```
 MMM
 MMMM       MMM
   MM      M MM
   M      M
  M      M     MMMMMMMMM
 MM     MM    MMMM    MMM
MMM     MM      MM    MMM
MMM    MMM      MM   MMM
MMMMMMMMMMM      MMMMMMM                 MM
 MMMMMMM MMMM    MMM   MM              MMMMM
          MMM    MM    MMM            M   MM
                 M     MMM           M    M
                M  MM  MMM          MM
            MMMM   MMMMMM          MMM
            MMM                    MMM
                                   MMM
                                    MMM    M
                                     MMMMM
```

****
*   *
*   USER'S MANUAL FOR "TOPPS" *
*   *
****

Edited by

Alan Ballard

Technical Manual 75-6

Revised October 1974

Department of Computer Science
University of British Columbia
Vancouver, B. C.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The TOY Language, on which TOPPS is based, was designed by J. J. Horning; the TOY compiler/interpreter was implemented by B. Clark and F. Pagan (University of Toronto, C.S.R.G., 1971). The concurrency features of TOPPS were designed by R. Holt and R. Kinread and were added to the TOY compiler/interpreter by M. Dryer.

The TOPPS compiler/interpreter has been modified, extended and stabilized by Alan Ballard and Bryon Czarnik (University of Toronto, Department of Computer Science, June 1973). It has been adapted for MTS and further modified by Alan Ballard (U.B.C., 1974).

The TOPPS user's guide was originally written by M. Dryer. It has been revised by A. Ballard and B. Czarnik to reflect changes incorporated in the current version of TOPPS. This version of the user's guide has also borrowed from a user's guide produced by R. Dalgleish and R. B. Bunt (University of Saskatchewan).

CHAPTER ONE

INTRODUCTION AND OVERVIEW

1.1. GENERAL INTRODUCTION

In the study of operating systems, the concepts of asynchronous processes and of process communication are particularly important. TOPPS is a simple language developed at the University of Toronto as a means of giving students practical experience with the problems involved. Facilities are provided for the simulation of asynchronous processes and for communication between them. The TOPPS processor consists of a compiler producing code for a pseudo-machine and of an interpreter which executes the generated code. Both are written in the XPL language.

The TOPPS language is a block-structured language with some semblance to Algol. Operations for the handling of both numeric and string data are provided, along with control structures for repetition and logical selection. There is no GO TO construct. Every executable statement in TOPPS returns a value; consequently, anywhere a value is required, a statement (or block of statements) may be used. It is certainly unnecessary to make use of this feature; however with a little practice it can provide considerable programming convenience (but often with a corresponding obscurity). Expressions are normally evaluated right to left, rather than by the more usual arithmetic precedence rules.

Probably the most important features of the language are those for implementing and communicating between processes. Processes are virtual processors executing procedures asynchronously and in parallel. Communication between processes is handled through a special data type called resources that essentially combine Dijkstra's Semaphores with message queues. Resources are manipulated through special primitives REQUEST and RELEASE.

1.2. PROGRAMS, PROCESSES AND VIRTUAL PROCESSORS

In computer systems, there are frequently many activities being performed simultaneously. The activities may depend on events external to the activity itself. Such things as completion of input-output operations, real-time clock interrupts, and other exceptional conditions are not predictable in a simple deterministic way. Since no simple timing relationship holds between such events, they are said to be _asynchronous_.

TOPPS allows a user to create _virtual processors_ which execute asynchronously and in parallel. The user can define PROGRAMs which are a special form of procedure. There is a PROCESS statement which, when executed, causes a new processor to be created (or, "fired up"). It begins execution of the PROGRAM named in the process statement. The combination of a processor executing a procedure is called a _process_. The new process continues executing independently of, and asynchronous to, the execution of the process which issued the PROCESS statement (and all other processes). Note that more than one processor may be executing the same procedure simultaneously. The code is fully reentrant.

Processes are hierarchically related, with the new process being referred to as a _child_ and the old process as the _parent_.

A process runs until the processor finishes the code for the program, executes a RETURN statement or becomes _blocked_ or _deadlocked_ by a request for an unavailable resource. The termination of a process has no effect on any other process.

The TOPPS interpreter actually achieves "logical parallelism" by interleaving execution of the processes. That is, slices of CPU time are randomly distributed across the existing virtual processors. Because of the randomness, a programmer cannot guarantee that two processes will reach particular points in their respective programs at the same time unless they are synchronized by the use of resources.

1.3. RESOURCES

Interprocess communication and synchronization is achieved through the use of _resources_ and the primitives REQUEST and RELEASE which operate upon resources. A process may REQUEST that a single _unit_ of a particular resource be allocated to it. If any units of that resource are available, the process is given one. If no units of the resource are available, the process is _blocked_ (i.e., its execution is suspended) until its request can be satisfied. Blocked processes are placed on a FIFO queue associated with the requested resource; this queue is checked each time a unit of the resource becomes available.

Units of a resource become available when some process issues a RELEASE statement for that resource. If there are blocked processes awaiting units of the resource, then the newly available unit is given to the first process on the queue and that process is allowed to continue execution. If there are no outstanding requests for a unit of the resource, the unit is placed on a FIFO queue. Each released unit has a string, numeric, or logical value (i.e., a message) associated with it which is made available to the process to which that unit is subsequently allocated.

There are two types of resources in TOPPS: REUSABLE and CONSUMABLE. The type of each resource is specified by the programmer in the declaration for that resource.

Units of a reusable resource are "borrowed" by a requesting process and subsequently "returned" using the RELEASE statement. A process can obviously only release units of resources which it already possesses. The programmer specifies a numeric unit count for each reusable resource which indicates the number of units of that resource which are initially available. This is also the number of units which will be circulating in the system at any time since units of reusable resources are neither created nor destroyed, but merely borrowed and returned.

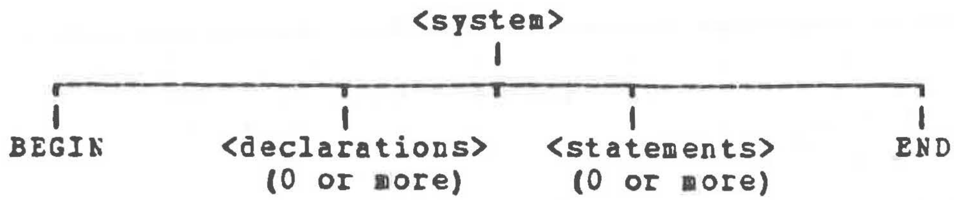
In contrast, each consumable resource has an initial unit count of zero. Processes may be given the capability to "mint" units of particular consumable resources. The RELEASE statement applied to such a resource thus has the effect of creating a new unit of that resource. When a unit is allocated to a process in response to a REQUEST, that unit is "consumed" and ceases to exist.

Consumable resources may be used to pass messages by having the "sending" process release a unit of a resource whose value is the message. The "receiving" process (which obviously must know that it is to receive a message) must request a unit of that resource.

Process synchronization and mutual exclusion are handled through the use of consumable or reusable resources. Mutual exclusion is best accomplished through the use of reusable resources with unit counts of one.

1.4. FLOW OF CONTROL

A program written in TOPPS is considered to be a PROGRAM and is executed by the main processor. Syntactically it is a <system> with the following structure:



Variables, arrays, resources, programs, and subprograms are declared at the beginning of the TOPPS program. Then follow statements to be executed. There are two kinds, expressions and PROCESS statements to fire up processes. Expressions may be of the following types:

(1) subprogram calls
(2) iterative expressions (i.e. REPEAT's)
(3) selection expressions (i.e. IF's)
(4) assignments
(5) return expressions
(6) declaration blocks.

Execution of processes is sequential except for branches caused by REPEAT's, IF's, RETURN's and subprogram calls. Termination of execution occurs whenever all PROCESSes finish (i.e. 'fall off' the ends of their respective PROGRAMS) or when the PROCESSes not finished are all deadlocked or blocked by requests for unavailable resources.

CHAPTER TWO

SYNTACTIC DEFINITION OF TOPPS

2.1. THE COMPLETE SYNTAX

In this section, a simplified syntax for TOPPS is listed. The actual syntax used by the TOPPS compiler is listed in Appendix II. In the following, square brackets [and] denote optional items.

```
<system> ::= <decl.block>

<decl.block> ::= BEGIN [<decl.state.list>] <block> END

<decl.state.list> ::= <decl.state.list> <decl.state.>
                    | <decl.state.>

<decl.state.> ::= VARIABLE <id list> ;
                | ARRAY <id list> BOUND <parameter list> ;
                | SUBPROGRAM <identifier> [ OF <id list> ]
                                             IS <expression> ;
                | CONSUMABLE <id list> ;
                | REUSABLE <id list> WITH <expression> ;
                | PROGRAM <identifier> [ OF <idlist> ]
                       [ PRODUCING <id list> ] IS <expression> ;

<block> ::= <statement>
          | <block> ; <statement>

<statement> ::= <expression>
              | PROCESS <primary> [ OF <parameter list> ]
                                  [ PRODUCING <parameter list> ]
              | <empty>

<expression> ::= <primary>
               | ¬ <primary>
               | <primary> <operator> <expression>
               | IF <expression> THEN <expression>
                                             ELSE <expression>
               | REPEAT <block> UNTIL <expression>
               | RETURN <expression>

<primary> ::= <identifier>
            | <constant>
            | ( <block> )
            | <decl.block>
```

```
            | <primary> ( [<parameter list>] )

<parameter list> ::= <expression>
                   | <parameter list> , <expression>

<id list> ::= <identifier>
            | <id list> , <identifier>

<operator> ::= +|/|*|<|=|:=|>|>=|<=|¬=|&| |
```

2.2. BLOCKS AND STATEMENTS

```
<block> ::= <statement>
          | <block> ; <statement>

<statement> ::= <expression>
              | PROCESS <primary> [OF <parameter list>]
                                  [PRODUCING <parameter list>]
              | <empty>

<decl.block> ::= BEGIN <decl.state.list> <block> END

<decl.state.list> ::= <decl.state.list> <decl.state.>
                    | <empty>
```

2.2.1. Statements And Declaration Statements

There are two types of statements in TOPPS, distinguished in the grammar by <statement> and <decl.state.>.

A <statement> is either an <expression> (in which case the value of the <statement> is the value of the <expression>), or a statement firing up a process, beginning with the word PROCESS (in which case the value of the <statement> is zero), or a null statement (with a value of zero). When a semicolon follows a <statement>, the value of the statement is discarded.

<u>Examples</u>:

The following are examples of <statement>'s.

```
(1)  A := B
(2)  OUTPUT('X IS', X)
(3)  RELEASE(Message, 'READY')
(4)  RESULT := CATENATE(A, B)
(5)  PROCESS Reader OF 2, I PRODUCING Message
(6)  PROCESS WRITE OF LINE
```

```
(7)  MAX := IF A < B
             THEN B
             ELSE A
(8)  RETURN 'ABNORMAL TERMINATION'
(9)  IF Switch
        THEN BEGIN
                VARIABLE Temp;
                Temp := Var2;
                Var2 := Var1;
                Var1 := Temp
             END
        ELSE 0
```

A <decl.state.> is a statement which declares a variable, array or resource, or defines a program or subprogram. Unlike <statement>'s, <decl.state.>'s do not possess values. All declared names are local to the block in which they are declared. All names must be declared before they are used. For further details on declarations and examples see Section 2.3.4.

2.2.2. Blocks And Declaration Blocks

There are two types of blocks in TOPPS, distinguished in the grammar as <block> and <decl.block>, which have two basic differences. A <block> consists simply of one or more <statement>'s separated by semi-colons. Note that declarations are not <statement>'s. A <decl.block> on the other hand must begin with the word BEGIN and end with the word END, and may include declarations. A <decl.block> does not have to contain declarations; however, they must occur at the beginning of the <decl.block> if they are present. The rest of the <decl.block> is the same as a <block>. A <decl.block> is a <primary> and hence may be used anywhere a value is required. A <block> may always be parenthesized to use it where a <primary> is wanted. Note that a <decl.block> always causes storage allocation at execution time (even if there are no declarations inside); hence it should not be used without declarations since a parenthesized <block> would be more efficient.

Note that a <block> should not end with a semi-colon. A <block> must end with a <statement>. The value of this <statement> becomes the value of the <block>. Since a <decl.block> contains a <block>, the same applies to a <decl.block>. The value of a <decl.block> is the value of its <block>. Thus all blocks, of both types, possess a value. If an extra semi-colon is present at the end, a null statement is assumed with a value of zero. A warning is printed at the end of compilation if this is detected.

A new scope (i.e., lexic level) is entered each time a <decl.block> is entered or a <subprogram name> or <program name> is encountered. Variables declared inside a scope may have the same name as variables declared outside. However, the same name may not be declared twice within the same scope. The implicitly defined subprograms have lexic level zero, while names declared in the outermost declaration block have lexic level one. Order numbers within given scope are assigned in the order declared with the first name having order number zero.

Examples:

The following are examples of <block>'s:

(1)
```
X
```

(2)
```
(X + 3)
```

(3)
```
X := Y + 3;
2*Y
```

(4)
```
INPUT(N);
N + (INPUT(A);
     OUTPUT(A))
```

The value of this block is N + A.

The following is an example of a <decl.block>:

(5)
```
BEGIN
   VARIABLE I;
   I := 10;
   OUTPUT(I)
END
```

2.3. TYPES AND DECLARATIONS

2.3.1. Data Types

The following data types exist in TOPPS:

(1) Constants may be of string or of numeric attribute.

(2) Variables may have either string or numeric values.

(3) Arrays are sequences of values, either string or numeric or a combination of both.

(4) Programs, as described, are special procedures used in simulating parallel processes.

(5) Subprograms are procedures, which always return a value.

2.3.1. DATA TYPES

(6) Resources, as described, synchronize processes and queue information.

Declarations are used to enter the name of the data item in the symbol table at compile time and allocate space on the run stack at execution time. The values of variables and arrays are initially undefined at execution time.

2.3.2. Constants

<constant> ::= <integer> | - <integer> | <string>

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

<decimal digit> ::= 0|1|2|3|4|5|6|7|8|9

<string> ::= '<characters>' | ''

<characters> ::= <character> | <characters> <character>

<character> ::= '' | {any EBCDIC character other than '}

Integers in the range 0 to $2^{31}-1$ are valid. Negative integers may also appear in TOPPS programs. However, it is not possible to read negative values from data files at execution time.

A string constant is a string of zero or more characters not including the apostrophe (') enclosed by apostrophes. Two apostrophes must be used to represent the occurrence of one apostrophe within a string while two apostrophes alone represent the null string. The maximum length allowed for a string constant is 255. Strings contained in input data may be of the same form and enclosed in apostrophes. Alternatively, if the READ function is used, then an entire input line is accepted as a character string.

2.3.3. Identifiers

<identifier> ::= <letter> | <identifier> <letter>
| <identifier> <decimal digit>

<letter> ::= A|B|C|...|Z|_|@|#|$|a|b|c|...|x|y|z

<decimal digit> ::= 0|1|2|3|4|5|6|7|8|9

An identifier is a string consisting of a letter followed by zero or more letters or digits, where _, @, $, # are considered to be letters. The following are reserved words in

TOPPS and may not be used as identifiers:

> OF, IS, IF, END, THEN, ELSE, WITH, ARRAY, BOUND, BEGIN, UNTIL, RETURN, REPEAT, PROCESS, PROGRAM, VARIABLE, REUSABLE, PRODUCING, SUBPROGRAM, CONSUMABLE

Implicitly declared names, described in chapter three, are treated as identifiers declared in a enclosing block and may be freely redeclared in TOPPS (if the corresponding functions are not required).

2.3.4. Declarations

All declarations of data items must occur before the item is referenced and at the beginning of a declaration block. Standard Algol scope rules are used for declared items. Therefore, data items declared in a block are not "visible" outside the scope of that block although interior to the block they may be referenced or redeclared (which causes a new data item to be entered into the symbol table).

2.3.4.1. Variables

<decl.state.> ::= VARIABLE <id list>;

An identifier is a variable if it occurs in the <id list> of a declaration statement of the form VARIABLE <id list>. Any variable may have numeric, string, or undefined values at any time. Thus in the block

```
BEGIN
   VARIABLE I;
   I := 1;
   I := 'OUTPUT IS';
   I := 3
END
```

the variable I is first undefined, then number-valued, then string-valued, and later number-valued again.

2.3.4.2. Arrays

<decl.state.> ::= ARRAY <id list> BOUND <parameter list>;

An identifier which appears in an ARRAY statement is array-valued and has the dimensions specified by the expressions in the <parameter list> after BOUND. The expressions are evaluated at the time execution of the <decl.block> begins. If an array has bounds B1,...,Bn, then the i'th subscript can take on values between 0 and Bi inclusive, so that the total number of elements in the array is (B1+1)x(B2+1)x...x(Bn+1).

Any use of an array identifier after its declaration is interpreted as a special kind of subprogram call which returns a reference to an element of the array. Array elements are used in the same way as variables.

Examples:

(1) ARRAY Code BOUND Code_Size;

This declares an array Code of size Code_Size + 1.

(2) ARRAY A,B,C BOUND I,J;

This declares three 2-dimensional arrays of size (I+1)x(J+1).

2.3.4.3. Resources

<decl.state.> ::= CONSUMABLE <id list>;
| REUSABLE <id list> WITH <expression>;

An identifier declared in a CONSUMABLE or a REUSABLE statement is a resource, and is respectively consumable or reusable. A resource identifier may only be used as a parameter for a subprogram or program or a resource parameter (i.e., after PRODUCING) for a program. Manipulation of resources is normally done by the use of the implicitly defined subprograms REQUEST and RELEASE.

The expression after WITH in the declaration of a reusable resource specifies the number of units of each resource in that REUSABLE statement. Initially all the units of a reusable resource are available to be requested. Each time a unit of a reusable resource is requested by a process, that process becomes the owner of one more unit of the resource and there is one less unit available. Each time a unit of a reusable resource is released, then that process owns one less unit of the resource and one more unit is available to be used again. If all the units of a reusable resource have been assigned to processes and some process requests a unit, then that process is placed in a queue of processes awaiting units of that resource and remains blocked until some other process releases a unit of the resource.

Initially there are zero units available of a consumable resource. There is no fixed number of units of a consumable resource, since units of a consumable resource are created when a process releases them, and they are destroyed as soon as they are obtained by some process. When a consumable unit is released it is placed in a queue of available units of that resource. When a process requests a unit of this resource it removes the unit at the front of this queue. However, if the queue is empty the process is placed in a queue of processes awaiting units of this resource.

With both types of resources, when a unit is released, the FIFO queue of processes waiting for that resource is checked; if the queue is not empty then the unit is given to the process at the head of this queue. That process is removed from the queue, and allowed to resume execution.

To release a unit of a reusable resource, the process must own a unit. To release a unit of a consumable resource, the process must be a legitimate producer of that resource. A process can produce a consumable resource only if it is declared within that process, or was included in the resource parameter list (i.e., the list following PRODUCING) when the process was started up.

Examples:

```
CONSUMABLE Messages;
REUSABLE Mutex WITH 1;
```

2.3.4.4. Subprograms

<decl.state.> ::= SUBPROGRAM <identifier> [OF <id list>] IS <expression>;

The definition of a subprogram is headed by the reserved word SUBPROGRAM followed by its name and its formal parameters (if any). The body of a subprogram is the expression following the reserved word IS. Every subprogram call returns the value of the <expression> forming the subprogram body; however, this value need not be used by the calling program.

The parameters of the subprogram are local to the <expression> constituting the subprogram body. They are implicitly defined by their presence in the parameter list, and must not be redeclared in the subprogram body. When a subprogram is called, all parameters are passed by reference except for those which are constants or which are expressions resulting in values which are not references. The number of arguments in the call (actual parameters) must match the number of formal parameters. Subprograms may be called recursively. An example of a subprogram is given in Appendix VII.

2.3.4.5. Programs

<decl.state.> ::= PROGRAM <identifier> [OF <id list>]
[PRODUCING <id list>] IS <expression>

Execution of a statement of the form

PROCESS <primary> [OF <parameter list>]
[PRODUCING <parameter list>]

creates a new processor executing the PROGRAM with the name specified by the <primary>. The new processor executes quite independently of the originating process (and any other processes). A processor continues until it runs off the end of its program, or executes a RETURN that is not inside a subprogram. Any number of processors can be executing the same program at the same time. The main program itself is treated as a PROGRAM being executed by a processor.

Processes communicate and interrelate by means of resources. Consumable resources are used for passing messages back and forth. Reusable resources with one unit can be used to give certain processes exclusive use of some critical section of a program. For example, suppose READER is a program which five processors are executing. If it is desired that some part of the program READER be executed by only one of the five processors at a time, then a reusable resource, declared

REUSABLE Mutex WITH 1;

can be requested on entry to the critical section of the program. Since there is only one unit of the resource, only one process can be in the critical section at a particular time.

The <id list> after the program name is the list of formal parameters, which are analogous to the formal parameters for a SUBPROGRAM, so what was said there applies here too. They correspond to the actual parameters specified in the parameter list of the PROCESS statement firing up the process. There is one very important difference, however. With subprograms, parameters are passed by reference whenever possible and by value only if necessary. Although program parameters are still passed by reference if the parameter is an array, a resource, a program, or a subprogram, they are passed by value if the parameter is number-valued or string-valued. Explicitly, the difference is this: with subprograms, if it is possible to pass a reference to a variable or an array element, the reference is passed. In analogous situations with programs, the value of the variable or array element is passed instead. This difference was considered desirable because if Process 1 fires up Process 2 with a parameter list including a variable, Process 1 might change the value of the variable before Process 2 could use it.

Programs also have a resource parameter list naming the consumable resources of which that program may release (i.e., produce) units. These formal parameters correspond to the actual parameters in the PRODUCING part of the PROCESS statement firing up the process. The actual parameters must be consumable resources (or expressions resulting in references to consumable resources). A process which is not declared a producer of a consumable resource may not release units of that resource. Whenever one process fires up another process the former must be a producer of the consumable resources included in the resource parameter list. As with the normal parameters, the formal resource parameters should <u>not</u> be redeclared within the body of the program expression, since their occurrence in the formal parameter list constitutes the declaration.

The PROCESS statement firing up a process must provide the number of parameters specified in the declaration of the program for both parameter lists. Unlike subprograms, programs do not return a value to the firing-up point. The result of firing up a process is that the <expression> is evaluated by the new processor. The value of the PROCESS statement in the parent process is always zero.

<u>Example</u>:

```
PROGRAM Inputter OF X PRODUCING Message IS
   REPEAT
      RELEASE(Message, X)
   UNTIL ¬ INPUT(X);
```

This program will release the consumable 'Message' with the value X. Then new data will be read into X and released. This process will continue until there is no further data to read.

Note: Further examples of programs and processes are presented in Appendix VII of the manual.

2.4. EXPRESSIONS

```
<expression> ::= <primary>
               | ¬ <primary>
               | <primary> <operator> <expression>
               | IF <expression> THEN <expression>
                                  ELSE <expression>
               | REPEAT <block> UNTIL <expression>
               | RETURN <expression>

<primary> ::= <identifier>
            | <constant>
            | ( <block> )
            | <decl.block>
            | <primary> ( [<parameter list>] )
```

2.4. EXPRESSIONS

```
<parameter list> ::= <expression>
                   | <parameter list> , <expression>

<operator> ::= +|-|/|*|=|:=|<|>|<=|>=|¬=|&| |
```

2.4.1. Operators

All operators in TOPPS have equal precedence and expression evaluation is from right to left, except where modified by parentheses. There are three classes of operators.

The first class contains the logical and arithmetic operators: +, -, *, /, &, |, and ¬. The operands for these operators must have numeric values. An attempt to use a string-valued variable as an operand causes an execution-time error message and terminates execution. Any overflow from these operations is ignored; the value after overflow is the same as in XPL. The logical operators (&, |) and the unary not (¬) treat their operands as bit strings and perform the operations on corresponding bits. For an expression occurring in the phrase IF <expression> THEN ..., as in REPEAT <block> UNTIL <expression>, only the least significant bit is used. Note there are no unary plus or minus operations. Therefore - <expression> must be represented by 0 - <expression>. (Negative <u>constants</u> are possible, however.)

The second class of operators is the relational operators (=, ¬=, <, <=, >, >=) for which the operands must be both numeric valued or both string-valued. String comparison is done as in XPL.

String1 < String2 means either

(i) LENGTH(String1) < LENGTH(String2)
or (ii) LENGTH(String1) = LENGTH(String2)
but there exists i such that
BYTE(String1,i) < BYTE(String2,i) where
BYTE(String1,j) = BYTE(String2,j) for j=0,...,i-1.

Hence String1 = String2 if and only if the two strings are identical (same length and each corresponding character the same).

The third class of operators contains the single operator :=, the assignment operator. The value on the right hand is stored in the location specified on the left hand side, destroying the old value. The operands may be either string-valued or numeric-valued and do not have to have the same type of value. The value of an expression of the form A := B is the value of B. Hence the expression (A := (B := 3) + 2) has value 5, and A := 6 - 1 + 3 assigns to A the value 2.

2.4.2. Conditionals

<expression> ::= IF <expression>^{1} THEN <expression>^{2}
ELSE <expression>^{3}

First, <expression>^{1} is evaluated. If the least significant bit is 1, then <expression>^{2} is evaluated, and its value becomes the value of the expression. If the least significant bit is not 1, then <expression>^{3} is evaluated and its value becomes the value of the expression. In general, the evaluation will result in a reference if possible; hence IF expressions may occur in contexts requiring references to variables, subprograms, resources, etc.

Examples:

```
(1)  IF A
        THEN OUTPUT(10)
        ELSE OUTPUT(20)
```

In this case, if A is an odd number (i.e., least significant bit is one) then '10' is printed; otherwise, '20' is printed.

```
(2)  (IF A
        THEN B
        ELSE C) := D
```

In this case, if A is odd then the value of the IF expression is a reference to B, otherwise it is a reference to C. Then the value of D is assigned to the variable returned.

2.4.3. Loops

<expression> ::= REPEAT <block> UNTIL <expression>

Loop expressions are realized by the REPEAT construct. The <block> is always executed at least once, and is reexecuted until the <expression> following UNTIL is true (i.e., has least significant bit with value 1). The value of the loop expression is the value resulting from the last execution of the block.

2.4.4. Subprogram Calls

<expression> ::= <primary> | ¬ <primary>
<primary> ::= <primary> ([<parameter list>])

A subprogram call causes execution to branch to the subprogram code while still remaining within the same process. The expression attached to the subprogram definition is evaluated; the result is returned, and execution continues in the calling procedure. Subprograms may be called recursively.

Parameters are passed by reference if possible (i.e., if the actual parameter is not a constant or an expression containing operators). The call must provide the number of parameters specified in the declaration for the subprogram. Each parameter is an expression which may also contain subprogram calls. Subprogram parameters may be references not only to variables, but also to arrays, subprograms, programs or resources.

Note that in calling a subprogram with 0 arguments, the brackets must still be retained (i.e., <primary>()).

Array references are treated as special cases of subprogram calls in which the parameters are interpreted as subscripts.

The resultant value of a subprogram is a reference where possible. (It is not possible if the final expression in the subprogram is a constant or an expression involving operators or a locally declared identifier.) It may be a reference to any type of identifier.

<u>Examples</u>:

Subprogram calls such as the following may be used:

(1) F(I):= EXP — If F(I) returns as its value a reference to a variable or array element, then the value of the expression, EXP, will be assigned to that variable or array element.

(2) F(I)(A,B,C) — If F(I) returns as its value a reference to a subprogram, then this expression will cause that subprogram to be called with parameters A, B, and C.

(3) PROCESS F(I) OF X,Y PRODUCING C1 — If F(I) returns as its value a reference to a program, then this statement will fire up a process using that program.

(4) REQUEST(F()) — If F() returns as its value a reference to a resource, then this expression causes the running process to request a unit of that resource.

2.4.5. Returns

<expression> ::= RETURN <expression>

By means of RETURN <expression> a return may be made from arbitrary points in a subprogram or in a program. This provides an easy way of branching out of deeply nested constructs (e.g., nested blocks). The value returned is that of the <expression> after the RETURN.

CHAPTER THREE

IMPLICITLY DEFINED SUBPROGRAMS

Unless an explicit declaration is used to redefine them, several identifiers have special meaning in TOPPS: INPUT, READ, OUTPUT, LENGTH, BYTE, SUBSTRING, CATENATE, REQUEST, RELEASE, NUMERIC, DELAY, TOGGLE, QUANTUM. The effect is as though they were declared in an outermost scope containing the entire <system>. The parameters may be any type of expression as long as the value of the expression is a value or a reference which abides by the rules specified below.

3.1. INPUT/OUTPUT

3.1.1. INPUT(E1,E2,...)

INPUT provides a form of "stream-oriented" input. It may have any number of parameters. These must be variables, array references or arrays, or expressions resulting in references to such. Unless an array is used as an input parameter, successive values in the input stream are assigned to successive parameters. If an array is used as an input parameter then values are read in from the input stream until a value is assigned to each array element. Array elements are assigned with the rightmost subscripts varying most rapidly.

The value returned by an expression of the form INPUT(E1,E2,...,En) is 1 if there was input data for all the E1,E2,...,En, and zero if there was no input data for En (or insufficient data if En is an array). Only one attempt is made to read past the end of data. Any further attempts result in termination of execution.

When an attempt is made to input data into a variable or array element, INPUT starts scanning the input lines, skipping blanks, from the particular column where it stopped scanning for the previous input value, and proceeds scanning until it finds a valid <integer> or <string> or until it encounters a character other than 0,1,2,3,4,5,6,7,8,9, blank or '. In this last case a warning is printed and execution continues. Note again that negative integers cannot be input.

3.1.2. READ(E1,...)

This function is similar to INPUT, except that it provides a "record oriented" input. For each item in the parameter list (or each element of an array), an input line is read and the entire line is assigned as a character string to the variable.

The value returned is as for INPUT, i.e., it is 1 if there was sufficient data for all parameters and 0 if there is no data for the last element.

3.1.3. OUTPUT(E1,E2,...)

OUTPUT is similar to INPUT in that it may have any number of parameters which must be number or string valued. Array output is analogous to array input.

The value of OUTPUT(E1,E2,...,En) as an expression is the value of En and may be either string or number valued. If En is an array then the value of OUTPUT(E1,E2,...,En) is the last value output. An attempt to output an undefined value causes printing of a question mark. The maximum possible length of an output line is 131 characters.

Each time a call to OUTPUT is made printing starts at the beginning of a new line and the values that are printed by that particular call to OUTPUT appear on the same line as far as possible. If there is insufficient space at the end of a line to print an entire string or number, then none of the value is printed on that line, but rather the printer skips to the beginning of the next line and starts printing the value there. If the string has more than 131 characters, then the first 131 characters will be printed on the first line, and the remainder will be printed on the next line. When values are output on the same line, a blank is automatically inserted between each value. Thus, OUTPUT(ONE,'.',TWO) where ONE has value 1 and TWO has value 2 will output the line 1 . 2. If blanks are not desired, then it is necessary to first concatenate the parameters so there will only be one output value: OUTPUT(CATENATE(ONE,'.',TWO)). It is possible to print negative values. For example, OUTPUT(-5) will cause -5 to be printed.

3.2. CHARACTER STRING MANIPULATION

There are four character functions in TOPPS: LENGTH, BYTE, SUBSTRING and CATENATE. The first three are similar to the analogous functions in XPL or PL/I, and CATENATE(E1,...,En) is like PL/I E1||...||En. If a numeric value is used in a string function, the expression is converted to a string. In the following it is assumed that this conversion has occurred if necessary.

3.2.1. LENGTH(E1)

This function must have only one parameter which may be either string-valued or number-valued. Its value is a numeric value equal to the number of characters in the (converted) string denoted by E1,

Examples:

```
LENGTH('ABC') = 3
LENGTH(-2) = 2
LENGTH(2) = 1
LENGTH ('''') = 1
```

3.2.2. BYTE(E1, E2) or BYTE(E1)

E1 may be number valued or string valued. E2 must be number valued. If E2 is omitted, 0 is assumed.

The value of this function is the numeric EBCDIC representation of the E2'th character (zero origin indexing) of the string E1. An attempt to use BYTE with a negative value for E2 or with a value greater than the length of E1 generates an error message and returns the zero'th byte. BYTE may not be used on the left of an assignment.

Examples:

BYTE('123',2)	has the EBCDIC value of '3' or F3 in hex.
BYTE('123')	has the EBCDIC value of '1' or F1 in hex.
BYTE('1',-1)	causes a warning to be printed and has the value of 0.

3.2.3. SUBSTRING(E1, E2, E3) or SUBSTRING(E1, E2)

E1 may be number or string valued. E2 and E3 must be number valued. This function has as its value the substring of the string E1, starting with the E2'th character (using zero origin indexing) and continuing for E3 characters, so that the length of the substring will be E3. SUBSTRING may be used with only two parameters in which case the substring consists of the characters from the E2'th to the end of E1. An attempt to take a substring beyond the end of the string results in an error message, and returns the remainder of the string. A negative value for E2 or E3 causes zero to be used and an error message to be printed. A length of zero results in a null string value without complaint.

Examples:

```
SUBSTRING('ABCD',1,3)      yields 'BCD'
SUBSTRING('ABCD',2)        yields 'CD'
SUBSTRING('ABCD',-1,-1)    causes a warning to be printed and
                           yields the null string.
```

3.2.4. CATENATE(E1,...,En)

This function may have any number of arguments greater than two, each of which may be either string valued or number valued. The value of the function is the string resulting from the concatenation of strings E1,...,En. If the result of the concatenation is a string with length greater than 255, then a warning is printed printed and the rightmost characters are deleted.

Example:

```
CATENATE('AB','','''','C') yields 'AB'C'
```

3.3. RESOURCE HANDLING FUNCTIONS

3.3.1. REQUEST(E1)

This function must have exactly one parameter which must be a reference to a resource.

A call to this function causes the following to occur: if a unit of the resource is available then the process performing the request obtains a unit of that resource. If there is no unit of resource E1 available, then the process is placed in a queue awaiting units of that resource and remains blocked until it obtains a unit.

The value of REQUEST(E1) as an expression is the value of the unit of E1 obtained. The value is numeric zero unless that unit has been released with some other value being placed in it. The units of a reusable resource are all initially available. Units of a consumable resource are not available until they have been released by some process. If a process obtains a unit of a reusable resource, then that process owns that unit until it releases it. If a process still owns some units of a reusable resource when it finishes (in other words, if it has failed to release a unit of some reusable resource it requested), then execution will terminate with an error message. If a process obtains a unit of a consumable resource then the unit is 'consumed'. In other words, the unit disappears, except that its value is transferred as the value of the expression REQUEST(E1).

3.3.2. RELEASE(E1, E2) or RELEASE(E1)

E1 must be a reference to a resource. E2 may be string valued or number valued. If E2 is omitted, a value of 0 is assumed.

A call to this function causes a unit of resource E1 to be released with value E2. The result of the expression is the value of E2. The value of RELEASE(E1) is zero.

If E1 is a reusable resource, then the releasing process must own a unit of that resource (i.e., the process must have requested and received a unit of the reusable resource in the past). Releasing a unit of a reusable resource returns that unit to the appropriate queue of available units of that resource with a value equal to that of E2. The process releasing the unit of the resource no longer owns that unit. Execution is terminated if a process attempts to release a unit of a reusable resource without owning one.

If E1 is a consumable resource, then the releasing process must be a producer of that resource (i.e., the resource must be contained within the resource parameter list for that process or be declared within the PROGRAM which was invoked as a process). Releasing a unit of E1 in effect creates a unit carrying the value of E2 and places that unit in a queue of available units of E1 unless a process is awaiting a unit of E1. This unit will be destroyed when some process obtains it.

For either type of resource, there is nothing to prevent a process from obtaining a unit of a resource which it previously released.

3.4. OTHER FUNCTIONS

3.4.1. NUMERIC(E1)

E1 may be number-valued or string-valued. This function may only have one argument. If the expression, E1, is number valued, then the function returns 1. If it is string valued, then it returns a value of 0.

Examples:

```
NUMERIC('NUMBER')     returns 0
NUMERIC(999)          returns 1
```

3.4.2. DELAY(E1)

E1 must be number valued. This function is provided as a simulation tool, for use in simulating processes that run at "different speeds". It should be used only for that purpose. No attempt should be made to "synchronize processes" by means of DELAY. That's what REQUEST and RELEASE are for.

To explain the DELAY function, it is first necessary to explain that there are two clocks internal to TOPPS which have no relationship either to each other or to real time. These clocks are called the "machine cycle clock" and the "simulation clock". The time statistics printed for each process after execution are based on the machine cycle clock which is based on one time unit per ideal machine instruction. The second clock is the simulation clock which is entirely controlled by calls to the subprogram DELAY. (Note that DELAY does not affect the machine cycle clock). Each process can be considered to have its own simulation clock.

These clocks initially have a time setting of zero. A simulation clock is changed by a call DELAY(E1) which causes the simulation clock of the process making the call to be set ahead E1 simulation clock time units. As long as there is a process whose simulation clock has an earlier setting than the delayed process, then the delayed process will not proceed. In general, only the processes with the currently smallest simulation clock time setting are executed. If all the processes with the currently smallest time setting finish or become blocked, then their simulation clocks are moved ahead to the time of the next smallest time setting, and then all unblocked processes with that time setting are executed. For example, suppose there are three processes P1, P2, and P3. Initially their simulation clocks all read zero. P1 calls DELAY(2). Its clock is reset to

2. P2 and P3 then proceed until both are blocked. Their clocks are reset to 2. P1 proceeds and suppose it releases units that cause P2 and P3 to become unblocked. All three processes continue execution. Suppose P1 calls DELAY(1) and P2 calls DELAY(2). Then their simulation clocks are set to 3 and to 4 respectively. If P3 should become blocked, then P1, the process with a clock time of 3, which is currently the lowest, proceeds. And so on ....

DELAY provides a way of controlling the relative speeds of execution of processes. However, to be effective, delays should be used in all processes. The effect would roughly be to slow down processes in proportion to their relative increments. An example of the use of DELAY is given in Appendix VII.

The simulation clock mechanism must be activated with the 'A' toggle (see Appendix IV).

3.4.3. TOGGLE(E1)

The argument E1 may be either string-valued or number-valued. The TOPPS interpreter has a number of control "toggles" which can be turned on or off by the TOGGLE function. These toggles affect the printing of debugging output, time slicing algorithm, and the delay function. A description of the toggles currently implemented is contained in Appendix IV. If the value of E1 is numeric, the specified toggle is inverted. If it is character, the first byte of the string is used (i.e. BYTE(E1)) instead. Note each call to TOGGLE inverts the setting.

The value returned by the expression TOGGLE(E1) is the new setting of the specified control (either 0 or 1).

3.4.4. QUANTUM(E1)

E1 must have a numeric value. This function provides another way of simulating processes that run at different speeds. In this case, the specified argument is used to directly specify the time-slice quantum, i.e., the number of instructions that the process calling the function is allowed to execute before the CPU is relinquished to another process.

Appendix IV describes the alternative time-slicing techniques available. Note again, that this is a simulation tool, not a method of process synchronization.

APPENDIX I

USING TOPPS UNDER MTS

The TOPPS compiler and interpreter are currently in the files Y410:TOPPSCOM and Y410:TOPPSINT respectively.

To compile a TOPPS program:

```
$RUN Y410:TOPPSCOM [i/o units] [PAR=SIZE=xxx]
```

where i/o units may be

```
SCARDS = source
SPRINT = listing
2      = "auxiliary source" (see Appendix III)
7      = machine code for the interpreter;
         defaults to -LOAD
```

and SIZE=xxx, if specified, controls the free-string area allowed the compiler. The default is 5P.

To interpret the object program:

```
$RUN Y410:TOPPSINT [i/o units] [PAR=SIZE=xxx]
```

where:

```
SCARDS = data read by the INPUT and READ functions
SPRINT = output from OUTPUT function
7      = "object program" produced by compiler;
         defaults to -LOAD
```

APPENDIX II

THE TOPPS GRAMMAR

This appendix contains the actual LALR(1) BNF grammar used by the TOPPS compiler. Parse stack dumps appearing with compiler syntax error messages use this grammar.

```
1     <system> ::= <decl block>

2     <block> ::= <statement>
3               | <block> <semicolon> <statement>

4     <semicolon> ::= ;

5     <statement> ::= <expression>
6                   | PROCESS <primary> <pars> <prod part>
7                   | <empty>

8     <prod part> ::= <empty>
9                   | OF <parameter list>

10    <expression> ::= <primary>
11                   | ¬ <primary>
12                   | <primary> <operator> <expression>
13                   | <if clause> <true part> <false part>
14                   | <repeat> <block> UNTIL <expression>
15                   | RETURN <expression>

16    <repeat> ::= REPEAT

17    <if clause> ::= IF <expression>

18    <true part> ::= THEN <expression>

19    <false part> ::= ELSE <expression>

20    <primary> ::= <identifier>
21                | <constant>
22                | ( <block> )
23                | <decl block>
24                | <primary> ( <parlist> )

25    <parlist> ::= <empty>
26                | <parameter list>

27    <constant> ::= <integer>
28                 | - <integer>
29                 | <string>

30    <parameter list> ::= <expression>
```

```
31                         | <parameter list> , <expression>

32   <decl block> ::= <begin> <decl st list> <block> END

33   <begin> ::= BEGIN

34   <decl st list> ::= <empty>
35                    | <decl st list> <decl st>

36   <decl st> ::= VARIABLE <id list> ;
37               | ARRAY <id list> BOUND <parameter list> ;
38               | <subprogram name> <arg list>
                                                IS <expression> ;
39               | CONSUMABLE <id list> ;
40               | REUSABLE <id list> WITH <expression> ;
41               | <program name> <arg list> <resource list>
                                                   <expression> ;

42   <id list> ::= <identifier>
43               | <id list> , <identifier>

44   <subprogram name> ::= SUBPROGRAM <identifier>

45   <arg list> ::= OF <id list>
46                | <empty>

47   <program name> ::= PROGRAM <identifier>

48   <resource list> ::= IS
49                     | PRODUCING <id list> IS

50   <operator> ::= +
51                | -
52                | *
53                | /
54                | <
55                | =
56                | :=
57                | >
58                | <=
59                | >=
60                | ¬=
61                | &
62                | |
```

APPENDIX III

COMPILER CONTROL STATEMENTS

The compiler recognizes a number of control statements which affect listing and other output information.

Control statements must occupy a separate input line, and must begin with a $-sign in column 1.

Control statements currently recognized are:

$LIST	Controls listing of source file. Each occurrence of the command inverts the status. Initially, listing is on.
$FILE	Input is normally read from the file attached to the unit SCARDS. However, the file attached to unit 2 may be used as an auxiliary input source. The $FILE control statement is used to flip between the two files. (This is convenient to include predefined program segments in with the source). A $FILE control statement in the auxiliary input file will cause input to be resumed from SCARDS; alternatively, this happens automatically if an END-OF-FILE occurs on unit 2.
$AUXLIST	Controls listing of auxiliary input file.
$SYMBOL	Sets a toggle causing the symbol-table to be dumped at the end of compilation.
$CODE	Sets a toggle causing the "object code" to be listed after compilation. Note there is no facility for selectively listing object-code; either all or none is listed.
$TITLE	Sets the title to be printed at the top of each listing page. The remainder of the input line is used as the new title.
$PAGE	Causes an immediate page skip.
$SPACE	Causes three blank lines to be printed.

APPENDIX IV

INTERPRETER OPTIONS

The interpreter provides a number of options, most of which are either for controlling processor scheduling, or producing debugging output.

The builtin TOGGLE function (see section 3.4.3) is used to turn on or off a number of switches controlling the options. For historical reasons, the switches used are referred to by one-character "names".

IV.1. PROCESS SCHEDULING OPTIONS

A number of switches, and the builtin DELAY and QUANTUM functions are used to affect the process scheduling performed by the interpreter.

The initial, default method used is to schedule processes in round-robin fashion, giving each a pseudo-random time-slice. (The time-slice is actually the number of pseudo-machine operations executed before scheduling the next process.)

The following toggles change the choice of time-slice:

'I' (201) This toggle sets the time-slicing method to instruction by instruction slicing for simulation of "completely interleaved" processes. This results in a large amount of process scheduling for the interpreter and should in general be avoided.

'J' (209) This toggle sets the time-slice to maximum, so that each process runs until it either blocks on a resource queue, or finishes execution. This results in the least amount of scheduling for the interpreter and is hence the cheapest way. On the other hand, it results in the "least parallel" effect. Use of this toggle is recommended during early stages of program development.

If both the 'I' toggle and 'J' toggle are specified, then the effect of 'J' is used.

An alternative way of controlling the time slice is with the QUANTUM builtin function. This allows the user to specify directly within a process what time-slice it is to receive (starting with the next time the process is scheduled). Using the QUANTUM function it is possible to simulate processes which have different speeds.

If the quantum has been set for a process by means of the QUANTUM function then the specified time-slice is used for that process, regardless of the settings of the 'I' and 'J' toggles. Processes for which a quantum has not been set will continue to be scheduled according to the settings of the 'I' and 'J' toggles. By calling QUANTUM with an argument of 0 it is possible for a process to revert to the default scheduling.

Note that use of the 'I' toggle is equivalent to each process calling QUANTUM with an argument of 1, and use of the 'J' toggle is equivalent ot each process setting its quantum to 32767.

The scheduling discipline can be modified from the usual round-robin technique by the use of the DELAY function. This is explained in some detail in section 3.4.1. Before using this function, it is necessary to activate the facility in the interpreter by means of the 'A' toggle:

'A' (193) Activates the delay options. That is, TOGGLE('A') is used to turn on the simulation clocking. If this is turned on and off in different places, the results are unpredictable.

IV.2. TRACING OPTIONS

'S' (226) Turning this switch either on or off causes an immediate dump of the segments of the Run Stack accessed by the executing process. The format of the dump is described in Appendix V.

The toggles described below provide a tracing facility. 'T' and 'U' turn tracing on and off; the remaining ones select what trace information is to be printed. Note that 'T' must be turned on for <u>any</u> of the others to have effect. All toggles are initially false.

'T' (227) Activates tracing.

'U' (228) Turns off tracing (i.e., this cancels out effect of 'T').

'D' (196) Dumps all segments of the stack that are accessible to the executing process, after execution of <u>each</u> instruction. The format of the resulting dump is described in Appendix V.

'X' (231) Prior to each instruction, prints one line specifying the instruction, address, and process.

'Y' (232) Prints out process statistics after each instruction.

'R' (217) Traces returns from subroutines by printing a dump at each return.

'1' (292) Prints a line specifying information about resources released and requested. This is mainly intended for use in debugging the TOPPS interpreter.

'2' (243) Prints a trace of string usage. This is mainly intended for use in debugging the TOPPS interpreter.

'3' (244) Prints trace of process scheduling. This is mainly intended for use in debugging the TOPPS interpreter.

IV.3. OTHER TOGGLES

'W' (230) If turned on, the interpreter will continue execution after a warning (non-fatal error). At most ten will be permitted. This is initiallly on.

'P' (215) If on at the end of execution, the interpreter will print process statistics after execution. It is initially on.

APPENDIX V

INTERPRETATION OF THE STACK DUMP

The TOPPS dump of the run stack is printed either when a job abends during execution with error code greater than 0 or when the 'R' or 'D' toggles are on (remember 'T' must be on also) or when the 'S' switch is set. The dump presents those segments of the cactus stack accessible to the currently active process.

Stack segments are formed whenever a declaration block, a program, or a subprogram is entered. The first segment is for the currently active program block. The bottom three locations of the segment form a base containing relevant information for that segment (see BASE below). Above the base are the descriptors for each element declared within that block, program or subprogram. By examining the lexic level and order number of a variable from the symbol table dump, one can then find the location of that variable in the stack dump and its corresponding value. Locations above the descriptors are for expression evaluation.

The run stack (RS) is four bytes wide. Beside each element is printed the attribute from the attribute stack (ATS). The meaning of the RS entries vary according to the ATS entries as follows:

0 REFER — The RS entry contains a reference (address) pointing to another location in the RS.

1 UNDEF — The variable declared is undefined. The RS entry is meaningless.

2 ARRAY — The RS entry contains an array descriptor:

N	BCUNDS	ADDR

where:

N — Number of dimensions of the array (4 bits);

BOUNDS — the address in the RS of the bounds segment (14 bits);

ADDR — the address in the RS of the array storage segment (14 bits).

3 SPROG The RS entry contains a subprogram descriptor:

SS	BASE	ENTRY

where:

SS The RS segment size required for the subprogram, coded as log base 2 less 2 (2 bits);

BASE base of RS segment for block in which subprogram is declared (14 bits);

ENTRY address in code of entry point (16 bits).

4 NUMBR The RS entry contains a number.

5 STRNG The RS entry contains the address into the string area of the string. The string is printed to the right of the RS.

6 ARREF The RS entry contains the address in array storage of an array element.

7 PBASE When a subprogram is called, this reference to the descriptor of the subprogram that was called is placed on the RS segment of the calling program.

8 BASE This attribute applies to the bottom three locations of a RS segment:

RET_ADDR		S_RS_PTR	
DYN_PTR		STAT_PTR	
SEG_TYPE	S_LL	CNT	SIZE

where:

SEG_TYPE Attribute code of segment (8 bits);

S_LL lexic level of this segment (8 bits);

CNT number of processes using the declarations of this segment (8 bits);

SIZE log base 2 of size of segment (8 bits);

V. INTERPRETATION OF THE STACK DUMP

DYN_PTR address of base of dynamically enclosing segment (16 bits);

STAT_PTR address of first segment below this segment in the stack whose lexic level is one less than that of this segment (16 bits);

RET_ADDR return address in CODE to which subprogram returns if SEG_TYPE is SPROG or the process index if SEG_TYPE is PROG (16 bits);

S_RS_PTR temporary storage for run stack pointer for this segment (16 bits).

9 PROG The RS entry is a program descriptor with the same format as for the SPROG.

10 BLOCK Used in SEG_TYPE (i.e., for declaration blocks). This attribute never occurs in the ATS.

11 CONSU The RS entry contains an index into the consumable resource area. The next available unit, or 'NO AVAILABLE UNITS' if there are none, is printed.

12 REUSA The RS entry contains a pointer into the reusable resource area. A message stating whether or not resource units are available is printed.

13 BLTIN The RS entry contains the number of the builtin function.

APPENDIX VI

IMPLEMENTATION RESTRICTIONS

The following are restrictions as of October 1975.

(1)	Code area	20K bytes
(2)	Run stack	8192 words
(3)	Number of string area descriptors	500
(4)	Number of consumable resources	32
(5)	Number of reusable resources	32
(6)	Number of consumable resource units (for all consumable resources)	512
(7)	Number of reusable resource units	32
(8)	Number of symbol table entries	400
(9)	Maximum number of declarations per block of variables, arrays, subprograms, and programs	32
(10)	Number of processes	32

APPENDIX VII

PROGRAM EXAMPLES

The following pages contain two sample TOPPS programs.

The first example consists of three simple processes which illustrate the essence of a spooling system. The first process read input data and adds it to an input queue. The second process receives input from this queue, performs a simple computation with it, and sends it to an output queue. The third process receives data from the output queue and prints it. The example also shows a compiler symbol table dump, and an interpreter stack dump.

The second example illustrates the use of the DELAY function.

TOPPS/MTS VERSION OF SEPTEMBER 27, 1974.

MTS line numbers · *line no's* · *object program location counter*

```
    1.  |    1|BEGIN                                                                      |  85
    2.  |    2|   /* AN EXAMPLE OF A TRIVIAL "SPOOLING SYSTEM". */                        |  87
    3.  |    3|                                                                           |  87
    4.  |    4|   /* ALSO CONTAINS AN EXAMPLE OF A STACK DUMP AND A SYMBOL TABLE LISTING */|  87
    6.  |    6|                                                                           |  87
    7.  |    7|   VARIABLE END_OF_FILE: /* VALUE PASSED WHEN END OF INPUT ENCOUNTERED */   |  87
    8.  |    8|   CONSUMABLE CARDS, LINES:                                                 |  89
    9.  |    9|                                                                           |  91
   10.  |   10|   PROGRAM INSPOOLER PRODUCING CARD_IMAGES IS                               |  91
   11.  |   11|   BEGIN                                                                   | 104 INSPOOLER
   12.  |   12|      VARIABLE CARD, /* THE VALUE TO BE INPUT */                           | 106 INSPOOLER
   13.  |   13|         EOF; /* END OF FILE SWITCH */                                     | 106 INSPOOLER
   14.  |   14|      REPEAT                                                               | 108 INSPOOLER
   15.  |   15|         EOF := ¬INPUT(CARD);                                              | 108 INSPOOLER
   16.  |   16|         IF ¬ EOF                                                          | 122 INSPOOLER
   17.  |   17|            THEN RELEASE(CARD_IMAGES, CARD)                                | 125 INSPOOLER
   18.  |   18|            ELSE RELEASE(CARD_IMAGES, END_OF_FILE)                         | 140 INSPOOLER
   19.  |   19|      UNTIL EOF                                                            | 154 INSPOOLER
   20.  |   20|   END;                                                                    | 157 INSPOOLER
   21.  |   21|                                                                           | 162
   22.  |   22|   PROGRAM USER PRODUCING LINE_IMAGES IS                                   | 162
   23.  |   23|   BEGIN                                                                   | 175 USER
   24.  |   24|      VARIABLE CARD;                                                       | 177 USER
   25.  |   25|                                                                           | 179 USER
   26.  |   26|      SUBPROGRAM REVERSE OF IMAGE IS                                       | 179 USER
   27.  |   27|      /* RECURSIVE PROCEDURE TO REVERSE THE STRING IMAGE */                | 190 REVERSE
   28.  |   28|         IF LENGTH(IMAGE) = 0                                              | 190 REVERSE
   29.  |   29|            THEN ''                                                        | 204 REVERSE
   30.  |   30|            ELSE CATENATE(REVERSE(SUBSTRING(IMAGE, 1)),                    | 214 REVERSE
   31.  |   31|                          SUBSTRING(IMAGE, 0, 1));                         | 239 REVERSE
   32.  |   32|                                                                           | 262 USER
   33.  |   33|      REPEAT                                                               | 262 USER
   34.  |   34|         CARD := REQUEST(CARDS);                                           | 262 USER
   34.25|   35|         RELEASE(LINE_IMAGES, CARD);                                       | 275 USER
   35.  |   36|         IF ¬ NUMERIC(CARD)                                                | 287 USER
   37.  |   37|            THEN RELEASE(LINE_IMAGES, REVERSE(CARD))                       | 295 USER
   38.  |   38|            ELSE 0                                                         | 315 USER
   41.  |   39|      UNTIL IF NUMERIC(CARD)                                               | 324 USER
   42.  |   40|               THEN CARD = END_OF_FILE                                     | 332 USER
   43.  |   41|               ELSE 0;                                                     | 341 USER
   44.  |   42|      TOGGLE('S') /* SHOW WHAT STACK DUMP LOOKS LIKE */                    | 355 USER
   45.  |   43|   END;                                                                    | 366 USER
   46.  |   44|                                                                           | 368
   47.  |   45|   PROGRAM OUTSPOOLER IS                                                   | 368
   48.  |   46|   BEGIN                                                                   | 381 OUTSPOOLER
   49.  |   47|      VARIABLE LINE;                                                       | 383 OUTSPOOLER
   50.  |   48|      REPEAT                                                               | 385 OUTSPOOLER
   51.  |   49|         OUTPUT(LINE := REQUEST(LINES))                                    | 385 OUTSPOOLER
   52.  |   50|      UNTIL IF NUMERIC(LINE)                                               | 402 OUTSPOOLER
   53.  |   51|               THEN LINE = END_OF_FILE                                     | 410 OUTSPOOLER
   54.  |   52|               ELSE 0                                                      | 419 OUTSPOOLER
   55.  |   53|   END;                                                                    | 429 OUTSPOOLER
   56.  |   54|                                                                           | 434
   57.  |   55|   END_OF_FILE := 2147483647;/* LARGEST NUMBER IS USED AS END-OF-FILE FLAG */ | 434
```

If there is input EOF = 0 ∴ then clause is executed


```
   58.  |   56|                                                                           | 445
   59.  |   57|   PROCESS INSPOOLER PRODUCING CARDS;                                      | 445
   60.  |   58|   PROCESS USER PRODUCING LINES;                                           | 455
   61.  |   59|   PROCESS OUTSPOOLER                                                      | 465
   62.  |   60|END                                                                        | 467
```

SYMBOL TABLE DUMP

NAME	LL.ON	TYPE	DECL	REFERENCE LINES
EOF	3,1	VARIABLE	13	15,16,19
CARD	3,0	VARIABLE	12	15,17
CARD_IMAGES	2,0	RESOURCE PARM	10	17,18
IMAGE	4,0	PARAMETER	26	28,30,31
REVERSE	3,1	SUBPROGRAM	26	30,37
CARD	3,0	VARIABLE	24	34,35,36,37,39,40
LINE_IMAGES	2,0	RESOURCE PARM	22	35,37
LINE	3,0	VARIABLE	47	49,50,51
OUTSPOOLER	1,5	PROGRAM	45	59
USER	1,4	PROGRAM	22	58
INSPOOLER	1,3	PROGRAM	10	57
LINES	1,2	CONSUMABLE	8	49,58
CARDS	1,1	CONSUMABLE	8	34,57
END_OF_FILE	1,0	VARIABLE	7	18,40,51,55
READ	0,12	SUBPROGRAM	0	
NUMERIC	0,11	SUBPROGRAM	0	36,39,50
DELAY	0,10	SUBPROGRAM	0	
RELEASE	0,9	SUBPROGRAM	0	17,18,35,37
REQUEST	0,8	SUBPROGRAM	0	34,49
QUANTUM	0,7	SUBPROGRAM	0	
TOGGLE	0,6	SUBPROGRAM	0	42
CATENATE	0,5	SUBPROGRAM	0	30
SUBSTRING	0,4	SUBPROGRAM	0	30,31
BYTE	0,3	SUBPROGRAM	0	
LENGTH	0,2	SUBPROGRAM	0	28
OUTPUT	0,1	SUBPROGRAM	0	49
INPUT	0,0	SUBPROGRAM	0	15

27 SYMBOLS.

END OF COMPILATION SEPTEMBER 24, 1974, TIME: 8:9:0.01.

NO ERRORS WERE DETECTED.

```
INPUT LINES READ     = 60
TOKENS USED          = 201
PRODUCTIONS USED     = 323
INSTRS. EMITTED      = 145
FREE STRING AREA     = 20224
COMPACTIFICATIONS    = 1
SIZE OF OBJECT CODE= 473
UNUSED CODE AREA     = 19527
NUMBER OF SYMBOLS    = 27
```

TOTAL TIME IN COMPILER   0:0:0.13.

```
 ...T INTERPRETER --- UNIVERSITY OF BRITISH COLUMBIA --- TOPPS/MTS VERSION OF SEPTEMBER 26, 1974.
 ...ION  BEGINS SEPTEMBER 29, 1974,  TIME: 8:42.54.

 ...RSE THIS PLEASE

 ... DI..NOSTIC DUMP REQUESTED AT LOCATION (364) IN PROCESS 3
     LA.. INSTRUCTION EXECUTED WAS CALL
                                        --> line 42

                 86|STRNG|        2 --> |   5 Parameter passed
                 85|REFEN|        9 --> |   reference to toggle function
   Reserve       84|SPROG|   2|  80|  188|
   Card          83|NUMBR|   2147483647|
                 82|BASE |       0|    87|
                 81|BASE |      56|    56|
                 80|BASE |BLOCK| 3| 2| 4|   Begin block

   Line.Image    59|REFEN|       37 --> |
                 58|BASE |       3|    54|
                 57|BASE |      32|    32|
                 56|BASE |PROG | 2| 1| 3|   Program 'User'

                 41|NUMBR|             0|   value of last statement in Main program
   Outspooler    40|PROG |   1|  32|  377|
   User          39|PROG |   1|  32|  171|
   Inspooler     38|PROG |   1|  32|  100|
   Lines         37|CONSU|        2 --> |    NEXT AVAILABLE UNIT IS  |NUMBR|    2147483647|
   Cards         36|CONSU|        1 --> |    NO UNITS AVAILABLE
   End-of-File   35|NUMBR|    2147483647|
                 34|BASE |       0|    41|
                 33|BASE |       0|     0|   Main program
                 32|BASE |BLOCK| 1| 2| 4|

   Read          15|BLTIN|            12|
   Numeric       14|BLTIN|            11|
   Delay         13|BLTIN|            10|
   Release       12|BLTIN|             9|
   Request       11|BLTIN|             8|
   Quantum       10|BLTIN|             7|
   Toggle         9|BLTIN|             6|
   Catenate       8|BLTIN|             5|
   Substring      7|BLTIN|             4|
   Byte           6|BLTIN|             3|
   Length         5|BLTIN|             2|
   Output         4|BLTIN|             1|
   Input          3|BLTIN|             0|
                  2|BASE |       1|    15|
                  1|BASE | 65535| 65535|   Outermost block
                  0|BASE |PROG | 0| 2| 5|
                    ATS         RS
                         RUN STACK
```

refer to the symbol table on page 39.

```
 ESAELP SIHT ESREVER

 2147483647

 EXECUTION ENDS WITH  NO PROCESSES DEADLOCKED.

 PROCESS  1 CLOCK TIME                 38
 PROCESS STARTED AT                     0
 PROCESS STATE                       DONE      Main program
 PROCESSOR TIME                        38
 BLOCKED TIME                           0
 TOTAL TIME                            38

 PROCESS  2 CLOCK TIME                 65
 PROCESS STARTED AT                    20
 PROCESS STATE                       DONE      inspooler
 PROCESSOR TIME                        56
 BLOCKED TIME                           0
 TOTAL TIME                            56

 PROCESS  3 CLOCK TIME                535
 PROCESS STARTED AT                    33
 PROCESS STATE                       DONE
 PROCESSOR TIME                       502
 BLOCKED TIME                           0
 TOTAL TIME                           502      User.

 PROCESS  4 CLOCK TIME                522
 PROCESS STARTED AT                    36
 PROCESS STATE                       DONE
 PROCESSOR TIME                        64      outspooler
 BLOCKED TIME                         422
 TOTAL TIME                           486

 EXECUTIONS ENDS SEPTEMBER 29, 1974,  TIME: 8:42.54.
 EXECUTION TIME      0:00:00.06.
 MAXIMUM USE OF RUN STACK:        408 WORDS
 AMOUNT OF RUN STACK UNUSED:     7784 WORDS
 MAXIMUM NO. OF STRINGS USED:      23
 FREE STRING AREA:              20324
 COMPACTIFICATIONS:                 0
```

TOPPS COMPILATION --- UNIVERSITY OF BRITISH COLUMBIA --- DATE: SEPTEMBER 29, 1974, TIME: 8:8:53.04.

TOPPS/MTS VERSION OF SEPTEMBER 27, 1974.

```
BEGIN                                                                      | 85
   /* PROGRAM TO SHOW THE USE OF THE DELAY FUNCTION.  */                   | 87
                                                                           | 87
   /* THE TWO PROCESSES WILL APPEAR TO EXECUTE THE LOOP AT DIFFERENT SPEEDS  | 87
      BECAUSE THE DELAY FUNCTION IS CALLED WITH DIFFERENT ARGUMENTS.    */ | 87
                                                                           | 87
   PROGRAM LOOP OF INCREMENT, NAME IS                                      | 87
   BEGIN                                                                   | 100 LOOP
      VARIABLE COUNT;                                                      | 102 LOOP
      COUNT := 1;                                                          | 104 LOOP
      REPEAT                                                               | 115 LOOP
         DELAY(INCREMENT);                                                 | 115 LOOP
         OUTPUT(COUNT, NAME)                                               | 124 LOOP
      UNTIL 10 < COUNT := COUNT + INCREMENT                                | 135 LOOP
   END;                                                                    | 150 LOOP
                                                                           | 158
   TOGGLE('A');  /* ACTIVATES DELAY MECHANISM */                           | 158
   PROCESS LOOP OF 1, 'FAST --->';                                         | 170
   PROCESS LOOP OF 2, 'SLOW <---'                                          | 189
END                                                                        | 204
```


END OF COMPILATION SEPTEMBER 29, 1974, TIME: 8:8:53.20.

NO ERRORS WERE DETECTED.

```
INPUT LINES READ    = 20
TOKENS USED         = 60
PRODUCTIONS USED    = 104
INSTRS. EMITTED     = 45
FREE STRING AREA    = 20224
COMPACTIFICATIONS   = 0
SIZE OF OBJECT CODE= 209
UNUSED CODE AREA    = 19791
NUMBER OF SYMBOLS   = 17

TOTAL TIME IN COMPILER   0:0:0.05.
```

TOPPS INTERPRETER --- UNIVERSITY OF BRITISH COLUMBIA --- TOPPS/MTS VERSION OF SEPTEMBER 28, 1974.

EXECUTION BEGINS SEPTEMBER 29, 1974. TIME: 8:8:59.14.

```
1 FAST --->
2 FAST --->
1 SLOW <---
3 FAST --->
4 FAST --->
3 SLOW <---
5 FAST --->
6 FAST --->
5 SLOW <---
7 FAST --->
8 FAST --->
7 SLOW <---
9 FAST --->
10 FAST --->
9 SLOW <---
```

EXECUTION ENDS WITH NO PROCESSES DEADLOCKED.

```
                                         DELAY TIME SIMULATION STATISTICS
PROCESS  1 CLOCK TIME          31        PROCESS  1 STARTED AT          0
PROCESS STARTED AT              0        FINISHED AT                    0
PROCESS STATE                DONE        TOTAL LIFE TIME                0
PROCESSOR TIME                 31        TOTAL DELAY TIME               0
BLOCKED TIME                    0        TOTAL BLOCKED TIME             0
TOTAL TIME                     31        AVERAGE UTILIZATION           0%

                                         DELAY TIME SIMULATION STATISTICS
PROCESS  2 CLOCK TIME         194        PROCESS  2 STARTED AT          0
PROCESS STARTED AT             24        FINISHED AT                   10
PROCESS STATE                DONE        TOTAL LIFE TIME               10
PROCESSOR TIME                170        TOTAL DELAY TIME              10
BLOCKED TIME                    0        TOTAL BLOCKED TIME             0
TOTAL TIME                    170        AVERAGE UTILIZATION         100%

                                         DELAY TIME SIMULATION STATISTICS
PROCESS  3 CLOCK TIME         119        PROCESS  3 STARTED AT          0
PROCESS STARTED AT             29        FINISHED AT                   10
PROCESS STATE                DONE        TOTAL LIFE TIME               10
PROCESSOR TIME                 90        TOTAL DELAY TIME              10
BLOCKED TIME                    0        TOTAL BLOCKED TIME             0
TOTAL TIME                     90        AVERAGE UTILIZATION         100%

EXECUTIONS ENDS SEPTEMBER 29, 1974.  TIME: 8:8:59.17.
EXECUTION TIME      0:0:0.03.
MAXIMUM USE OF RUN STACK:        88 WORDS
AMOUNT OF RUN STACK UNUSED:      6104 WORDS
MAXIMUM NO. OF STRINGS USED:     9
FREE STRING AREA:                20224
COMPACTIFICATIONS:               0
```