CALIFORNIA INSTITUTE OF TECHNOLOGY

TTM: AN EXPERIMENTAL INTERPRETIVE LANGUAGE

by

Stephen H. Caine

E. Kent Gordon

Willis H. Booth Computing Center

Programming Report No. 7

July 1968

TABLE OF CONTENTS

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, TTM 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 TTM, 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 TTM 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 character

In certain contexts the special characters

< > # ; @ carriage return

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:

$$\#\langle s_1;s_2;s_3;\ldots;s_n\rangle$$

where the s_i are strings and $n \geq 1$. The first string, s_1, 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:

$$\#\#\langle s_1;s_2;s_3;\ldots;s_n\rangle$$

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 ^. 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: { < < > >>
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;n_1;n_2>

and results in a value which is the sum of the numbers n_1 and n_2.

```
Given:    #<PS;#<ad;1;2>>
          ↑

Yields:   {PS_^#<ad;1;2>>
              ↑

          {PS_^{ad_^1_^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 n_1,n_2,r

where name is the function name as used in the call;

n_1 is the minimum number of parameters required;

n_2 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.

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;$name_1$,$name_2$,...>

Parameter: the $name_i$ 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 built-in function, then new-name will be defined as if it were also that built-in function.

Segment a String

SS 2,63,S

#<SS;name;S_1;S_2;...;S_n>

Parameters: name is the name of a string.

The S_i, $1 \leq i \leq 62$, are strings.

Side Effects: All occurrences of the S_i 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;S_1;S_2;...;S_n>

Parameters: name is the name of a string.

The S_i, $1 \leq i \leq 62$, are strings.

Side Effects: All occurrences of the S_i 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.

Initial Character Scan

ISC 4,4,VS

#<SCN;$string_1$;name;$string_2$;$string_3$>

Value: $string_2$ if the characters of $string_1$ match the characters of the string name which immediately follow the residual pointer

$string_3$ 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;$string_1$;name;$string_2$>

Value: If the string $string_1$ 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 $string_2$ is returned.

Side Effect: If $string_1$ 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. 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.

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

Note: 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;clname$_1$;clname$_2$;...>

Side Effect: The classes named by the clname$_i$ 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;string$_1$;string$_2$>

Value: If the character at the position in string name marked by the residual pointer belongs to the class clname, the value is $string_1$, otherwise it is $string_2$.

Note: If name is null the value will be $string_2$.

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;num_1;num_2>

Value: the sum of integers num_1 and num_2

Notes: Overflow will produce a number modulo 10^{15}.

Subtract

SU 2,2,V

#<SU;num_1;num_2>

Value: the difference between integers num_1 and num_2

Notes: Overflow will produce a number modulo 10^{15}.

Multiply

MU 2,2,V

#<MU;num_1;num_2>

Value: The product of integers num_1 and num_2.

Divide and Give Quotient

DV 2,2,V

#<DV;num_1;num_2>

Value: the quotient resulting from dividing num_1 by num_2

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

Divide and Give Remainder

DVR 2,2,V

#<DVR;num_1;num_2>

Value: the remainder resulting from dividing num_1 by num_2

Notes: 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>

Parameter: A signed or unsigned decimal integer of up to 15 digits.

Value: the absolute value of 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;num_1;num_2;$string_1$;$string_2$>

Value: If num_1 equals num_2, the result is $string_1$; otherwise, it is $string_2$.

Compare Numeric "Greater Than"

GT 4,4,V

#<GT;num_1;num_2;$string_1$;$string_2$>

Value: If num_1 is greater than num_2, the result is $string_1$; otherwise, it is $string_2$.

Compare Numeric "Less Than"

LT 4,4,V

#<LT;num_1;num_2;$string_1$;$string_2$>

Value: If num_1 is less than num_2, the result is $string_1$; otherwise, it is $string_2$.

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?;$string_1$;$string_2$;$string_3$;$string_4$>

Value: If $string_1$ is equal to $string_2$, the result is $string_3$; otherwise, it is $string_4$.

Compare Logical "Greater Than"

GT? 4,4,V

#<GT?;$string_1$;$string_2$;$string_3$;$string_4$>

Value: If $string_1$ is greater than $string_2$, the result is $string_3$; otherwise it is $string_4$.

Compare Logical "Less Than"

LT? 4,4,V

#<LT?;string$_1$;string$_2$;string$_3$;string$_4$>

Value: If string$_1$ is less than string$_2$ the result is string$_3$: otherwise it is string$_4$.

PERIPHERAL INPUT/OUTPUT OPERATIONS

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.

FORMATED 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;string$_1$;string$_2$;...>

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 continua-

tion is required, all strings will be separated by two blanks.

Declare Tab Positions

TABS 1,8,S

#<TABS;tab_1;tab_2;...;tab_8>

Parameters: positive decimal integers specifying tab positions and lying in the range

$$1 \leq tab_i \leq tab_{i+1} \leq 130$$

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 $1 \leq num \leq 130$ 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;<(CR)bbbbbX>>

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

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.

Store a Program

STORE 2,*,S

#<STORE;programname;$name_1$;$name_2$;...>

Parameters: the $name_i$ 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

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.

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;$string_1$;$string_2$>

Value: If a string in the dictionary has been given a name of name, the value is $string_1$; otherwise, it is $string_2$.

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 is 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

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

EXIT 0,0,S

#<EXIT>

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

EXAMPLES

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

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

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;1;#<su;N;1>>>>

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

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

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

The call

```
#<binary;37>
```

will yield 100101

4. A function to construct a table of length 2^n 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;1;
 (NN);<#<bitct;##<dv;LL;2>;NN>
#<bitct;##<dv;LL;2>;##<ad;NN;1>>>>>>
```

To see how this function works, note that a table of length 2^n can be constructed by appending to a table of length 2^{n-1} the same table with all entries increased by one.

If this function is called as:

```
#<bitct;8;0>
```

it will yield

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

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

The function is

```
<(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;1;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;<p1;p2;p3;p4;p5>;
<##<ds;KW;<;p1;p2;p3;p4;p5>>
#<FF.k;#<KW1;X;17>;#<KW1;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;df&t>;
<#<eq;##<sc;KW;<;key=>>;0;<df&t>;
<#<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 removed ;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:

```
#<defkw;name;subs;keys;text>
```

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>>
#<dkw1;#<dkw2>>#<rrp;key1>#<rrp;key2>#<rrp;key3>
#<dkw3;<name>;<subs>;##<key1>;##<key2>>
#<dkw4;<name>;<subs>;##<key3>;<text>>
#<es;key;key1;key2;key3;key4>>>

#<def;dkw1;<keyw;dflt>;<#<eq?;keyw;;;<##<ap;key1;<keyw*;>>
##<ap;key2;<;#<KW1;keyw;dflt>>>##<ap;key3;<keyw;>>
#<dkw1;#<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;##<$$name>>
#<Mod-1;$$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-1;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;Mod-2;<NAME;answer>;
<#<eq?;answer;yes;<#<Mod-1;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;##<prl##<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>|<sum>-<term>

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

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

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

<digit>::=0|1|....|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 or 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 BTSS. 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.

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 the idle program. 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 synonomous.

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

Other users initials cannot be used in STORE and DELETE.

Could Not Attach

An external device could not be attached.

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

abs	1	1	V
ad	2	2	V
ap	2	2	S
break	0	1	S
cc	1	1	VS
cc&	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
dc&	2	2	S
delete	1	1	S
dnc&	2	2	S
ds	2	2	S
dv	2	2	V
dvr	2	2	V
ec&	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
isc	4	4	VS
&t	4	4	V
&t?	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
rcd	0	2	S
rrp	1	1	S
rs	0	0	V
sc	2	63	VS
scc	2	2	S
sc&	2	2	S
scn	3	3	VS
show	0	1	S
ss	2	63	S
store	2	*	S
su	2	2	V
tabs	1	8	S
tc&	4	4	V
tf	0	0	S
time	0	0	V
tn	0	0	S
z&c	1	1	V
z&cp	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 McIlroy, 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. McIlroy, M.D., Macro Instruction Extensions of Compiler Languages. CACM 3, No. 4 (1960), 214-220.

8. McIlroy, 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.