected by the escape character or by brackets. It is very dangerous to define the meta character to be #, <, >, or ; as once defined, no further redefinitions can be made.

## Formatted Output Operations

The functions in this group are used to format cards to be punched or lines to be printed or typed. Since these lines or cards may be used as input to other language processors (e.g., assemblers, compilers) provision is made for automatic tabulation and generation of continuation cards.

Lines are formated by the FM function and placed in an output buffer.

Any number of lines may be placed in the buffer before it is outputed. Because the printer and punch are shared with other users in a time sharing environment, it is good practice to format as many lines as possible before outputing the buffer.

The maximum size for a line to be punched Is 80 characters and for a line to be typed or printed is 130 characters.

### Format a Line or Card

FM 1,\*,S #<FM;stringy;string^;...>

Side Effects: One line is formated and placed in the output buffer. The first eight strings will be placed starting at the eight tab positions. If this would place a string such that fewer than two blanks separated it from the previous string, it will be placd two blanks after the start of the previous string. If there are more strings than declared tabs, the additional strings will be separated by two blanks. Tabs are effective only on the first line of a line group. If continuation is required, all strings will be separated by two blanks.

### Declare Tab Positions

TABS 1,8,S #<TABS;tab1;tab2;...;tabn>

Parameters: positive decimal integers specifying tab positions and lying in the range: 1 < tabi < tab(i+1) < 130.

Side Effects: The tab positions are saved for use by the FM function.

Note: The tabs apply only to the lines being built by the FM function and are not related in any way to the physical tabs on the remote terminal.

### Set Continuation Convention

SCC 2,2,S # <SCC; num; string>

Parameters: num is a positive decimal integer in the range 1 < num < 130 and string is an arbitrary string which must include a carriage return character.

Side Effect: num is established as the last column into which text will be placed in a line or card. When that column is passed by the FM function, the contents of string (which must contain a carriage return character) are placed in the buffer, thus representing the characters which terminate one card and begin the continuation card. Notes: The carriage return in string must be protected by brackets or an escape character and must occur in or before column 131 for lines or column 81 for cards. In the absence of a specification, the last text column is set to 80 and the continuation string consists of a single carriage return.

Most currently practiced continuation conventions can be supported by this function. For example, the FORTRAN convention of a non-blank character in column 6 of the continuation card could be set by: #<SCC;72;<CRbbbbbX>> where CR stands for the carriage return and b stands for a blank.

### Insert a Control Character

ICC 1,1,S #<ICC;control>

Parameter: control is one of the following characters

0 - 1 +

Side Effects: A control code is inserted in the output buffer. The meanings are

0 = double space

- = triple space

1 = page eject

+ = do not space

Notes: The line printer uses all of these; the remote terminal uses the first three; and the punch uses none. On terminals without page ejection hardware facilities, the page eject code will cause four blank lines to be outputed.

### Output the Buffer

OUTB 0,3,S #<OUTB;device;upper;save>

Parameters: device is TYPE, PRINT, or PUNCH causing output to be directed to that device. If upper is non-null, all lower case alphabetic characters are translated to upper case on output. If save is non-null, the buffer is saved; otherwise, it is emptied after output. If all parameters are null, the buffer is simply emptied.

Side Effects: The contents of the output buffer are processed as described above.

## Library Operations

### [Interactive Description]

TTM supports a library into which a user may store collections of named strings. Such collections are known as programs and each must be named. A standard program name consists of one through twenty-six characters.

In addition to his own programs, a user may wish to refer to programs stored by someone else. To do this, he takes the program name assigned by the user and precedes it with a code of the form: &uid. where uid is the user’s three character identification code. Thus, if user XXX wished to refer to his own program AAA, he would use

AAA

as the name. However, if user ZZZ wanted to refer to this same program, he would use the name

&XXX.AAA

This form of name can be used only with the COPY, and NAMES functions.

The uid should be in upper case.

### [Batch Description]

TTM supports a direct access library into which collections of named strings may be stored. Such collections are known as programs and each must be named. A complete program name consists of an ampersand, a qualifier, a period, and member name. Qualifiers consist of one through eight characters and member names consist of one through twenty characters. For example, &TEST.LOAD would be a complete name consisting of qualifier, TEST, and the member name, LOAD.

When a new program is to be added to the library (STORE function) or an old program is to be deleted (DELETE function), the complete program name must be given. Complete names may also be given when searching the library for an existing program (COPY, NAMES, PDF functions). However, a set of standard qualifiers may be declared (LIBS function) which will automatically be placed, one at a time, in front of the given unqualified member name whenever search is requested. The default qualifier, SYSLIB, will be used in the absence of any declared standard qualifiers. The SYSLIB qualifier will also be used in a search which exhausts the declared qualifiers without finding the desired member.

For example, assume that the following named programs have been stored: &SYSLIB.A, &XX.B, &YY.B, &YY.C, &YY.D. Assume also that YY and XX have been declared standard qualifiers, in that order. Then, a search for B would retrieve YY.B, a search for &XX.B would retrieve XX.B and a search for A would retrieve SYSLIB.A.

In addition to explicit retrieval of programs, TTM supports implicit retrieval. If a function is called which is not defined, the name is assumed to be a library member name and a search is made. If the member is found, its strings are brought into core and defined. If one of these has the same name as the function, it is executed with the original parameters; otherwise, an error message is issued.

### Store a Program

STORE 2,\*,S #<STORE;programname;name1;name2;…>

Parameters: the namei are the names of functions to be stored under the name program name.

Side Effects: The named functions are placed in the library under the specified name. The residual pointers of the named strings are not stored.

### Delete a Program

DELETE 1,1,S #<DELETE;programname>

Side Effects: The named program is deleted from the library.

### Copy a Program

COPY 1,1,S #<COPy ;programname>

Side Effect: The strings stored under the given program name will be copied from the library and put into the TTM memory.

### Show Program Names [Interactive]

SHOW 0,1,S #<SHOW;uid>

Parameters: If supplied, uid is the three-character identification code of some user. It must be in upper case.

Side Effects: If uid is absent, the names of all stored programs belonging to the current user will be typed. If uid is present, the names of all stored programs belonging to the specified user will be typed.

### Obtain String Names

NAMES 0,1,V #<NAMES; programname>

Value: If program name is absent the value will be the names of all strings in the TTM memory, arranged in sorted order and separated by commas. If program name is present, the value will be the names of all strings stored in the library under the given name, separated by commas, and in the order in which they appeared in the original store function.

### Determine if a Program is in the Library [Batch]

PDF 3,3,V #<PDF;programname;string1;string2>

Value: If a program in the library has the given name, the value is string1; otherwise, it is string2.

### Declare Standard Qualifiers [Batch]

LIBS 0,10,S #<LIBS;string^;string2;. • • ;string^>

Parameters: Each string should be one through eight characters in length.

Side Effects: The strings become the standard qualifiers which will be applied, in the order given, whenever an unqualified member name is being searched for. If no parameters are given, only the default standard qualifier will be used.

Notes: Any subsequent calls to LIBS will redefine the whole set of qualifiers.

## Utility Operations

The operations in this group perform certain utility functions which are not easily classifiable into one of the other groups.

### Determine if a Name is Defined

NDF 3,3,V #<NDF;name;stringy;string2>

Value: If a string in the dictionary has been given a name of name, the value is string1; otherwise, it is string2.

### Obtain the Norm of a String

NORM 1,1,V #<NORM;string>

Value: The number of characters in the string will be returned as an unsigned decimal integer with leading zeros suppressed.

### Program Break

BREAK 0,1,S #<BREAK;string>

Side Effect: Processing of the present program string is terminated. If this has a parameter it becomes the new program string, otherwise the Idle program becomes the new program string and in either case processing resumes at the start of the new program string. All information contained in the old program string is lost.

### Obtain Execution Time [Interactive]

TIME 0,0,V #<TIME>

Value: The total execution time since the current session began will be returned as an unsigned decimal integer with leading zeros suppressed representing hundredths of seconds.

### Turn Trace On

TN 0,0,S #<TN>

Side Effect: The trace mode of operation will be activated.

Note: In the trace mode, TTM will type the name of each function, as well as its parameters, just before a function is executed.

### Turn Trace Off

TF 0,0,S #<TF>

Side Effects: The trace mode of operation will be deactivated.

### Return from TTM [Interactive]

EXIT 0,0,S #<EXIT>

Side Effects: TTM operation is terminated and control is returned to the Basic Time-Sharing System.

### Test for End of String [Batch]

EOS 3,3,V #<E0S;name;string1;string2>

Value: If string name, starting at the residual pointer, is null, the value is string1; otherwise, it is string2.

# System Functions for the Batch Version [Batch]

The functions described in the previous sections are available in the base version of TTM. The batch version augments these by a number of additional functions which primarily control input and output operations of the system.

The batch version consists of two logical parts — an input monitor and TTM. The Input monitor scans the input stream and passes control to TTM whenever a TTM program is detected. This is detected by recognizing #< or ##< starting in column one of an input card. While TTM is in control it may call upon the input monitor for more input. If there is no more input, processing is terminated. When a TTM program ends, control is returned to the input monitor. Output may be done by both the input monitor and TTM.

## Monitor Control Functions

### Control Output of Input Monitor

INSW 2,2,S #<INSW;PR;PU>

Parameters: PR and PU are integers.

Side Effects: If PR=0 no input lines are printed; if PR=1 only lines not passed to TTM are printed; if PR=2 all lines are printed. PU takes the same values as PR but controls punching of input lines.

Note: The default values are PR = 2 and PU = 0.

### Control Handling of TTM Programs

TTMSW 2,2,S #<TTMSW;ACT;E0L>

Parameters: ACT and EOL are integers.

Side Effects: If ACT = 0 TTM complete programs are returned by the functions CD, PK, and FOR. If ACT = 1 the current TTM program is terminated and control returns to the input monitor which then calls TTM with the new program. If ACT = 2 TTM programs are given no special significance. If EOL = 0 end of line characters will not be inserted between lines of a TTM program. If EOL = 1 end of line characters will be inserted.

Notes: The default values are ACT = 1 and EOL = 0.

A TTM program is continued by placing a non-blank character in column 72 of the card to be continued. All lending and trailing blanks will be stripped. Columns 73 through 80 are not examined and may be used for any purpose.

## Input Functions

### Input One Card

CD 0,0,V#<CD>

Value: The next card image in the input stream as controlled by the CDSW function.

### Control CD Input

CDSW 2,2,S#<CDSW;SUP,NN>

Parameters: SUP and NN are integers. If SUP = 0, trailing blanks will not be suppressed. If SUP = 1, trailing blanks will be suppressed. NN (1<^NN<^80) specifies the number of columns of a card to be returned by DC.

Notes: The default values are SUP = 1 and NN = 80. Suppression of trailing blanks always begins in column NN.

### Input Next Complete Fortran Statement

FOR 0,0,V#<F0R>

Value: The next card in the input stream is considered to have the form of a Fortran statement and that statement, including any continuation, is returned.

Note: Columns 73 through 80 of all cards and columns 1 through 6 of continuation cards are deleted.

### Control FOR input

FORSW 2,2,S#<F0RSW;SUP;EOL>

Parameters: If SUP = 0 trailing blanks are not suppressed. If SUP = 1 trailing blanks are suppressed. If EOL = 0 end of line characters are not inserted between continuation lines. If EOL = 1 end of line characters are inserted.

Note: The default values are SUP = 1 and EOL = 0.

### Look Ahead One Card

PK 0,0,V #<PK>

Value: The next input card is returned as the value of PK, However, the card is not removed from the input stream and may be read again with the CD or FOR functions.

If the card is a TTM program, the action specified by TTMSW overrides. Successive calls on PK without intervening calls on CD or FOR cause the previously seen card to be processed by the input monitor as if the card had not been seen by TTM.

### Control PK Input

PKSW 2,2,S #<PKSW;SUP;NN>

Note: Has same effects and defaults for PK as CDSW has for CD.

### Output Functions Print a String

PS 1,1,S #<PS;string>

Side Effects: The string, of any length, will be printed. If the string is longer than 132 characters, multiple lines will be printed.

Note: End of line characters can be used to force multiple lines.

### Specify Length of Page

PAGE 1,1,S #<PAGE;NN>

Parameter: NN is an integer specifying the number of lines to be printed per page.

Note: The default value of NN is 60.

### Space Before Printing

SP 1,1,S #<SP;NN>

Parameter: NN is an integer specifying the number of lines to space before printing the next line.

Notes: NN = 1 is normal single spacing and is the default for each line. A space past the end of the logical page causes an eject as does a value of NN of 100 or more. If NN = 0, no spacing will occur before printing and over-printing will result. The effect of a call on SP holds only for the next line printed.

## Formated Output Operations

The functions in this group are used to format cards to be punched or lines to be printed. Since these lines or cards may be used as input to other language processors (e.g., assemblers, compilers) provision is made for automatic tabulation and generation of continuation cards.

### Format a Line or Card

FM 0,\*,S #<FM;string1;string2;...>

Side Effects: One line is formated and printed or punched. The first ten strings will be placed starting at the ten tab positions. If this would place a string such that no blanks separated it from the previous string, it will be placed one blank after the end of the previous string. If there are more strings than declared tabs, the additional strings will be separated by one blank. Tabs are effective only on the first line of a line group. If continuation is required, all strings in the continuation lines will be separated by one blank.

Note: If an end of line character is detected in one of the parameters, continuation will be forced at that point.

### Declare Tab Positions

TABS 1,10,S #<TABS;tab1;tab2;. . .; tabn>

Parameters: positive decimal integers specifying tab positions and lying in the range 1 < tabi < tab(i+1)

Side Effects: The tab positions are saved for use by the FM function,

Note: The default action of FM is as if #<TAB;1> had been issued. Tabs past the end of the line will cause continuation.

### Set Continuation Convention

SCC 3,3,S #<SCC;LCH;END;ST>

Parameters: LCH (1 <= LCH <= 132) is an integer specifying the position on a line after which a continuation line is produced. END is a string which will be placed at the end of a line to be continued. ST is a string which will be placed at the beginning of a continuation line.

Notes: The default values are LCH = 132, END = null, and ST = null. As an example, the FORTRAN continuation conventions could be established by coding

#<SCC;72;;bbbbbC> where b means "blank".

### Control FM Output

FMSW 2,2,S #<FMSW;PR;PU>

Parameters: If PR = 0 output from FM is not printed; if PR = 1 output from FM is printed. If PU = 0 output from FM is not punched; if PU = 1 output from FM is punched. Notes: The default values are PR = 1 and PU - 0. Output lines longer than 80 characters will be truncated to 80 characters before they are punched.

## Utility Functions

### Obtain Time of Day

Time 0,0,V #<TIME>

Value: The time of day, in hundredths of seconds since midnight, will be returned as an unsigned decimal integer.

### Obtain Execution Time

XTIME 0,0,V #<XTIME>

Value: The current execution (task) time for the job, in hundredths of seconds, will be returned as an unsigned decimal integer.

### Define Error String

DES 1,1,S #<DES;string>

Side Effect: The string is considered to be a TTM program which will be called whenever a processing error occurs.

# Examples

Since TTM is oriented toward systems programming it is natural to write some TTM functions which will make TTM programming easier.

## Function Definition

The most basic of these functions is def. It is written as:

#<ds;def;<##<ds;name;<text>>##<ss;name;subs>>>

and

#<ss;def;name;subs;text>

To define the string XX as 12345 and then segment XX on 34, write:

#<def;XX;34;12345>

The call

#<XX;0000>

will get

1200005

This function is so convenient that it will be used in presenting the following examples.

In addition to defining and segmenting a string it is frequently desirable to mark it for name creation also. This can be done with defcr which is defined using def as:

#<def;defcr;<name;subs;crs;text>;<##<ds;name;<text>>##<ss;name;subs>##<cr;name;crs>>>

This description in the "def" format is equivalent to:

#<ds;defcr;<##<ds;name;<text>>##<ss;name;subs>##<cr;name;crs>>>

followed by

#<ss;defcr;name;subs;crs;text>

### Basic Examples

##### Example 1.

A function which, given the name of a string, will return that string with a plus sign following each character but the last can be defined as:

#<def;plus;SN;<#<cc;SN>#<plusx;SN;#<cc;SN>>>>

and

#<def;plusx;<SN;CH>;<#<eq?;CH;; ;<+CH#<plusx;SN;#<cc;SN >> >>>>

Writing functions in pairs with one function to start the process and another to extend it for a number of iterations is quite useful.

Given X = ABCD then #<plus;X> will yield A+B+C+D

##### Example 2.

A function which given a number N will return the power of 2 which is equal to or next greater than N can be defined as:

#<def; power ;N; <#<pw;l; #<su;N;l>> >>

and

#<def ;pw;<pp;XX>;<#<eq;XX;0;pp;<#<pw;#<mu;pp;2> ;#<dv;XX;2 >>>>>>

when called as

#<power;12>

it will return 16.

##### Example 3.

A function to convert a decimal number into a binary number is defined as:

#<def;binary;<N;bnum>;<#<eq;N;0;bnum;<#<binary;##<dv;N;2>;##<dvr;N;2>bnum>>>>>

The call

#<binary;37>

will yield 100101.

##### Example 4.

A function to construct a table of length 2n which contains in each position the count of the number of one-bits in the binary number representing that position can be defined as:

#<def;bitct;<LL;NN>;<#<eq;LL;l;(NN);<#<bitct;##<dv;LL;2>;HN>

#<bitct;##<dv;LL;2> <ad;NN;l>>>>>>

To see how this function works, note that a table of length 2n can be constructed by appending to a table of length 2 the same table with all entries Increased by one.

If this function is called as:

#<bitct;8;0>

it will yield this.

(0)(1)(1)(2)(1)(2)(2)(3)

##### Example 5.

Ackerman’s function is a recursive function which grows more rapidly than any primitive recursive function.

The function is defined as follows.

A(m,n) = A(m-l,A(m,n-l))

A(m,0) = A(m-l,l)

A(0,n) = n+1

The TTM function to evaluate Ackerman’s function for some M and N is:

#<DEF;A;<M;N>;<#<EQ;M;0;<##<AD;N;1>>;

<#<EQ;N;0;<#<A;#<SU;M;1>;1>>;

<#<A;##<SU;M;1>;#<A;M;##<SU;N;X>>>>>>

When called as

#<A;3;1>

it will have the value 13.

##### Example 6.

The following is an example of a function which creates another function and then executes the created function. The purpose of this function is to duplicate a section of text and increment numbers within the text. The first function is defined as:

#<DEFCR;DO;<FROM;BY;TO;SUBS;TEXT>;NN;

<#<DEF;DO-NN;SUBS;#@<#<GT;BY;0;GT;LT><;SUBS;T0;;<TEXT#<DO-NN

##<AD;SUBS;BY>>>>@>>#<DO-NN;FROM>##<ES;DO-NN>>>

The call:

#<DO;l;2;7;K;(K)>

will first define a function with a created name. This function would be

#<DEF;DO-0005;K;<#<GT;K;7;;<(K)#<DO-0005;##<AD;K;2>>>>>>

The function DO uses the escape character instead of brackets around all of the text of the DEF because it needs to evaluate the GT to see if the increment is going up or down.

Now the newly created function will be called as:

#<DO-0005;1>

and it will produce

(1)(3)(5)(7)

If DO is called as

#<DO;-2;3;8;AA;XXAA>

it will produce

XX-2 XX1 XX4 XX7

If DO is called as

#<DO;1;1;3;I;#<DO;2;-1;1;J;(I, J)>>

it will produce

(1,2)(1,1)(2,2)(2,1)(3,2)(3,1)

If DO is called as

#<DO;0;l;4;AA;<(#<mu;AA;AA>)>>

it will produce

(0)(1)(4)(9)(16)

Since DO always erases the functions it creates it does not fill memory with useless strings.

### Keyword Parameters

The following example demonstrates the writing of a TTM function which can be called with keyword parameters.

Construct a function, FF, which will produce:

(aa) (bb) (cc) (dd) (ee)

when it is called with aa, bb and cc specified by position; and with dd and ee specified by the keywords X= and Y= respectively. In addition, dd is to have the default value 17 if it is not specified.

For example if FF is called as

#<FF;1;2;3;Y=5;X=4>

it should produce

(1)(2)(3)(4) (5)

or if it is called as

#<FF;10;Y=25>

it should produce

(10)()()(17)(25)

Such a function is constructed as two functions. The first, FF, is the one which is called. The second, FF.k, is the one which actually produces the result.

The function FF will be defined as:

#<def;FF;<pl;p2;p3;p4;p5>;

<##<ds;KW;<;pl;p2;p3;p4;p5>>#<FF.k;#<KWl;X;17>;#<KWl;Y;>#<KW>>>>

This function expects from 0 to 5 parameters with the keyword parameters following the positional parameters. It will rearrange these into positional parameters using the keywords and default text it contains and will call FF.k with these parameters.

The function FF.k will be defined as:

#<def;FF.k;<dd;ee;aa;bb;cc>;<(aa)(bb)(cc)(dd)(ee)>>

It is important to note that in the call to FF.k the two parameters which were specified by keywords will come before the three parameters defined by position.

In addition to FF and FF.k functions KW1 and KW2 will be needed. These functions can be shared by all functions having keyword parameters. These will be described later.

The operation of the keyword functions is as follows. First the parameters of FF are defined into a string, KW, with semicolons before each parameter including the first. Next, KW1 is called with a keyword and default text. KW1 will return the keyword parameter. This is done for each keyword. When all of the keywords have been searched for, the string KW will contain only the positional parameters separated by semicolons. An active call on KW is used to finish the parameter list for FF.k and a call on FF.k is made.

An active call on KW is used to break it into parameters. A semicolon is not put between the last call on KW1 and the call on KW because the first character of KW will be a semicolon. Since the keyword parameters are now before the positional parameters, FF.k should be defined accordingly.

The function KW1 is defined as

#<def ;KW1;<key;dflt>;

<#<eq;##<sc;KW;<;key=>>;0;<dflt>;<#<KW2;##<cs ;KW>;##<cp;KW>;##<KW>>>>>>

This function first segments the string KW on ;key= . If the key is not in the string, that is if SC returns 0, then KW1 returns the default text.

If the key was in the string, it calls KW2 with that part of the string KW up to the key, the keyword parameter, and the remaining part of the string KW. It is very important to use passive calls on ##<cs;KW> and ##<KW> because the strings that these functions return may have semicolons in them. However, they are to be passed to KW2 without being broken apart.

The function KW2 is defined as:

#<def;KW2;<first;parm;last>;<parm##<ds;KW;<first;last>>>>

it returns the keyword parameter and redefines KW to be what remains of the

parameter string after ;key=parm has been removed. The primitive SC emoved

;key= and CP removed parm; so a semicolon must be put between first and last.

A series of functions can be defined which will aid in producing pairs of functions like FF and FF.k. These functions are given below. They are called by:

#<def kw;name;subs;keys; tex t>

where name, subs, and text are the same as they are in def and keys has the form

<keyword=default;----;---->

So to produce FF and FF.k we would call defkw as:

#<defkw;FF;<aa;bb;cc>;<X=17;Y=>;<(aa)(bb)(cc)(X)(Y) >>

It is important to note that there are two independent sets of functions here. Those beginning with defkw are used when we are defining a function which can be called with keywords. The two functions KW1 and KW2 are needed when we actually call such a function.

#<def;defkw;<name;subs;keys;text>;<##<ds;key;<keys>>

#<dkwl;#<dkw2>>#<rrp;keyl>#<rrp;key2>#<rrp;key3>

#<dkw3;<name>;<subs>;##<keyl>;##<key2>>

#<dkw4;<name>;<eubs>;##<key3>;<text>>

#<es;key;keyl;key2;key3;key4>>>

#<def;dkwl;<keyw;dflt>;<#<eq?;keyw;;;<##<ap;keyl;<keyw\*;>>

##<ap;key2;<;#<KW1;keyw;df11>>>##<ap;key3;<keyw;>>#<dkwl;#<dkw2>>>>>>

#<def;dkw2;<>;<##<ds;key4;##<cp;key>>##<ss;key4;=>

##<cs;key4>;##<key4;->>>

#<def;dkw3;<name;subs;keys;calls>;<#<def;<name>;<keyssubs>;

<##<ds;KW;<;keyssubs>>#<name,kcalls#<KW>>>>>>

#<def;dkw4;<name;subs;keys;text>;<#<def;<name,k>;

<keyssubs>;<text>>>>

### A TTM Operating System

As another example of the way a TTM programmer can create an "operating system" for himself, examine the functions Def and Modify.

Def is written as:

#<def;Def;<name;subs;text>;

<#<def;name;<subs>;<text>>##<ds;$$name;<#<def;name;<subs>;<text>>>>>

If a function is written using Def instead of def, the function will be defined and it will be saved under the name $$name as an identical copy of the call on def.

The importance of the copy of the call on def can be seen by examining the functions Modify, Mod-1, and Mod-2.

The Modify function is defined as:

#<def;Modify;name;<##<ps ;#<$$name>>#<Mod-l;$$name>>>

When the user calls Modify with the name of a function created using Def, it first types the original call and then calls Mod-1 with the name of the string it just typed.

The passive call on $$name is necessary because the entire string is needed as a parameter of PS.

The Mod-1 function is defined as:

#<def;Mod-l;NAME;<##<ss;NAME;

##<psr;delete? >>##<ds;NAME;

##<NAME;##<psr;insert? >>>

##<ps;##<NAME>>#<Mod-2;NAME;

##<psr ;modify? >>>>

The parameter NAME in Mod-1 is the name of the definition copy. That is, it is $$name. This string was typed in Modify. The string should now be segmented on the characters to be modified.

When Mod-1 is executed, it types the message “delete? “ and issues a read string without returning the carriage. The user will now type the characters to be deleted. These can be any characters because a passive call was made on psr and the characters will, therefore, not be rescanned. For instance,

>>>> might be deleted.

The string NAME can now be redefined with the deleted characters replaced by the characters which will be inputted by the user after the Mod-1 function presents the message “insert?”. Again, these characters will not be rescanned so they can be anything. The modified version of the string NAME will be typed so that the user may examine it for corrections. Next, the user is asked if any more modifications are to be made; and Mod-2 is called with NAME and the answer the user gave.

The Mod-2 function is defined as:

#<def;Mbd-2;<NAME;answer>;

<#<eq?;answer;yes;<#<Mod-l;NAME>>;

<#<NAME>>>>

If the answer is “yes”, Mod-1 is called again. Otherwise, an active call is made on NAME. This will cause the def to be executed thereby creating a new function name with the proper changes.

For example, define the string XX using Def.

#<Def;XX;5;123456>

The call

#<XX;ZZ>

will give

1234ZZ6

To modify XX, call Modify:

#<Modify;XX>

The conversation will appear as:

#<def;XX;<5>;<123456>>

delete? <5

insert? <5;6

#<def;XX;<5;6>;<123456>>

modify? No

Finally, a call on XX,

#<XX;YY;ZZ>

will give

1234YYZZ

It was necessary to use <5 so that the function could determine where the change was wanted. A deletion of >; and insertion of ;6>; would have produced the same results.

### Expression Evaluation

The following TTM functions form a recursive expression evaluator. This expression evaluation will handle expressions containing unary ‘+’and ‘-‘ and any level of parentheses.

It is entered by calling Eval with an expression as the parameter. The Eval function is defined by:

#<def;Eval;EXP;<##<ds;exp;EXP>#<sum>>>

The sum function will return the value of the expression. The other functions are:

#<ds;sum;<#<suml;#<term>;#<op>>>>

#<def;suml;<num;PR;oper>;

<#<eq?;PR;SS;<#<suml;##<oper;num;

#<term>>;#<op>>>;<num>>>>

#<ds;term;<#<terml;#<prime>;#<op>>>>

#<def;terml;<num;PR;oper>;

<#<eq?;PR;TT;<#<terml;##<oper ;num; #<prime>>;#<op>>>;<num>>>>

#<ds;prime;<#<tcl;digit;exp;

<##<ccl;digit;exp>

#<ds;op;##<pr1##<cc;exp>>>>;

<#<do##<cc;exp>>>>>>

#<dcl;digit;0123456789>

#<ds;do;0>

#<ds;do+; <#<prime>>>

#<ds;do-;<##<su;0;#<prime >> >>

#<ds;do(; <#<sum>//#<ds; op;

##<pr|;##<cc;exp>>>>>

In addition to these functions there is also what might be thought of as a translate table. That is, the string pr|+ is defined to be ss;ad. The

complete table is

pr|+ SS;ad

pr|- SS;su

pr|\* TT;mu

pr|/ TT;dv

pr|)

pr|

the last two entries having null values.

It can be seen that this expression evaluator follows closely the BNF description of an expression:

<sum>::=<term>) <sum>+<term> j <sum>-<term>

<term>::=<prime>|<term>\*<prime>|<term>/<prime>

<prime>::=<integer> | + <prime> | -<prime> | (<sum>)

<integer>::=<digit> | <integer><digit>

<digit>::=0|l|....|9

In suml and terml the class of operator is tested instead of testing for both operators separately. The class of the operator and its TTM name are in the string op so an active call on op is needed to separate them. However when translating the operator into its class and name and putting it into the string op, it is very important to do a passive call on the string pr|operator which does this translation. Otherwise, only the class will be in the string op.

Also note that in priml when it is determined that a character is not a digit, a four way branch is made to the strings which process the unary operators, left parentheses, and null expressions.

# Using the Conversational System

The TTM processor operates as a problem program under the Caltech Basic Time Sharing System (BTSS) [13]. It resides on a direct access device along with its library of user-written TTM programs.

## Initiating a Session

To start a session from a 2741 remote terminal, perform the following operations:

1. Turn the terminal power on
2. Place the terminal in the "communicate” mode
3. Press the attention (ATTN) button

The system will respond with an identifying message and will then type:

ENTER NAME AND NUMBER This is a request for the user identification code and account number. These should be entered, separated by a comma or a blank. If they are correctly verified, the system will respond with READY

## Calling for TTM

TTM is called by typing:

EXECUTE DISK TTM RECORD 1 When TTM is loaded, it will respond with an identifying message giving version number and version creation date. TTM is then ready for use.

## Terminating Session

To terminate a session, it is first necessary to exit from TTM. This can be accomplished by executing the EXIT function. The session can then be terminated by the STOP statement.

## Upper and Lower Case

In TTM, upper and lower case letters are treated as completely different characters. However, each of the built-in functions may be called by upper case or lower case names. For example, the addition function may be called as AD or as ad.

## General Error Correction

Typing errors can be corrected by changing part of a line cr by deleting the line and re-entering it correctly. These corrections apply only to the line which is currently being typed (i.e., before the carriage return key is pressed).

The backspace and forward space keys operate as on a normal typewriter and cause the carrier to move back and forth across the line. Previously typed characters are not erased by this action. Overstriking a character causes it to be replaced by the new character. To change an existing character to a blank, position the carrier at that character and press the "cent-sign (¢) key.

The entire line can be erased by typing a backspace followed by pressing the attention (ATTN) key.

## Entering a TTM Program

When control is passed to TTM from BTSS, TTM performs certain initializing operations and then loads the program string with the idle program.

The idle program consists of:

#<PS;#<RS>>

The execution of this causes TTM to read from the remote terminal until a meta character is seen, execute what was read because an active call was used on RS, type the results (if any), and stop. The act of stopping, however, causes the idle program to be reloaded into the program string thus restarting the process.

## The Attention (ATTN) Key

The attention key is used mainly to stop a “run-away” program. When the terminal is printing or idle (not printing, keyboard locked), pressing the attention key causes a “session break.” TTM processing will be suspended and control will be sent to the command language processor (CLP) of DTSS. To return to TTM at the exact point of interruption, type CONTINUE.

To return to TTM and cause the idle program to be reloaded immediately, type BREAK.

In addition, return by the BREAK command will cause the TTM trace mode to be deactivated if it had been activated previously.

For convenience, the CONTINUE command may be abbreviated as C and the BREAK command may be abbreviated as B

To cause a session break when the keyboard is unlocked, type a pound-sign (#) followed by pressing the attention key.

## System Commands

When TTM is not in control (e.g., during a session break) the following commands may be entered:

1. SHOW CLOCK

The system will display the date, time of day, elapsed time, and compute time.

2. SET TABS

The system will request that the physical tab positions be defined and will then unlock the keyboard. The user should then enter a line containing non-blank characters in each position that a tab is desired. The physical setting of the tab stops must be performed by the user. Physical tab settings are used only for input.

## Color Shift

Terminals have a two-color ribbon. All output from the command language processor of BTSS is typed in red. All user input and all TTM output is typed in black.

# Using the Batch Version [Batch]

The batch version of TTM consists of one load module in standard OS/360 format. It is designed to execute under any version of OS/360 which can provide a region of at least 56K for execution. With this size region, a work area of approximately 10K bytes is available. This can be expanded by increasing the region size as TTM performs a variable GETMAIN to obtain all available care for its own use. Sufficient care is returned to the system so that abnormal terminations will produce a core dump if a data set with a DDNAME of SYSUDUMP is provided.

TTM uses four data sets — SYSIN, SYSLIB, SYSPRINT, and SYSPUNCH. All, except SYSLIB are standard sequential sets and QSAM is used to process them.

The library, SYSLIB, must reside on a direct access device and is processed at the EXCP level. However, physical unit independence is maintained by obtaining device characteristics of the library through use of the DEVTYP system service. Before first use, the library must be initialized by a special utility program.

TTM can process blocked SYSIN, SYSPRINT, and SYSPUNCH. DCB exit routines are utilized for SYSPRINT and SYSPUNCH to provide automatic default blocking of these data sets. In the absence of any over-riding DCB parameters, SYSPRINT is assigned BLKSIZE<L596 and RECFM=FBSM. The defaults for SYSPUNCH are BLKSIZE=1680 and RECFM=FBS. Either BLKSIZE or RECFM or both can be over-ridden for each data set. However, RECFM for SYSPRINT will always be forced to contain an M to indicate machine control codes. Block sizes need not be specified exactly as they are always forced down to an exact multiple of the appropriate LRECL. RECFM is automatically adjusted to indicate whether the records are blocked or not.

It is quite acceptable to specify DISP-SHR in the DD card for SYSLIB, even in an MFT or MVT environment. TTM uses the system ENQ and DEQ facilities to insure library integrity during all storing and deleting operations.

The following is sample JCL for an execution of TTM. The processor is assumed to reside on a data set named EKG.TTMLOAD and the library name is SHC.TTMLIB. The JCL may need to be modified to reflect varying installation requirements:

//NAME JOB ACCOUNTING PARAMETERS

//JOBLIB DD DSN=EKG, TTMLOAD, UNIT=SYSDA, DISP=SHR,

// V0L=SER=CITSC5

//STEP EXEC PGM=TTMSA

//SYSLIB DD DSN=SHC, TTMLIB, DISP=SHR

//SYSPRINT DD SYSOUT=A,UNIT=(,SEP=SYSLIB)

//SYSPUNCH DD SYSOUT-B,UNIT=(,SEP=(SYSLIB,SYSPRNT))

//SYSIN DD \*

# Appendix A Error Messages

Whenever an error occurs, TTM prints an error message and the name and parameters of the function being processed. It then returns to its caller: [Interactive]the idle program [Batch] the input monitor. The contents of the program string are lost but any side effects which occurred before the error will remain.

The following list gives all of the error messages and a short description of those which are not completely self-explanatory. In this discussion, the terms “primitive” and “built-in function” are synonymous.

#### Function Name Not Found

Either the function which is being called is not in the memory; or if it is a primitive which has a function name as a parameter then that function is not in the memory.

#### Primitives Not Allowed

Some primitives which take function names as parameters will not operate on primitives.

#### Too Few Parameters Given

Many primitives have a minimum number of parameters which they must have to operate.

#### Incorrect Format

A parameter does not have the format that the primitive expects.

#### Quotient Is Too Large

The quotient in a division operation is too large to be represented.

#### Decimal Integer Required

Some primitives require certain parameters to be decimal integers.

#### Too Many Digits

An input parameter contained more than 15 decimal digits.

#### Too Many Segment Marks

The number of segment marks has exceeded 62. The function will contain the first 62 such marks.

#### Dynamic Storage Overflow

No space left in core. There may be room enough to do an Erase String.

#### Parm Roll Overflow

Discussed in Appendix C.

#### Input Roll Overflow

Discussed in Appendix C.

#### Name Already On Library

An attempt has been made to store a program on top of an existing one.

The existing program can be deleted.

#### Name Not On Library

A call to COPY or NAMES requested a non-existent program.

#### No Space On Library

A delete will create space.

#### Initials Not Allowed [Interactive]

Other users initials cannot be used in STORE and DELETE.

#### Could Not Attach [Interactive]

An external device could not be attached.

#### No Library [Batch]

The library data set was not provided.

#### Incorrect Library [Batch]

The library data set does not have the format of a TTM library. Wrong Name Format

Name is not &QUAL.NAME or NAME.

#### Too Many Lib Names [Batch]

More qualifiers provided than can be used.

#### Lib Name Too Long [Batch]

Qualifier Is too long.

#### Error on Library [Batch]

An error was detected in format of library.

#### Too Many Nested Copies [Batch]

The library routine supports nested copies, this facility is not used by TTM but this message could appear if TTM was called during a copy by an outside routine sharing the library.

#### Zero Level Only [Batch]

Deletes cannot be nested inside copies.

### Only Unsigned Decimal Integers [Batch]

The numbers used in functions like TABS, CDSW, etc. must be unsigned decimal integers.

#### An I/O Error Occurred

An external device has given a read/write error.

#### A TTM Processing Error Occurred

This message is issued when a machine or software error is detected.

It is followed by a listing of the PSW and general purpose registers.

#### Error In Storage Format

Another type of TTM processing error.

# Appendix B Function Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Function Name | Min Args | Max Args | Remarks |
| abs | 1 | 1 | V |
| ad | 2 | 2 | V |
| ap | 2 | 2 | s |
| break | 0 | 1 | s |
| cc | 1 | 1 | vs |
| ccl | 2 | 2 | vs |
| cf | 2 | 2 | s |
| cm | 1 | 1 | s |
| cn | 2 | 2 | vs |
| copy | 1 | 1 | s |
| cp | 1 | 1 | vs |
| cr | 2 | 2 | s |
| cs | 1 | 1 | vs |
| dcl | 2 | 2 | s |
| delete | 1 | 1 | s |
| dncl | 2 | 2 | s |
| ds | 2 | 2 | s |
| dv | 2 | 2 | V |
| dvr | 2 | 2 | V |
| ecl | 1 | \* | s |
| eq | 4 | 4 | V |
| eq? | 4 | 4 | V |
| es | 1 | \* | s |
| exit | 0 | 0 | s |
| fm | 1 | \* | s |
| gn | 2 | 2 | V |
| gt | 4 | 4 | V |
| gt? | 4 | 4 | V |
| icc | 1 | 1 | s |
| lsc | 4 | 4 | vs |
| lt | 4 | 4 | V |
| lt? | 4 | 4 | V |
| mu | 2 | 2 | V |
| names | 0 | 1 | V |
| ndf | 3 | 3 | V |
| norm | 1 | 1 | V |
| outb | 0 | 3 | s |
| PS | 1 | 1 | s |
| psr | 1 | 1 | VS |
| red | 0 | 2 | s |
| rrp | 1 | 1 | s |
| rs | 0 | 0 | V |
| sc | 2 | 63 | vs |
| scl | 2 | 2 | s |
| sc | 2 | 2 | vs |
| scn | 3 | 3 | vs |
| show | 0 | 1 | s |
| ss | 2 | 63 | s |
| store | 2 | \* | s |
| su | 2 | 2 | v |
| tabs | 1 | 8 | s |
| tcl | 4 | 4 | v |
| tf | 0 | 0 | s |
| time | 0 | 0 | V |
| tn | 0 | 0 | S |
| zlc | 1 | 1 | v |
| zlcp | 1 | 1 | V |

# Appendix C Storage Overflow

The following simple programs show the kinds of overflow which are possible with the present implementation of TTM.

The first example is

#<ds ;X;<#<X >>>#<X>

This program defines the string X. to be an active call on the string X and then does an active call on X. This program is interesting because it does not cause an overflow. It is an infinite loop. Any program which ends in a call on itself is such a loop unless it creates strings which it does not erase. Such programs are of particular importance as idle programs for conversational programs written in TTM.

The program:

#<ds;X;<#<X>Z >>#<X>

fills the program string to the right of the scan position with ZZZ...Z and will eventually cause an input roll overflow.

This occurs since a call on X appears before all of the characters produced by the previous call on X have been scanned. With the present implementation, X will be called about 300 times before the overflow occurs.

The program:

#<ds;X;<#<Z#<X >> >>#<X>

fills the program string to the left of the scan position with {Z{Z{Z…>>{Z, and fills the program string to the right of the scan position with >>>...>. It will eventually cause a parameter roll overflow. In this function, X is called before the right bracket of the call on Z is reached. The function X will be called about 300 times before the overflow occurs.

The program:

#<ds;X;<Z#<X>>>#<X>

will fill the program string to the left of the scan position with ZZZ. - .Z and will eventually cause a dynamic storage overflow. This overflow will not occur until the entire core is filled with ZZ.. .Z. It does not cause a parameter roll overflow because it is not producing uncompleted function calls, but only a string of characters <>

The program

#<def ;X;Z;<#<X;ZZ>>>#<X;Z>

will also cause a dynamic storage overflow by filling the program string but does it in a quite different way. The single parameter of X doubles in length each time X is called and eventually becomes too large for core.

It is also possible to get a dynamic storage overflow by defining too many strings.

# Bibliography

1. Greenwald, I.D. and Kane, M,, The Share 709 System: Programming and Modification. JACM 6 No. 2 (1959). pp. 128-133.
2. Greenwald, I.D\*, Handling Macro Instructions. CACM 2, No. 11 (1959), 21-22.
3. Remington Rand UNIVAC Division, UNIVAC Generalized Programming. Philadelphia, 1957.
4. Eastwood, D.E. and Mcllroy, M.D., Macro Compiler Modification of SAP. Bell Telephone Laboratories Computation Center, 1959.
5. McClure, R.M., Description of CODAPT Assembler, 1960.
6. Caine, S.H., Reference Manual for CIT 7090/7040 Experimental Macro Assembly Program (XMAP). California Institute of Technology, Willis H. Booth Computing Center (1964).
7. Mcllroy, M.D., Macro Instruction Extensions of Compiler Languages. CACM 3, No. 4 (1960), 214-220.
8. Mcllroy, M.D., Using SAP Macro Instructions to Manipulate Symbolic Expressions. Bell Telephone Laboratories Computation Center (1960).
9. IBM, System/360 Assembler Language, C28-6514-4, (1967).
10. Caine, S.H. et al., Report of the Systems Objectives and Requirements Committee, SHARE, 1965, pp. 29-40.
11. Struachey, C., A General Purpose Macro Generator. Comput J 8, 3(1965), pp. 225-241.
12. Farber, D. J., 635 Assembly System - GAP. Bell Telephone Laboratories Computation Center (1964).
13. Caine, S.H., et al., An Operating Environment for Programming Research. California Institute of Technology, Willis H. Booth Computing Center Programming Report No. 1, 1967.
14. Caine, S.H. and Gordon, E.K., TTM: An Experimental Interpretive Language. California Institute of Technology, Willis H. Booth Computing Center, Programming Report No. 7, 1968.
15. Caine, S.H. and Gordon, E.K., TTM: A Macro Language for Batch Processing. California Institute of Technology, Willis H. Booth Computing Center, Programming Report No. 8 May, 1969.