1979

Concur: a High-Level Language for Concurrent Programming

Karen Anderson

ROCHESTER INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND TECHNOLOGY

CONCUR: A HIGH-LEVEL LANGUAGE FOR CONCURRENT PROGRAMMING

A Thesis Submitted in Partial Fulfillment of the Master of Science in Computer Science Degree Program.

BY: Karen K. Anderson

Approved By:

_____

_____

_____

_____

Date: _____

TABLE OF CONTENTS

Page

Abstract 1

Introduction 2

The Language CONCUR

- Basic Symbols 4
- Data Types 5
- Data Structure 6
- Program Structure 6
- Constant Declarations 6
- Variable Declarations 6
- Process Declarations 7
- Expressions 8
- Assignment Statements 8
- Process Invocations 9
- IF Statements 9
- LOOP Statement 10
- Semaphore Manipulation 11
- Input/Output 13
- Statement Sequences 13
- Processes 14

The Use of CONCUR

- Overview 15
- A Programming Approach 17
- The Producer/Consumer Example 18
- Sequential Programming in CONCUR 24
- The Concurrent Quicksort Example 26

The Implementation of CONCUR

- Overview 35
- The Interpreter 36
- The Machine Instruction Set 40
- Code Generation 44

Conclusion 51

Appendix A CONCUR Syntax Graphs 59

Appendix B Sample Programs 65

Appendix C Source Listing of CONCUR Compiler 99

Appendix D Compiler Operating Instructions 151

Appendix E Compilation Errors 153

Appendix F Source Program with Object Code Listing 158

References 165

ABSTRACT

A language CONCUR is defined which permits the definition and initiation of asynchronous processes. The language was inspired by Modula, a language proposed by Wirth for realtime programming. CONCUR removes Modula's restrictions on the placement of process declarations and invocations in order to study the implications of process support more fully. Most of the other sophisticated features of Modula, such as modules, structure types, and procedures, have also been removed to focus attention on processes and their particular requirements. A general methodology is suggested for concurrent programming, and several sample programs are presented which demonstrate concurrent programming with CONCUR. Finally, a compiler is presented which translates CONCUR into the object language for a hypothetical machine. An interpreter for this object language is also included.

INTRODUCTION

With current technology processor speeds have approached the natural limitations of the materials from which the processors are made. Quantum reductions in program execution time through hardware breakthroughs, as have been seen in the past, cannot really be expected. Future significant increases in processing speed appear to be possible only through multiprocessing. (1)

Parallelism has been traditionally employed in hardware design to create faster equipment. More recently, parallelism has been recognized as applicable to operating systems to improve their efficiency. The last area for application of parallelism is within individual programs. A high-level language supporting the creation of concurrent processes within a program would facilitate effective parallel programming.

Such languages are beginning to emerge, two examples being Brinch Hansen's Concurrent PASCAL (2) and Wirth's Modula (3,4,5,6). This thesis presents a language called CONCUR, which was inspired primarily by Modula. CONCUR is not offered as an alternative to the production languages mentioned. It serves instead as a vehicle for exploring aspects of high-level language support for parallel processing. This study

should increase understanding of language design and compiler writing. In addition, selected concepts in operating systems and machine architecture will be explored. Finally, some of the implications of concurrent programming for programming methodology will be discussed.

THE LANGUAGE CONCUR

BASIC SYMBOLS

The basic symbols from which a CONCUR program is built are numbers, identifiers, keywords, and special symbols.

A number must be an integer. It can be signed or unsigned.

An identifier must begin with a letter and must contain only letters and digits. There is no limit to the number of characters in the identifier, but only the first twelve are used by the compiler.

Keywords are used to show explicitly the structure of a program. The keywords are reserved; i.e., they may not be used as identifiers. The following words are keywords.

AND	ELSIF	OF	THEN
ARRAY	END	OR	VAR
BEGIN	EXIT	PROCESS	VSIGNAL
CONST	IF	PWAIT	WHEN
DO	INTEGER	READ	WRITE
DUMP	LOOP	SEMA	XOR
ELSE	NOT	SET	

Special symbols serve as operators or delimiters. Some of them are composed of two characters which must appear contiguously.

ASSIGNMENT OPERATOR	:=
RELATIONAL OPERATORS	= <> < <= > >=
ARITHMETIC OPERATORS	+ - * /
DELIMITERS	: ; , ( ) (. .) (* *) .

Blanks, end-of-line characters, and comments are used to separate consecutive symbols but are otherwise ignored.

Comments are enclosed by (* *). They are listed but are not considered part of the program. Comments may appear between any two symbols in a program.

DATA TYPES

Two types of data are available in CONCUR: integer and semaphore.

A semaphore is a synchronization element which is manipulated by a restricted set of operators. A semaphore may be initialized by the SET operation and modified and tested by the PWAIT and VSIGNAL operations. Semaphores are explained in more detail on pages 11 through 13 and pages 16 and 17.

Since there is no Boolean type, relational operators produce integer results. True is represented as the integer one, and false is represented by the integer zero. In cases where the values cannot be closely controlled, any non-zero value is considered true, and a zero value is considered false.

DATA STRUCTURE

The only data structure permitted is the one-dimensional array. Array elements must be integers. Legal subscripts for an array range from one to the declared length. A specific element of an array can be referenced by the array name and an index enclosed in brackets (. .).

PROGRAM STRUCTURE

A CONCUR program consists of a block followed by a period. A block has a declaration part and a statement part. Within the declaration part symbolic constants, identifiers, and processes are defined. Within the statement part the algorithmic actions to be performed are specified. A process itself contains a block. Thus, a block is defined recursively.

CONSTANT DECLARATIONS

A constant declaration relates a symbolic name to a constant numeric value.

Examples of constant declarations are:

```
CONST    LIMIT = 10;
         ZERO = 0;
```

VARIABLE DECLARATIONS

A variable declaration establishes the name, type, and structure of a variable; however, no initialization of the variable takes place. The program can reference that variable by its name or, in the case of an array, its name and an index.

The index may be a constant, a variable, or an expression.

Examples of variable declarations are:

```
VAR   SUM:    INTEGER;
      LIST:   ARRAY 5 OF INTEGER;
      MUTEX:  SEMA;
```

Example variables are:

```
SUM   LIST(.I-1.)   MUTEX
```

PROCESS DECLARATIONS

A process declaration defines a program segment called a process that can be activated by an explicit call. The declaration first names the process and describes its parameters. Then the body of the process follows as a block. Finally, the process is delimited by its name.

Examples of process declarations are:

```
PROCESS  POWER  (AVALUE:  INTEGER,  ITSPOWER:  INTEGER)
                  {
                  {
                  {
         BLOCK
                  {
                  {
                  {
         POWER;


PROCESS  WAITLOOP;
                  {
                  {
                  {
         BLOCK
                  {
                  {
                  {
         WAITLOOP;
```

(Processes are discussed in greater depth on page 14.)

EXPRESSIONS

An expression is a rule for computing values. The rule is defined in terms of operators, operands (constants or variables), and parentheses. The operators are grouped into four priority classes. The relational operators (= <> < <= > >=) have the lowest priority. The arithmetic operators + and - and the logical operators OR and XOR have the next highest priority. The arithmetic operators * and / and the logical operator AND have the next highest priority. The logical operator NOT has the highest priority. Operators of equal priority are applied left to right. Parentheses can be used to override the normal order of application and to form compound logical expressions.

Examples of expressions are:

```
2*PI*D
PAYMENT < MINIMUMDUE
(HOURS = 0) OR ((GROSSPAY-TAXES-DEDUCTIONS)<0)
```

ASSIGNMENT STATEMENTS

An assignment statement places the value of an expression into a variable. The variable must be of type integer or array. If the expression results in a logical value, the value will be expressed automatically as an integer. True equals one and false equals zero.

Examples of assignement statements are:

```
START := 1
COST := LIST - DISCOUNT
AVG := (T1 + T2 + T3)/3
```

PROCESS INVOCATIONS

A process is called into execution when its name appears in the statement part of a block. The name may be followed by a parameter list of constants, variables, or expressions separated by commas. All parameters are passed by value.

Examples of process invocations are:

```
POWER (VALUE-AVERAGE,2)
WAITLOOP
```

IF STATEMENTS

An IF statement selects actions for execution on the basis of a logical expression. Three forms of the IF are allowed. The first (IF...THEN...) only specifies actions to be performed when the expression is true. The second (IF... THEN...ELSE...) specifies actions for both the true and false conditions. The third (IF...ELSIF...THEN...) permits nesting of IF statements. The ELSIF can be repeated many times within the IF to achieve any level of nesting. Each form of the IF is terminated with the keyword END.

Examples of IF statements are:

```
IF NETPAY < 0 THEN NETPAY := 0 END
IF A > B THEN
   ABSDIF := A - B
ELSE
   ABSDIF := B - A
END
IF CREDITHRS < 12 THEN
   TUITION := 82 * CREDITHRS
ELSIF CREDITHRS > 18 THEN
   TUITION := 966 + 82 * (CREDITHRS-18)
ELSE
   TUITION := 966
END
```

LOOP STATEMENT

A loop statement is a generalized form of iteration which can serve as any of the DO-UNTIL, DO-WHILE, or DO-WHILE-DO constructs. The WHEN portion of a loop statement acts as a conditional break. Placement of the WHEN at the bottom of a loop creates a DO-UNTIL; placement at the top of a loop creates a DO-WHILE; and placement in the midst of the loop creates a DO-WHILE-DO.

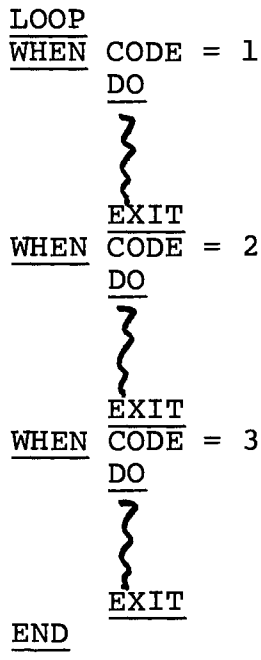
A bodyless loop with a series of WHEN portions can serve as a case statement, since each WHEN may be followed by a statement sequence that is performed once before exiting the loop. Any loop statement must terminate with the keyword END.

Examples of loop statements are:

```
LOOP
  ~
WHEN INDEX = DIMENSION EXIT
END

LOOP
WHEN LIST = EMPTY EXIT
  ~
END

LOOP
  ~
WHEN INPUT = ENDSIGNAL EXIT
  ~
END
```



SEMAPHORE MANIPULATION

As noted previously, semaphores can be referenced by only three kinds of statements. These are SET statements, PWAIT statements, and VSIGNAL statements.

A SET statement initializes a semaphore to the value of a specified expression. In addition, it clears the wait queue for the semaphore. The programmer must be very careful to include a properly placed SET statement for each semaphore declared. If a SET is omitted, neither the semaphore's value nor the pointer to its wait queue are initialized. The arbitrary values which they would have could cause a PWAIT or VSIGNAL to produce quite bizarre behavior. The value of the semaphore determines whether processes are suspended or

released and prompts manipulation of the wait queue. Also, since any pointer other than the end-of-queue value is interpreted as a process number, the next process performing a PWAIT on the semaphore would be linked to this "arbitrary process". Similarly, a VSIGNAL could enter the arbitrary process into the ready list. If a SET is misplaced or repeated, the initialization of the pointer to the wait queue would cause any processes already waiting on the semaphore to be lost when the SET is executed. The language does not prevent such misuse of the SET statement.

Examples of SET statements are:

```
SET (MUTEX,1)
SET (AVAIL,NOBUFFERS-2)
```

A PWAIT statement follows the formal definition of the P operation. Specifically, it decrements the semaphore by 1 and then tests the resultant value. If the semaphore is greater or equal to zero, execution continues; otherwise execution is suspended and the process containing the PWAIT is inserted into the wait queue for the semaphore. Clearly, the PWAIT operation can cause a semaphore to take on negative values. If a semaphore is negative, its absolute value shows the number of processes waiting on that semaphore.

A VSIGNAL statement follows the formal definition of the V operation. That is, it increments the semaphore by 1 and then tests the resultant value. If the semaphore is greater than zero, execution continues; otherwise execution continues

after another process is deleted from the wait queue for that semaphore and made ready for execution.

Examples of PWAIT and VSIGNAL statements are:

```
PWAIT(MUTEX)
VSIGNAL(AVAIL)
```

INPUT/OUTPUT

Input and output operations in CONCUR are very primitive. The operations are: read a single integer value from a new record, write a single integer value on a new line, and dump the active portions of descriptor store and data store. The argument of a READ statement must be either an integer variable or an array item. The argument of a WRITE statement can be an expression. The DUMP takes no arguments.

Examples of input/output statements are:

```
READ(MATRIX(.K.))
WRITE(RATE*TIME)
DUMP
```

STATEMENT SEQUENCES

A statement sequence is a series of statements separated by semicolons. The last statement in a sequence does not require a semicolon since the last statement will always be followed by a keyword. Statements in a statement sequence are executed sequentially in the order of their occurrence.

PROCESSES

Processes exist in CONCUR to permit the programming task to be broken into subtasks. Procedures in languages such as ALGOL, PL/I, and PASCAL serve the same purpose. Unlike a procedure, however, a process has an independent character. That is, a calling process does not wait until the called process completes, and many processes may be executed concurrently. Synchronization of process execution is accomplished by using semaphores.

All constants, variables, and processes defined within a process are local to that process. Communication among processes must take place through parameters (one-way communication with call by value) and variables which are "global" to both processes.

The main block may be considered a process which is called into execution at the beginning of program execution. The process terminates when the last statement in the block is executed, but the program does not end until all activated processes are terminated.

Syntax graphs for CONCUR are shown in APPENDIX A.

THE USE OF CONCUR

OVERVIEW

The value of concurrent programming is derived from the basic premise that a task or process can often be broken into subtasks or subprocesses which can be performed independently and simultaneously. Complete independence of subprocesses is generally an unrealistic expectation, however. (Two subtasks which are in no way dependent on each other might more logically be considered separate tasks altogether.) Instead, one can expect to encounter processes whose subprocesses are what Dijkstra calles "loosely connected" (7,p.52). That is, the subprocesses are independent of each other except for infrequent periods of explicit intercommunication. "Loosely connected" further implies that no assumptions can be made about the relative speeds of the subprocesses. Successful completion of the whole task cannot depend on one subtask consistently completing before another.

Intercommunication suggests sharing some resource, such as a variable, data structure, or I/O device. A shared resource which can only be used by one process at a time is

called a "critical resource" (1,p.27). That portion of a process which refers to a critical resource is called a "critical section" (7,p.53). In order to insure mutually exclusive access to a resource, a mechanism must be provided to synchronize the execution of parallel processes. This mechanism must satisfy two criteria:

1) If one or more processes attempt to enter their critical sections at the same time, at least one will be able to enter.
2. Only one process can be in its critical section at any given time. (1)

Dijkstra has proposed a synchronization mechanism called a semaphore. (7) A semaphore is a special-purpose integer which may be accessed only through two primitive operations--the P operation and the V operation. The P operation decreases the value of the semaphore by one. If the value of the semaphore becomes negative, the process executing the P operation cannot proceed. In other words, the process must wait. A waiting process is liberated when another process performs the V operation. The V operation increases the value of the semaphore by one. If the value of the semaphore remains negative or becomes zero, there must be at least one process waiting at a P operation. Thus, the V selects one waiting process and permits it to proceed. The process performing the V operation can also continue execution.

Semaphores may be classified as binary semaphores or general semaphores. There is no syntactic difference between the two types of semaphore, but they can be distinguished by their use and their maximum values. Binary semaphores are used to enforce mutual exclusion. The maximum value of a binary semaphore is one, since there is essentially only one "permit" for which processes must compete in order to enter their critical sections. General semaphores are used to allocate more plentiful resources. The maximum value of a general semaphore is determined by the number of units (e.g. buffers, disk drives, terminals,...) available. Note that the initial value of a semaphore is also crucial. The initial value indicates how many processes, if any, will be initially granted unrestrained access to the resource associated with the semaphore.

A PROGRAMMING APPROACH

From the above discussion, a general approach to writing a concurrent program in CONCUR can be suggested. Given a task, first identify the independent or loosely connected subtasks into which it can be divided. Each subtask should be accomplished by a process call.

The next step is to determine which resources are shared among the subtasks. These are critical resources. Then, pinpoint the critical sections within each process. (To take

full advantage of concurrent execution, the critical sections should be kept as short as possible.) Mutual exclusion must be enforced for the critical sections; thus, introducing a binary semaphore, surround each critical section between a P operation (a PWAIT instruction) and a V operation (a VSIGNAL instruction) on that semaphore. A distinct semaphore will be needed for each type of critical resource. If several units of a resource are available, introduce a general semaphore for resource allocation, in addition to the binary semaphore.

Establishing the intial value of each semaphore is the final crucial step. Initial values are established using a SET instruction. Mutual exclusion semaphores should be initialized to one so that one process will be permitted to enter its critical section. Resource allocation semaphores should be initialized to the number of units originally available. For example, if a pool of six tape drives are available at the start, the general semaphore used to allocate tape drives should be initialized to six.

THE PRODUCER/CONSUMER EXAMPLE

A classic problem to illustrate concurrent programming and the synchronization primitives is the producer/consumer problem. The problem involves passing blocks of data from a "producer" to a "consumer". A specific example of such a problem is the task of reading a list of integers from an

input device and writing the list on an output device. In this case, the producer is a process to bring an integer into memory, and the consumer is a process to take the integer from memory and send it out. The two processes share the memory location to which the producer moves the integer and from which the consumer takes it. If a single memory location is used, however, the producer and consumer processes are not loosely connected. In fact, they are quite dependent on each other's execution. The consumer cannot move the integer from memory until the producer has placed it there, and the producer cannot place the next integer in memory until the consumer has moved the previous one. A pool of memory locations must be introduced to decrease this dependency so that concurrent operation is feasible.

In the sample program shown in Appendix B the main program performs the original task. To allow parrallel processing, it accomplishes the task by invoking the producer and consumer subprocesses. Thus, the main process simply establishes and initializes the memory pool which the processes will share and then invokes them. The memory pool is implemented as a queue.

```
CONST NOBUFFERS=20;ENDSIGNAL=9999;EMPTY=99;
VAR   BUFFER:ARRAY NOBUFFERS OF INTEGER;
      INDEX:INTEGER;QUEUEBEGIN:INTEGER:QUEUEND:INTEGER;
.
.
.
BEGIN (*MAIN PROGRAM*)
      INDEX := 0;
      QUEUEBEGIN := EMPTY;
      LOOP
         INDEX := INDEX + 1;
         BUFFERPTR(.INDEX.) := 0
         WHEN INDEX=NOBUFFERS EXIT
      END;
      PRODUCER;
      CONSUMER
END.
```

The producer process reads an integer, places it in a free memory location and links that location to the queue. It continues reading and storing until an end signal is detected.

```
PROCESS PRODUCER;
      VAR INPUTDATA:INTEGER;
BEGIN
      LOOP
         READ(INPUTDATA);
         INDEX := 0;
         LOOP
             INDEX := INDEX +1
             WHEN BUFFERPTR(.INDEX.) = 0 DO
                 IF QUEUEBEGIN = EMPTY THEN
                    QUEUEBEGIN := INDEX;
                    QUEUEND := INDEX
                 END;
                 BUFFERPTR(.QUEUEND.) := INDEX;
                 BUFFER(.INDEX.) := INPUTDATA;
                 BUFFERPTR(.INDEX.) := EMPTY;
                 QUEUEND := INDEX
                 EXIT
         END;
         WHEN INPUTDATA=ENDSIGNAL EXIT
      END
END PRODUCER;
```

The consumer process takes an integer from the beginning of the queue and writes it out. It repeats the activity until it detects the end signal. The end signal is not written out.

```
PROCESS CONSUMER;
      VAR OUTPUTDATA:INTEGER;
BEGIN
      LOOP
         INDEX := QUEUEBEGIN;
         OUTPUTDATA := BUFFER(.INDEX.);
         QUEUEBEGIN := BUFFERPTR (.INDEX.);
         BUFFERPTR(.INDEX.) := 0
         WHEN OUTPUTDATA = ENDSIGNAL EXIT
         WRITE(OUTPUTDATA)
      END
END CONSUMER;
```

At this point the main program and its subprocesses contain all the instructions necessary to accomplish their task; but the program as written is prone to error. The memory pool established in the main program is shared by the producer and consumer processes. Therefore, it is a critical resource and those instructions that manipulate it in each process comprise critical sections. A binary semaphore must be introduced to guarantee mutually exclusive execution of these sections. Following the naming convention of Tsichritzis and Bernstein (1), the semaphore is called MUTEX. The main program must declare the semaphore.

```
...MUTEX:SEMA;
```

There is no need to use the semaphore in the main program when initializing the memory pool, since no subprocesses are invoked until after the initialization is complete. In the

producer process, however, a P operation on MUTEX must be included to make sure that the consumer is not trying to manipulate the memory pool at the same time. A V operation after the critical section lets the consumer proceed, if it has been waiting.

```
READ (INPUTDATA);
PWAIT(MUTEX);
INDEX := 0;
LOOP
.
.
.
END;
VSIGNAL(MUTEX);
```

similar additions are also required in the consumer process to insure mutual exclusion.

```
LOOP
     PWAIT(MUTEX);
     .
     .
     .
     VSIGNAL(MUTEX)
     WHEN OUTPUTDATA=ENDSIGNAL EXIT
```

Although the MUTEX semaphore will prevent the producer and consumer processes from manipulating the memory pool at the same time, one more restriction needs to be placed on each process. The producer should not be allowed to move an integer into memory unless an empty location is available. The consumer should not attempt to take an integer from memory unless one is there to be taken. Two general sema-

phores are needed to implement these restrictions. The semaphore AVAIL shows the number of empty memory locations, and the semaphore FULL shows the number of full memory locations. The producer must perform a P operation on AVAIL before moving the input data to memory. After moving the data, it must signal the presence of another full location by performing a V operation on FULL.

```
READ (INPUTDATA);
PWAIT (AVAIL);
PWAIT (MUTEX);
.
.
.
VSIGNAL(MUTEX);
VSIGNAL(FULL)
```

The inverse is true for the consumer, which must perform a P operation on FULL before attempting to take data from memory and then perform a V operation on AVAIL after taking the data.

```
LOOP
     PWAIT (FULL);
     PWAIT (MUTEX);
     .
     .
     .
     VSIGNAL(MUTEX);
     VSIGNAL(AVAIL);
```

Finally, the initial value for each semaphore must be established. Since AVAIL represents the number of empty memory locations, AVAIL should be set to the total number of

locations. FULL should be initialized to zero because there are no full locations at the start. MUTEX must be initialized to one. All the semaphores must be initialized before any processes are invoked, i.e., in the main program.

```
SET (AVAIL,NOBUFFERS);
SET (FULL,0);
SET (MUTEX,1);
```

The producer/consumer program is now complete. We can be confident that it will perform its task regardless of the relative execution speeds of the two processes. It should also take maximum advantage of parallel execution. The complete listing of the producer/consumer program is given in Appendix B.

SEQUENTIAL PROGRAMMING IN CONCUR

Many tasks remain which do not lend themselves to concurrent programming. For them, the sequence of subtask performance is crucial. In such cases, when a subtask is begun, the main program must wait until the process completes before invoking the next subtask.

In many concurrent programming languages, such as MODULA and extended ALGOL for the Burroughs 6700, separate facilities exist for defining sequential procedures and concurrent processes. CONCUR, however, has only one method for process definition. A process so defined is always considered con-

current. A synchronization mechanism must be employed to force the main program to wait while the process executes. For this purpose, a binary semaphore will suffice. The main program follows each process call with a P operation on the binary semaphore. If the process is not complete, the P will cause the main program to wait. The main program will continue when the process performs a V operation on the semaphore. For synchronization to occur properly the binary semaphore must be initialized to zero.

Appendix B includes a listing of a sequential program. The program simply reads in a list of integers, inverts the list, and writes out the inverted list. The program uses a process to accomplish the inversion. Clearly this straightforward approach requires that the list be inverted before it is printed. Therefore, the main program must wait until the INVERT process is complete. The semaphore DONE is defined and initialized in the main program.

```
DONE:SEMA;
.
.
.
SET(DONE,0);
```

After calling the INVERT routine, the main program performs a P operation on DONE

```
INVERT(START,STOP);
PWAIT(DONE);
```

The INVERT process cooperates by performing a V operation on DONE when it ends.

```
        VSIGNAL(DONE)
END INVERT;
```

In this way the main program will indeed wait until the INVERT process has finished, and the sequential execution required by the algorithm will be enforced.

Note that CONCUR can simulate the execution protocol of traditional languages but the inverse is not true. This shows that sequential programming is a proper subset of parallel programming.

THE CONCURRENT QUICKSORT EXAMPLE

C.A.R. Hoare has developed an internal sort algorithm called the "quicksort" (8). This algorithm basically partitions the array to be sorted around an arbitrarily-chosen pivot point so that all the elements in the array before the pivot point are less than or equal to the pivot element and all the elements in the array after the pivot point are greater than or equal to the pivot element. Once the array is partitioned in this fashion, the quicksort algorithm can be applied recursively to each of the partitions until all the segments to be sorted contain only one element. At this point the entire array has been sorted.

To implement this algorithm, a main program that controls input of the unsorted array and output of the sorted result needs only to invoke a QUICKSORT process and pass it the beginning and ending index for the array. The main program in this example reads in the array and counts the number of elements it reads. To satisfy an assumption for the partitioning procedure (discussed below), the main program appends to the array a trailer with a value that exceeds any of the actual array values. This trailer element is not included in the element count, however. The main program code follows:

```
CONST MAXSIZE=100; ENDSIG=9999;
VAR A:ARRAY MAXSIZE OF INTEGER;
     ELEMENTCNT:INTEGER;
     INDEX:INTEGER;
BEGIN (*MAIN PROGRAM*)
     ELEMENTCNT:=0;
     LOOP
          ELEMENTCNT:=ELEMENTCNT+1;
          READ (A(.ELEMENTCNT.))
          WHEN A(.ELEMENTCNT.)=ENDSIG EXIT
     END;
     ELEMENTCNT:=ELEMENTCNT-1;
     QUICKSORT(1,ELEMENTCNT);
     INDEX:=1;
     LOOP
          WRITE (A(.INDEX.));
          INDEX := INDEX+1
          WHEN INDEX > ELEMENTCNT EXIT
     END
END.
```

After the quicksort process is started by main, the quicksort calls itself recursively until it passes itself a single element partition. To end the recursion, the quicksort

must first test whether it indeed has elements to sort; if not, it simply exists without calling itself again. If it has at least two elements, the quicksort invokes the partitioning procedure to divide the array. The quicksort then calls itself to sort the elements below the pivot point and calls itself again to sort the elements above the pivot point. The pivot element itself is in the appropriate position; therefore, it is not included in future sorts. Since the pivot point returned is chosen arbitrarily, one of the two resulting partitions may be empty. Thus, the quicksort must compare the pivot point to the partitioned segment's end points to avoid calling itself to sort any empty partition.

```
PROCESS QUICKSORT (M:INTEGER,N:INTEGER);
     VAR I:INTEGER;
BEGIN (*QUICKSORT*)
     IF N > M THEN
          PARTITION (M,N+1);
          IF M < I THEN
               QUICKSORT (M,I-1)
          END;
          IF I < N THEN
               QUICKSORT (I+1,N)
          END
     END
END QUICKSORT;
```

The partitioning procedure assumes that the last element in the segment to be partitioned is greater than or equal to all other elements in the segment. Thus, initially, the main program appends a trailer element to the array; and, subsequently, the quicksort passes to the partitioning

procedure the index for the first element in the segment to be sorted and the index for the next element beyond the end of that segment.

There are many methods for selecting the pivot element for the partitioning procedure. The choice of pivot element can seriously influence the efficiency of the quicksort (9). The purpose of this example, however, is to explore the possible application of concurrent programming to the quicksort algorithm; therefore, the first element of the segment is chosen as the pivot element.

After selecting the pivot element, the partitioning procedure must exchange the position of this element with others in the segment until the pivot element is placed in its appropriate position. This position is the pivot point; the elements before it are less than or equal to the pivot element, and the elements after it are greater than or equal to the pivot element. Positioning of the pivot element is accomplished by the following algorithm: (1) the segment is scanned from the second element for an element that is greater than or equal to the pivot element; (2) the segment is scanned from the last element for an element that is less than or equal to the pivot element; (3) if the index of the element found in (1) is less than or equal to the index of the element found in (2), the elements are exchanged and the scanning continues; otherwise the pivot element is exchanged with the

element found in (2) and the partitioning procedure is complete. The partitioning procedure uses an exchange procedure to exchange specified elements. Its code is included with the partition code below.

```
PROCESS PARTITION (M:INTEGER,N:INTEGER);
     VAR X:INTEGER;
     PROCESS EXCHANGE (M:INTEGER,N:INTEGER);
          VAR TEMP:INTEGER;
     BEGIN (*EXCHANGE*)
          TEMP:=A(.M.);
          A(.M.):=A(.M.);
          A(.M.):=TEMP
          END EXCHANGE;
     BEGIN (*PARTITION*)
          X:=A(.M.);
          I:=M;
          LOOP
               LOOP
                    I:=I+1
                    WHEN A(.I.) >= X EXIT
               END;
               LOOP
                    N:=N-1
                    WHEN A(.N.) <= X EXIT
               END
               WHEN I > N EXIT
               EXCHANGE (I,N)
          END;
          EXCHANGE (M,N);
          I:=N
     END PARTITION;
```

The quicksort presents an excellent opportunity for concurrent programming. Although each invocation of the quicksort works on the array to be sorted, no two invocations work on the same segment or overlapping segments of the array. After the array has been partitioned, the quicksorts for the partitions of the array can proceed independently. Since all

processes in CONCUR are concurrent by default, the existing invocations of the quicksort already take advantage of this independence.

Within the quicksort, however, there are procedures which must be executed sequentially. For example, the quicksort cannot call itself recursively until the partitioning is complete. A semaphore must be declared in the quicksort to force it to wait while the partition procedure executes. The quicksort sets the semaphore to zero, invokes PARTITION, and then waits on the semaphore. The following code is added:

```
PARTDONE:SEMA;
.
.
.
SET (PARTDONE,0);
     .
     .
     .
     PWAIT(PARTDONE);
```

The partition procedure simply signals when it is complete.

```
VSIGNAL(PARTDONE)
```

Within the partition procedure the exchange procedure must be employed sequentially. That is, further scanning of the segment or completion of the partition procedure cannot occur until an exchange is accomplished. The partition procedure must declare a semaphore on which to wait until an exchange is complete. It must set the semaphore to zero

initially and then wait on it after every call to EXCHANGE.

```
EXCHDONE:SEMA;
      .
      .
      .
SET (EXCHDONE,0);
      .
      .
      .
      PWAIT(EXCHDONE)
            .
            .
            .
PWAIT(EXCHDONE);
```

The exchange procedure signals when it is complete:

```
VSIGNAL(EXCHDONE).
```

With the addition of these two semaphores most of the synchronization for the quicksort is complete. A challenging problem remains to be solved, however. Although it only invokes the quicksort process once, the main program must wait until all invocations of the quicksort process are complete. If it does not wait, it might print the array when it is only partially sorted. A simple semaphore, such as PARTDONE or EXCHDONE, is not sufficient. A mechanism must be added so that only the last quicksort process to complete will signal the main program.

In this example a counter is introduced to count the number of quicksort processes that have not completed. The main program and the quicksort itself must increment the

counter before each invocation of the quicksort. The quicksort process decrements the counter upon exiting. If the counter is decremented to zero, the process knows that it is the last quicksort and signals the main program to continue. The counter is declared in the main program along with the semaphore used to signal completion of the sort. The counter is initialized to one immediately before QUICKSORT is invoked. The semaphore is set to zero.

```
QUICKDONE:SEMA;
QUICKSIGS:INTEGER;
     .
     .
     .
SET(QUICKDONE,0);
     .
     .
     .
QUICKSIGS:=1;
```

In the quicksort process the counter is incremented before each invocation and decremented upon exiting the process. A count of zero prompts the quicksort to signal the main program via QUICKDONE.

```
QUICKSIGS:=QUICKSIGS+1;
     .
     .
     .
QUICKSIGS:=QUICKSIGS+1;
     .
     .
     .
QUICKSIGS:=QUICKSIGS-1;
IF QUICKSIGS <= 0 THEN  VSIGNAL (QUICKDONE)END
```

One last synchronization problem remains. Several copies of the quicksort may try to access the counter at the same time. To protect this critical resource, a mutual exclusion semaphore must be employed. The main program declares and initializes the semaphore.

```
CHECKSIGS:SEMA;
     .
     .
     .
SET (CHECKSIGS,1);
```

The main program does not need to use the semaphore when QUICKSIGS is initialized because no other processes have been invoked at that time. The quicksort process, however, must surround the increments, decrement, and test of the counter with a wait and signal on that semaphore.

```
PWAIT(CHECKSIGS);
QUICKSIGS := QUICKSIGS+1;
VSIGNAL(CHECKSIGS);
     .
     .
     .
PWAIT(CHECKSIGS);
QUICKSIGS := QUICKSIGS+1;
VSIGNAL(CHECKSIGS);
     .
     .
     .
PWAIT(CHECKSIGS);
QUICKSIGS:=QUICKSIGS-1;
IF QUICKSIGS <= 0 THEN VSIGNAL (QUICKDONE) END;
VSIGNAL(CHECKSIGS);
```

With the introduction of CHECKSIGS all the required synchronization mechanisms have been added to the quicksort program. A complete listing of the program as well as two sample runs are included in Appendix B.

THE IMPLEMENTATION OF CONCUR

OVERVIEW

The CONCUR compiler is a one-pass compiler written in PASCAL for the Xerox Sigma 9 computer. The compiler uses the technique of recursive descent. PASCAL was chosen as the implementation language because it is a well-structured, high-level language with recursion and facilities for programmer-defined data types. Recursive descent is a popular top-down parsing technique which can be readily programmed from a language's syntax graphs.

The compilation produces a hypothetical object language rather than Sigma 9 machine or assembly language. An interpreter is included with the compiler to execute the object code. This approach simplifies code generation since the compiler is not constrained by the hardware characteristics of the specific computer on which it is run. Indeed, such freedom improves the portability of the compiler. For this reason, many compiler writers such as Richards (10) and Waite (11) actually promote this approach for production compilers.

The scheduling algorithm used by the interpreter is a version of round robin. From the list of ready processes the interpreter sequentially selects the next process to be executed. The interpreter also generates a "random" number between 1 and 10 from the tens position of the current clock value to determine how many instructions will be performed within the selected process.

The CONCUR compiler is not presented as a "production compiler". Compiler options are almost non-existent, error indications are primitive, and error recovery is unimaginative. The compiler does, however, create a reasonable source code listing. In addition, a compilation trace, an object code list, and/or an execution trace can be produced by activating the appropriate output files.

A source listing of the CONCUR compiler is presented in Appendix C. Operating instructions for the compiler are given in Appendix D. Error numbers are listed and explained and a sample program with errors is shown in Appendix E.

THE INTERPRETER

The CONCUR interpreter executes the object code generated for the hypothetical machine. The machine consists of an accumulator, an index register, three stores (program store, descriptor store, and data store), and four pointers (a pointer to the current process being executed, a pointer to the current

instruction within that process, a pointer to the list of processes ready to execute, and a pointer to the list of free data store segments). This architecture is illustrated in Figure 1.

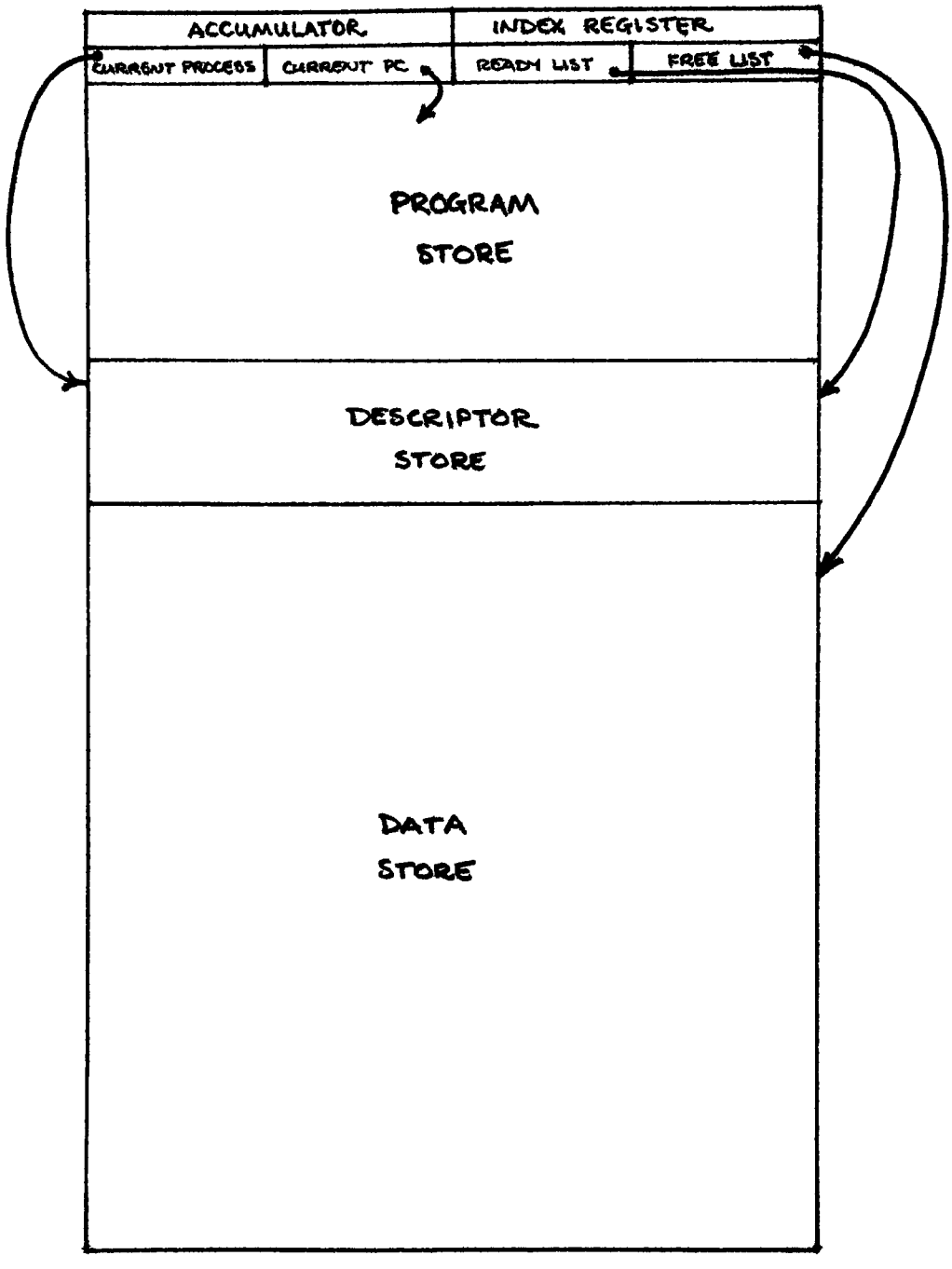


Figure 1
Hypothetical Machine Organization

Program store contains the machine instructions generated by the compiler. The interpreter is only permitted to read from this store. Descriptor store contains a descriptor for the main process and any other processes that have been invoked. A descriptor is a fixed-size data structure which contains the following information:

1. The current status of the descriptor or process (free, ready, waiting, or terminated).
2. The number of direct descendents of the process.
3. The static link to the process within which this process was defined.
4. The forward link to the next process in the ready or wait list.
5. The backward link to the preceding process in the ready or wait list.
6. The pointer to the beginning of this process's data area in data store.
7. The size of the data area.
8. The program counter.
9. The index register.
10. The accumulator.

The descriptors of all processes currently ready for execution are linked together in a READY list.

Data store is an array of integers which are allocated dynamically as processes are invoked and terminated. Within data store the integer data type is represented as a single

element; the semaphore data type is represented as a pair of adjacent elements; and the array data structure is represented as a group of contiguous elements. A FREE list is maintained for data store management. List manipulation and data store allocation are accomplished using versions of Knuth's first-fit and liberation algorithms (12). Since several executions of a process can proceed concurrently, each invocation of a process requires creation of a new process descriptor and allocation of a corresponding data store. In addition, because these process variables must be available to subordinate processes, the descriptor and data store cannot be released until all the subordinate processes, as well as the process itself, are terminated.

Dynamic storage allocation prevents the compiler from supplying absolute addresses in the object code. Thus, addresses of variables for the hypothetical machine are expressed as displacements from the beginning of some process data store. Since a process may refer to variables in its antecedents also, a level number must accompany each displacement to identify the specific data store to which this displacement applies. The actual starting address for the specific data store is obtained by following the static link chain up the appropriate number of levels.

THE MACHINE INSTRUCTION SET

The hypothetical machine responds to thirty-one instructions. These instructions were selected to complement the features of CONCUR. The instructions fall into five groups, each of which has a unique format.

In the descriptions below, ACC represents the accumulator, PC represents the program counter, XREG represents the index register, ADDR represents a data or program store address, and ( ) indicates its contents.

The format of Group 1 is OPCODE, INDEX FLAG, LEVEL, and DISPLACEMENT. If the index flag is set, the address represented by the level and displacement is modified by the contents of the index register. The following list gives the function of each instruction in Group 1.

ADD	ACC ← ACC + (ADDR)
SUB	ACC ← ACC - (ADDR)
RSUB	ACC ← (ADDR) - ACC
MULT	ACC ← ACC * (ADDR)
DIV	ACC ← ACC ÷ (ADDR)
RDIV	ACC ← (ADDR) ÷ ACC
EQL	if ACC = (ADDR) then
	ACC ← 1
	else
	ACC ← 0
NEQ	if ACC ≠ (ADDR) then
	ACC ← 1

	else
	ACC ← 0
LSS	if ACC < (ADDR) then
	ACC ← 1
	else
	ACC ← 0
LEQ	if ACC $\leq$ (ADDR) then
	ACC ← 1
	else
	ACC ← 0
GTR	if ACC > (ADDR) then
	ACC ← 1
	else
	ACC ← 0
GEQ	if ACC $\geq$ (ADDR) then
	ACC ← 1
	else
	ACC ← 0
AND	ACC ← ACC $\wedge$ (ADDR)
OR	ACC ← ACC $\vee$ (ADDR)
XOR	ACC ← ACC $\overline{\vee}$ (ADDR)
LDX	XREG ← (ADDR)
STX	(ADDR) ← XREG
LDA	ACC ← (ADDR)
STA	(ADDR) ← ACC
PWAIT	(ADDR) ← (ADDR) - 1

```
                    if (ADDR) < 0 then
                           the process waits
                    else
                           it continues
VSIGNAL             (ADDR) ← (ADDR) + 1
                    if (ADDR) <= 0 then
                           restart a waiting process
                           and continue
                    else
                           continue alone
```

It is crucial that instructions requiring two or more steps, especially PWAIT and VSIGNAL, are executed indivisibly. This requirement pertains to instructions in every group. TERM, DUMP, and CREATE are examples of other instructions that depend heavily on this requirement.

The format of Group 2 instructions is OPCODE and INSTRUCTION ADDRESS. The instruction address is simply the displacement from the start of the process code in program store. The function of each of the two instructions is as follows.

```
BR                  PC ← ADDR
BRF                 if ACC = 0 then
                           PC ← ADDR
```

Group 3 only contains one instruction. This instruction is a load immediate (LDI) with the format OPCODE and NUMERIC CONSTANT. The instruction loads the constant into the accumulator.

Group 4 contains the six instructions which require no operands. Thus, their format is simply OPCODE. The name and function of each is given below.

NOT	ACC ← ¬ ACC
STAX	XREG ← ACC
TERM	the process status is set to TERMINATED
	the process is deleted from the READY list
	the descendent count is decremented for each process linked to this one in the static link
	the descriptor store and data store are liberated for any terminated process in the chain whose descendent count becomes 0
READ	an integer is read into the ACC from M:SI
WRITE	the contents of the ACC are printed on M:LO
DUMP	the values for the four pointers are printed on the M:LO followed by each descriptor and its corresponding data store

Finally, the CREATE instruction is the sole occupant of Group 5. It has the format OPCODE, INITIAL PC, STATIC LINK, DATA STORE SIZE, PARAMETER COUNT, and PARAMETER PASS AREA START. The CREATE instruction creates a process descriptor, allocates the required data store, increments the descendent count of all processes related via the static link

chain, and copies any parameters into the first part of the data store segment.

CODE GENERATION

The relationship between CONCUR statements and the hypothetical machine instructions is relatively straight forward. The following examples illustrate this relationship.

The assignment statement has the general form:

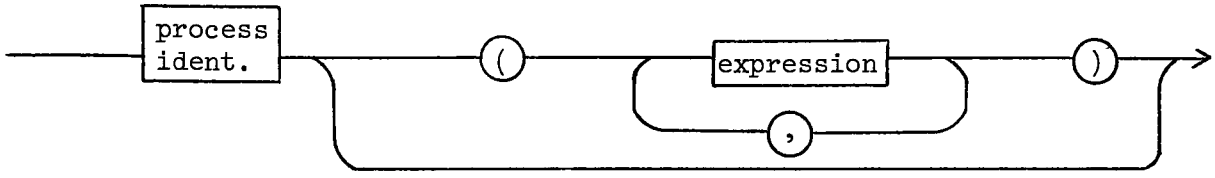
The compiler must generate code first to evaluate the expression and then to store its result in the specified variable. For example:

```
COST := LIST-DISCOUNT  becomes  LDA    LIST      ┐
                                STA    TEMP      │
                                LDA    DISCOUNT  } evaluating the expression
                                RSUB   TEMP1     ┘
                                STA    COST      } storing the result
```

Since the hypothetical machine is a single accumulator machine rather than a stack machine, the code generated, particularly for expression evaluation, makes frequent use of temporary variables. The compiler must keep track of the maximum number of temporary variables used by each process so that space for them may be allocated along with space for declared variables. Temporary variables occupy consecutive memory locations; that is, $TEMP_{i+1}$ is always allocated following

$TEMP_i$. No attempt is made to optimize the code generated.

A process is invoked when its name is used in the statement part of a block.



In this case, if the process has parameters, the first code that is generated stores each parameter in a temporary storage location in the calling process. The actual invocation is accomplished with a CREATE instruction. For example:

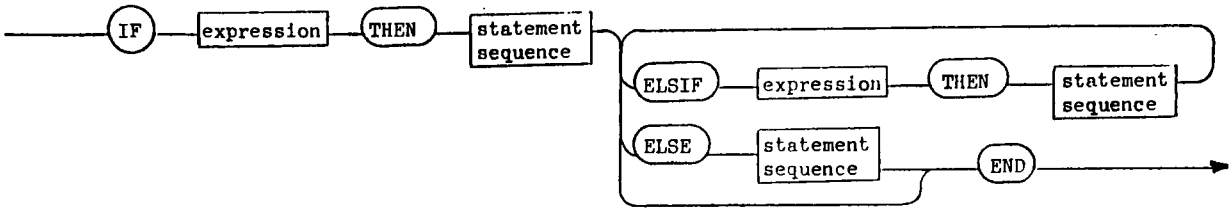
POWER (VALUE-AVERAGE,2) becomes

```
LDA     VALUE   ┐
STA     TEMP1   │
LDA     AVERAGE │ evaluating the expression
RSUB    TEMP1   ┘
STA     TEMP1 }storing the resulting parameter
LDI     2
STA     TEMP2 }storing the second parameter
CREATE  POWER,<static link>,<data store size>,2,TEMP1 }invoking the process
```

and WAITLOOP becomes CREATE WAITLOOP,<static link>,<data store size>,0,<null>.

Although only a simple END marks the end of a process, this particular END does prompt code generation. Specifically, a TERM instruction is generated to terminate the process. Thus, a process is invoked through a CREATE generated at its call and ended by a TERM which follows its last executable statement.

CONCUR'S IF statement is defined as:



The compiler generates code to evaluate the initial expression. A conditional branch must follow to select the appropriate statement sequence. When the ELSIF is employed, expression evaluation and conditional branching are required several times. For example:

```
IF CREDITHRS < 12 THEN
     TUITION := 82 * CREDITHRS
ELSIF CREDITHRS > 18 THEN
     TUITION := 966 + 82 * (CREDITHRS-18)
ELSE
     TUITION := 966
END
```

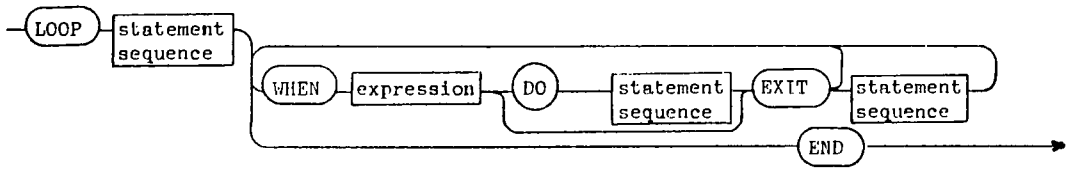
becomes

```
        LDA   CREDITHRS  ┐
        STA   TEMP1      │
        LDI   12         │ evaluating the first expression
        GTR   TEMP1      ┘
        BRF   ADDR1      } conditional branch
        LDI   82         ┐
        STA   TEMP1      │
        LDA   CREDITHRS  │
        MULT  TEMP1      │ first statement sequence
        STA   TUITION    │
        BR    ADDR3      ┘
ADDR1   LDA   CREDITHRS  ┐
        STA   TEMP1      │
        LDI   18         │ evaluating the second expression
        LSS   TEMP1      ┘
```

```
        BRF   ADDR2 }conditional branch
        LDI   966
        STA   TEMP1
        LDI   82
        STA   TEMP2
        LDA   CREDITHRS
        STA   TEMP3          } second statement sequence
        LDI   18
        RSUB  TEMP3
        MULT  TEMP2
        ADD   TEMP1
        STA   TUITION
        BR    ADDR3
ADDR2   LDI   966
        STA   TUITION        } third statement sequence
ADDR3   ___   }end
```

Notice that the conditional branches require forward references to later parts of the IF statement. Since the CONCUR compiler is a one-pass compiler, a record must be kept of these references. A FIXUP routine is used to supply the appropriate addresses when they are available. This routine is also used for the LOOP statement and for process invocations encountered before the process is fully defined.

The LOOP statement's format is:

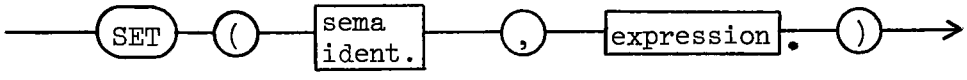


This construct is a general purpose loop. Thus, it may require expression evaluation first or code for a statement sequence, depending on how it is used. As mentioned above, the LOOP generates forward references requiring FIXUP. Two forward references occur within the ocde for each WHEN clause.

For example:

```
LOOP                 becomes ADDR1 LDA  CODE
WHEN CODE = 1                      STA  TEMP1   } evaluating the first expression
                                   LDI  1       }
                                   EQL  TEMP1   }
                                   BRF  ADDR2     conditional branch
DO                                              }
     ≈                                          } code for the first statement
     EXIT                          BR   ADDR5     exit
WHEN CODE = 2              ADDR2 LDA  CODE      }
                                   STA  TEMP1   } evaluating the second expression
                                   LDI  2       }
                                   EQL  TEMP1   }
                                   BRF  ADDR3     conditional branch
DO                                              }
     ≈                                          } code for the second statement
                                                } sequence
     EXIT                          BR   ADDR5     exit
WHEN CODE = 3              ADDR3 LDA  CODE      }
                                   STA  TEMP1   } evaluating the third expression
                                   LDI  3       }
                                   EQL  TEMP1   }
                                   BRF  ADDR4     conditional branch
DO                                              }
     ≈                                          } code for the third statement
                                                } sequence
     EXIT                          BR   ADDR5     exit
                           ADDR4   BR   ADDR1     looping back
END                        ADDR5   _____          end
```

Semaphore manipulation is accomplished through the SET, PWAIT, and VSIGNAL statements. The PWAIT and VSIGNAL instructions are defined as primitives in the hypothetical machine; thus, the CONCUR PWAIT and VSIGNAL translate directly into their machine language counterparts. This insures that they are uninterrupted. The SET statement has no corresponding primitive, however. The format of the SET is



The statement translates into the instructions required to evaluate and store the expression to which the semaphore is initialized. Finally, an additional load and store are required to initialize the waiting list for that semaphore. For example:

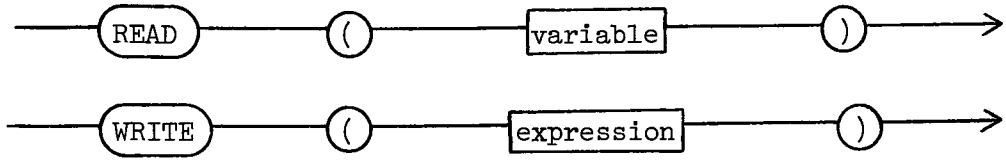
```
SET(MUTEX,1) becomes     LDI  1          } initializing the semaphore value
                         STA  MUTEX
                         LDI  9999*      } starting the waiting list empty
                         STA  MUTEX+1
```

*9999 is the value used to signal end-of-list.

The set statement is not indivisible. When correctly used, it will be executed before any processes depending on the semaphore are created. Thus, uninterrupted execution is not necessary.

Although machine instructions exist with names matching each of the CONCUR I/O statements (READ, WRITE, and DUMP), only the DUMP statement translates directly into its corresponding machine instruction. The READ and WRITE machine instructions

use the accumulator as the destination or source of the I/O operation. The READ and WRITE statement syntax shows that a more general destination or source may be specified in CONCUR itself.



Thus, for the READ statement the compiler must first generate a READ instruction and then include a STA to move the accumulator to the desired variable's address. For the WRITE statement the compiler generates code to evaluate the expression and then generates a WRITE instruction. For example:

```
READ(MATRIX(.K.)) becomes      READ    }reading the value into ACC
                               STA     TEMP1  }saving it
                               LDA     K
                               STAX      }evaluating the index expression
                               LDA     TEMP1
                               STA     MATRIX(K)    }storing the input value

and WRITE(RATE*TIME) becomes   LDA     RATE   ┐
                               STA     TEMP1  │
                               LDA     TIME   │ evaluating the expression
                               MULT    TEMP1  ┘
                               WRITE   }writing it out
```

The CONCUR compiler produces an object code list upon request. The JCL required to obtain this list is specified in Appendix D. Appendix F presents a complete CONCUR program source list followed by its corresponding object code. All of the examples in this section were verified through the compiler option.

CONCLUSION

This project has explored the syntax and run-time support features that can be supplied with a high-level language to permit concurrent programming. The minimal set of such facilities is the following:

1) the ability to define an asynchronous process,

2) a descriptor for an active process that maintains its status and data independent of other processes or other invocations of the same process,

3) the ability to create and destroy active processes, and

4) a mechanism for synchronizing processes when necessary.

The specific methods used in CONCUR to implement these facilities are not the only methods available. For example, Dijkstra extends ALGOL 60 with a construction he calls "a parallel compound" to permit definition of parallel processes (7,p.53). Bobrow and Wegbreit propose the use of a stack rather than dynamic storage allocation of linked blocks for maintenance of process descriptors (13). Tsichritzis and Bernstein discuss how processes can be created and destroyed and list a variety of synchronization mechanisms (1).

The synchronization feature appears to be the one most worthy of attention in CONCUR. The chosen implementation for the other features may lead to inefficiencies, but the use of semaphores for synchronization actually introduces significant potential for error. As noted earlier, the correct functioning of semaphores is particularly dependent on their proper initialization. Errors can also occur when the P and V operations on a semaphore are misplaced or omitted. Implementation of more tightly controlled synchronization, such as that included in Modula, would be a notable improvement.

The "interface module" of Modula is essentially a monitor, as developed by Brinch Hansen and Hoare (4). A monitor encompasses a critical resource and the procedures through which the resource can be accessed. Processes can only use the critical resource by invoking the appropriate procedure. The monitor guarantees mutual exclusion by allowing only one process to be active within it at one time. "Wait" and "signal" operations may be used by the monitor procedures to suspend and awaken processes executing within them. The monitor relieves concurrent processes of the duty of dealing directly with the synchronization primitives and removes the shared resource from direct, and thus possibly incorrect, manipulation by the processes. Each process needs only to call a procedure for the operation it requires. Thus, the risk of error is reduced.

A monitor feature could be added to CONCUR using its existing synchronization primitives -- semaphores and the SET, PWAIT, and VSIGNAL commands. First, to insure that the procedures within the monitor actually acted as procedures rather than processes, CONCUR would be required to generate a semaphore which the calling process would set to zero before the call and wait on immediately after the call. Each procedure within the module would need to end by signalling its caller via that semaphore. Second, to guarantee that only one process had access to the monitor at one time, CONCUR would need to associate a mutual exclusion semaphore with the monitor. The semaphore would be initialized to one. Then each procedure within the monitor would begin by performing a P operation on the mutual exclusion semaphore and end by performing a V operation on it.

Finally, CONCUR would be required to translate the monitor "wait" and "signal" operations into corresponding CONCUR operations. Monitor waits and signals do not translate directly into semaphore waits and signals. The former involve a "condition variable" which has no memory associated with it. That is, if a signal is given for a condition and no process is waiting for the signal, the signal will be lost; in contrast, a semaphore's value holds a record of every operation performed on the semaphore. Also, when a process waiting on

a semaphore is signalled, both it and the signalling process continue. Since only one process may use the monitor at one time, the process signalling the condition must pause until the process it awakened exits the monitor or is suspended again.

To insure that only the awakened process can continue and to give the signalling process higher priority than other processes waiting to enter the monitor, a second semaphore must be used to handle signalling processes. Hoare calls this semaphore "urgent" (14,p.551). It is initialized to zero. Associated with the semaphore is an "urgentcount" that shows how many signalling processes are currently suspended. To honor the higher priority of signalling processes, the exit from each procedure can no longer be a simple P operation on the monitor's mutual exclusion semaphore. Instead, the urgentcount must be tested. If a signalling process is waiting, the procedure exits with a V on the urgent semaphore to let the signalling process continue; otherwise it exits with the V on the mutual exclusion semaphore, and other processes may enter the monitor.

Condition variables and monitor waits and signals are further implemented by introducing a semaphore and a counter for each condition variable. Both are initialized to zero. The counter indicates the number of processes waiting on a condition. The semaphore actually suspends and awakens a

process for that condition. A wait operation can be stated in CONCUR as

```
COUNTER := COUNTER + 1;
IF URGENTCOUNT > 0 THEN
     VSIGNAL(URGENT)
ELSE
     VSIGNAL(MUTEX)
END;
PWAIT(CONDSEM);
COUNTER := COUNTER - 1;
```

The signal operation can be written as

```
URGENTCOUNT := URGENTCOUNT + 1;
IF COUNTER > 0 THEN
     VSIGNAL(CONDSEM);
     PWAIT(URGENT)
END;
URGENTCOUNT := URGENTCOUNT - 1;
```

Thus, it appears that more sophisticated and reliable synchronization facilities can be built upon primitives already implemented in CONCUR. After this improvement is made, however, CONCUR would still need additional modification to bring it to production level. Some of these modifications are discussed in the areas of interest listed below.

The main purpose of this thesis project was to increase understanding of language design, compiler writing, and selected concepts in operating systems and machine architecture as well as to explore the implications of concurrent programming methodology. Further areas of investigation are prompted by this thesis.

1) Additional language features. Many possible extensions become quickly evident when CONCUR is compared with another high-level language such as PASCAL. For example, there is an obvious need for more data types and data structures. Also, more control structures could be added. The CASE control structure would eliminate the obscure use of the LOOP construct for case testing. The ability to pass arrays, and perhaps even semaphores, as parameters and to pass specified parameters by reference rather than by value could prove helpful. The ability to declare a procedure instead of a process would free the programmer from responsibility for introducing and manipulating a semaphore for each procedure. Finally, more sophisticated I/O functions are clearly needed. (The inability to print headings or at least force a page break frustrated any attempt to print the unsorted list as well as the sorted one produced by the quicksort program in Appendix B.)

2) Improved compiler characteristics. For example, the inclusion of options to control listing, summaries, and execution would be

convenient. These options could encompass such things as suppressing the listing, forcing page breaks in the listing, permitting starts and stops for tracing execution, and allowing redirection of input data and output so that they were not tied to the source input or list output devices specified. The compiler's error handling could also be improved. Error checking could be introduced to decrease the potential for semaphore abuse. Better diagnostic messages could be provided, and more clever recovery techniques could be employed. Finally, some code optimization would be welcome. Unnecessary use of temporary variables should be one of the first inefficiencies addressed.

3) More sophisticated operating systems. For example, the round robin scheduling algorithm could be replaced by an algorithm that is more sensitive to the actual mix of ready processes. For manipulating wait queues, other techniques in addition to the FIFO approach could be explored. Other methods of storage allocation could be used.

4) More complex machine architecture. Although the hypothetical machine has primitives that support CONCUR's parallel processing features very well, it lacks comparable support for more common language features. A stack-oriented architecture could well be more effective than the single accumulator, single register organization selected.

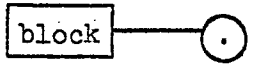
5) More rigorous programming methodologies. Specifically, the quicksort program indicates a need to consider the implications of recursive programming within a concurrent environment. In addition, the work of Susan Owicki (15), David Gries (15,16), and others investigating correctness proofs for parallel programs deserves attention.

With the focus of hardware and software development turning to parallelism to increase the performance of computer systems in spite of physical limitations, these topics and others should prove to be of great interest.

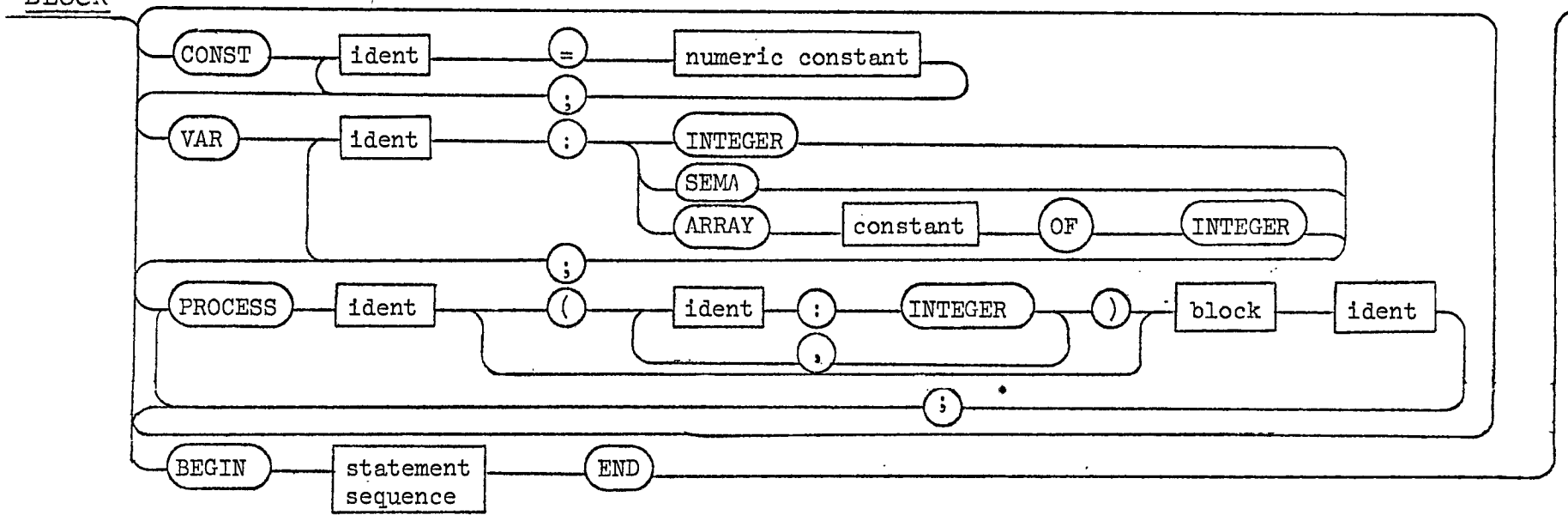
APPENDIX A
CONCUR SYNTAX GRAPHS

Syntax graphs are used by Niklaus Wirth to represent the syntax of a programming language. In his book, Algorithms + Data Structures = Programs, he provides a thorough explanation of the rules for constructing a graph as well as the rules for translating a graph into a program (17).

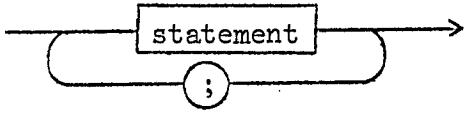
PROGRAM

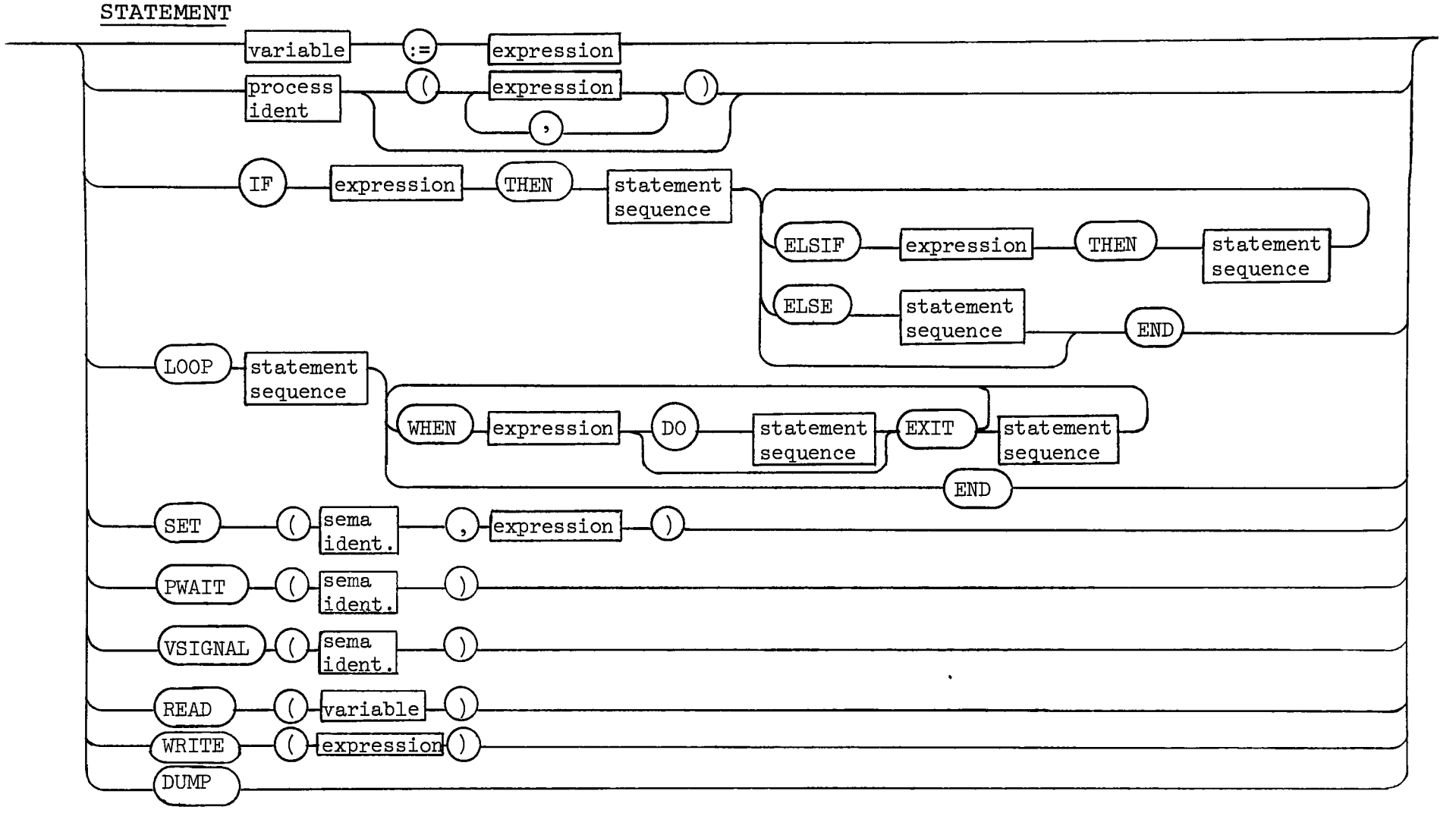


BLOCK

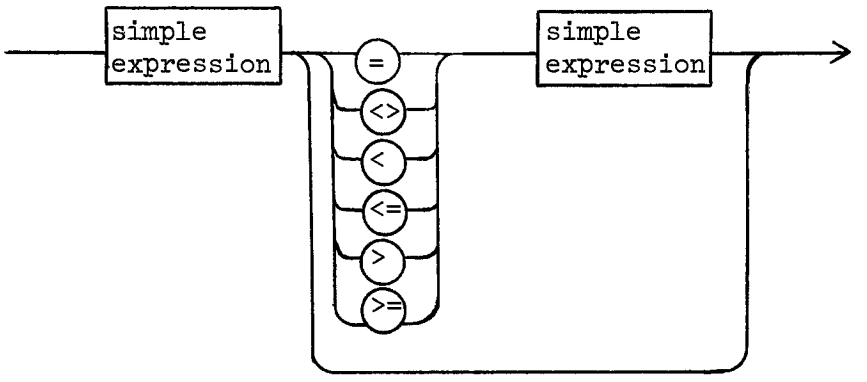


STATEMENT SEQUENCE

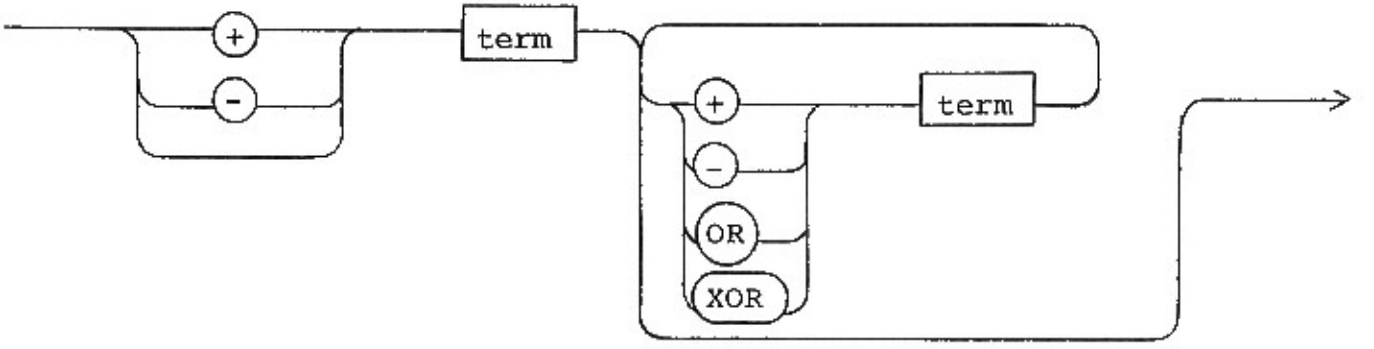



STATEMENT
variable
:=
expression
process ident
(
expression
,
)
IF
expression
THEN
statement sequence
ELSIF
expression
THEN
statement sequence
ELSE
statement sequence
END
LOOP
statement sequence
WHEN
expression
DO
statement sequence
EXIT
statement sequence
END
SET
(
sema ident.
,
expression
)
PWAIT
(
sema ident.
)
VSIGNAL
(
sema ident.
)
READ
(
variable
)
WRITE
(
expression
)
DUMP

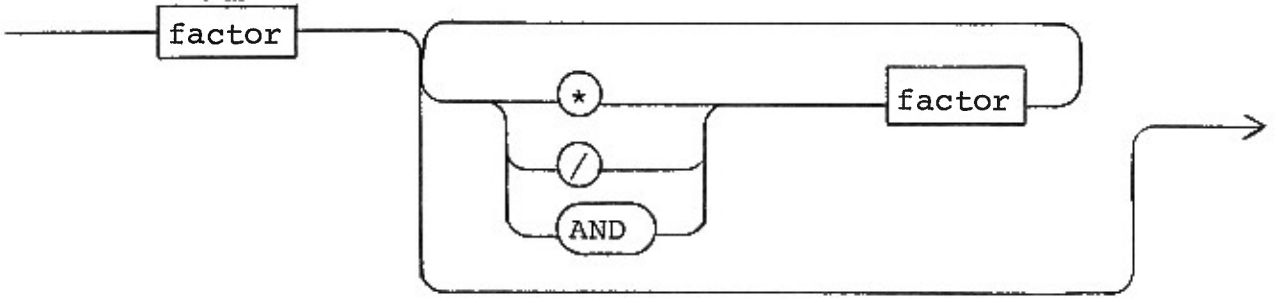
EXPRESSION



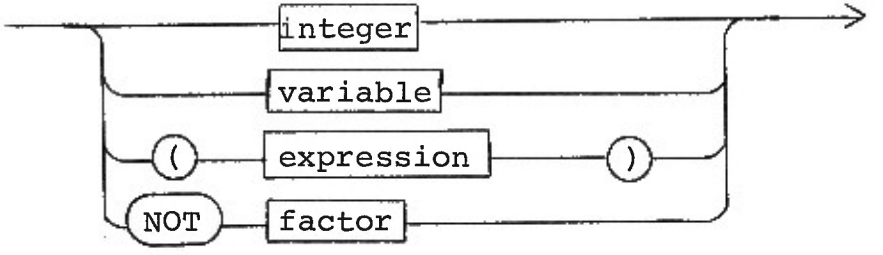
SIMPLE EXPRESSION



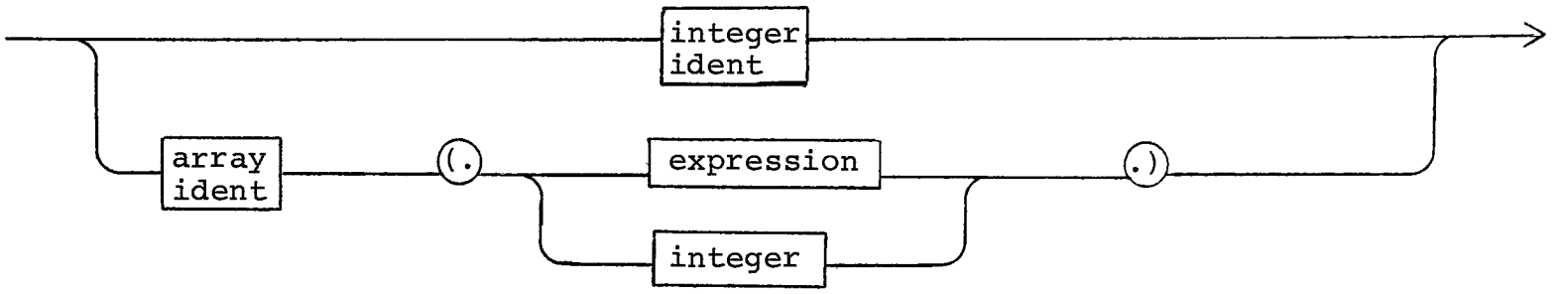
TERM



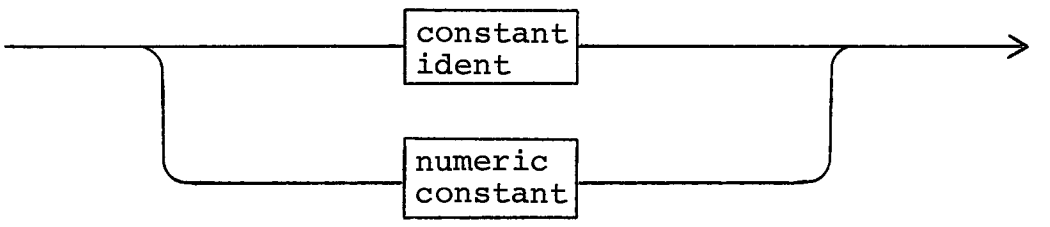
FACTOR



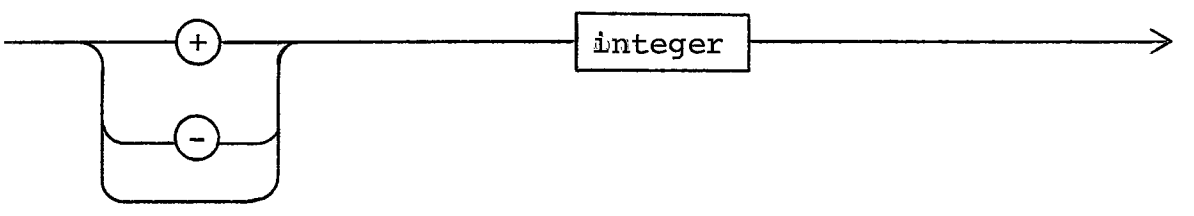
VARIABLE



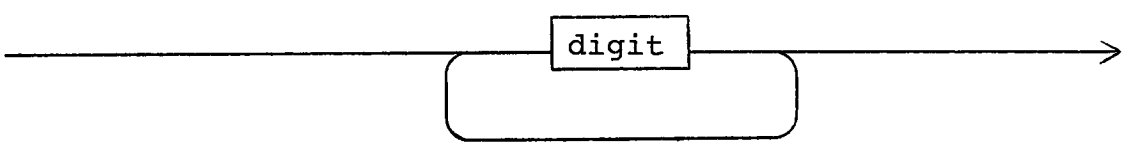
CONSTANT



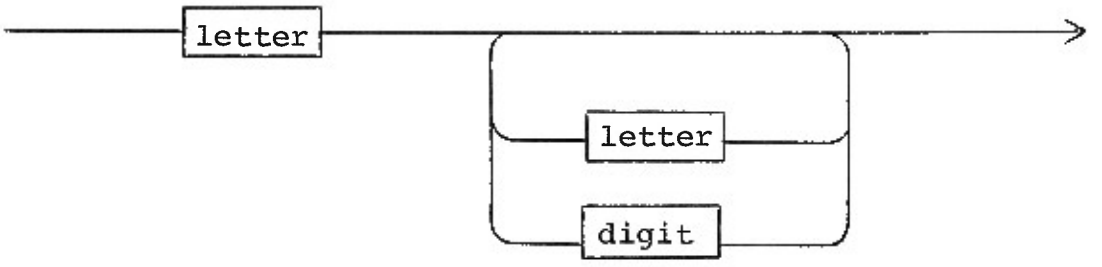
NUMERIC CONSTANT



INTEGER



IDENT



APPENDIX B

SAMPLE PROGRAMS

PRODUCER/CONSUMER PROGRAM

A SEQUENTIAL PROGRAM

CONCURRENT QUICKSORT PROGRAM

PRODUCER/CONSUMER PROGRAM

```
!COPY EXPROG1,460GMYUL
(* SIMPLE PRODUCER-CONSUMER PROBLEM...PAGE 37 TSICHRITZIS AND BERNSTEIN *)

CONST NOBUFFERS = 20; ENDSIGNAL = 9999; EMPTY = 99;
VAR   BUFFER: ARRAY NOBUFFERS OF INTEGER;
      BUFFERPTR: ARRAY NOBUFFERS OF INTEGER;
      INDEX : INTEGER;  QUEUEBEGIN: INTEGER;  QUEUEND: INTEGER;
      AVAIL: SEMA;  FULL: SEMA;   MUTEX: SEMA;

PROCESS PRODUCER;
  VAR INPUTDATA: INTEGER;
BEGIN
  LOOP
     READ (INPUTDATA);
     PWAIT(AVAIL);
     PWAIT(MUTEX);
     INDEX := 0;
     LOOP
        INDEX := INDEX + 1
        WHEN BUFFERPTR(.INDEX.) = 0 DO
           IF QUEUEBEGIN = EMPTY THEN
              QUEUEBEGIN := INDEX;
              QUEUEND := INDEX
           END;
           BUFFERPTR(.QUEUEND.) := INDEX;
           BUFFER(.INDEX.) := INPUTDATA;
           BUFFERPTR(.INDEX.) := EMPTY;
           QUEUEND := INDEX
        EXIT
     END;
     VSIGNAL(MUTEX);
     VSIGNAL(FULL)
     WHEN INPUTDATA = ENDSIGNAL EXIT
  END
END PRODUCER;

PROCESS CONSUMER;
  VAR OUTPUTDATA: INTEGER;
BEGIN
  LOOP
     PWAIT(FULL);
     PWAIT(MUTEX);
     INDEX := QUEUEBEGIN;
     OUTPUTDATA := BUFFER(.INDEX.);
     QUEUEBEGIN := BUFFERPTR(.INDEX.);
     BUFFERPTR(.INDEX.) := 0;
     VSIGNAL(MUTEX);
     VSIGNAL(AVAIL);
     WHEN OUTPUTDATA = ENDSIGNAL EXIT
     WRITE (OUTPUTDATA)
  END
END CONSUMER;
```

```
BEGIN (* MAIN PROGRAM *)
  SET (AVAIL,NOBUFFERS);
  SET (FULL,0);
  SET (MUTEX,1);
  INDEX := 0;
  QUEUEBEGIN := EMPTY;
  LOOP
     INDEX := INDEX + 1;
     BUFFERPTR(.INDEX.) := 0
     WHEN INDEX = NOBUFFERS EXIT
  END;
  PRODUCER;
  CONSUMER
END.
1900
1905
1919
1922
1923
1924
1946
1947
1949
1950
1953
1955
1958
1959
1960
1961
1965
1969
1970
1973
1974
1975
1976
1977
1978
1984
2001
2976
3500
4692
9999

!
```

```
08:16 04/28/78 607JN39M 16-42    L4] SYSTEM UP TILL 5:00PM....$$$ A C E $$$

!SET M:SI/EXPROG1.480GMYUL

!SET M:LO ME

!SET F:00 NO

!SET F:01 NO

!SET F:02 NO

!CONCUR.
 *PASCAL: START COMPILER
```

```
CONCUR COMPILER VERSION 1 JANUARY,1978                    08:16 APR 28,'78
OPTIONS: LS...
   1    !    (* SIMPLE PRODUCER-CONSUMER PROBLEM...PAGE 37 TSICHRITZIS AND BERNSTEIN *)
   2    !
   3    !    CONST NOBUFFERS = 20; ENDSIGNAL = 9999; EMPTY = 99;
   4    !    VAR   BUFFER: ARRAY NOBUFFERS OF INTEGER;
   5    !          BUFFERPTR: ARRAY NOBUFFERS OF INTEGER;
   6    !          INDEX : INTEGER;  QUEUEBEGIN: INTEGER;  QUEUEND: INTEGER;
   7    !          AVAIL: SEMA;  FULL: SEMA;   MUTEX: SEMA;
   8    !
   9    !    PROCESS PRODUCER;
  10    !      VAR INPUTDATA: INTEGER;
  11    !    BEGIN
  12    !      LOOP
  13    !         READ (INPUTDATA);
  14    !         PWAIT(AVAIL);
  15    !         PWAIT(MUTEX);
  16    !         INDEX := 0;
  17    !         LOOP
  18    !            INDEX := INDEX + 1
  19    !            WHEN BUFFERPTR(.INDEX.) = 0 DO
  20    !               IF QUEUEBEGIN = EMPTY THEN
  21    !                  QUEUEBEGIN := INDEX;
  22    !                  QUEUEND := INDEX
  23    !               END;
  24    !               BUFFERPTR(.QUEUEND.) := INDEX;
  25    !               BUFFER(.INDEX.) := INPUTDATA;
  26    !               BUFFERPTR(.INDEX.) := EMPTY;
  27    !               QUEUEND := INDEX
  28    !            EXIT
  29    !         END;
  30    !         VSIGNAL(MUTEX);
  31    !         VSIGNAL(FULL)
  32    !         WHEN INPUTDATA = ENDSIGNAL EXIT
  33    !      END
  34    !    END PRODUCER;
  35    !
  36    !    PROCESS CONSUMER;
  37    !      VAR OUTPUTDATA: INTEGER;
  38    !    BEGIN
  39    !      LOOP
  40    !         PWAIT(FULL);
  41    !         PWAIT(MUTEX);
  42    !         INDEX := QUEUEBEGIN;
  43    !         OUTPUTDATA := BUFFER(.INDEX.);
  44    !         QUEUEBEGIN := BUFFERPTR(.INDEX.);
  45    !         BUFFERPTR(.INDEX.) := 0;
  46    !         VSIGNAL(MUTEX);
  47    !         VSIGNAL(AVAIL);
  48    !         WHEN OUTPUTDATA = ENDSIGNAL EXIT
  49    !         WRITE (OUTPUTDATA)
  50    !      END
  51    !    END CONSUMER;
```

```
 53     !
 54     !     BEGIN (* MAIN PROGRAM *)
 55     !       SET (AVAIL,NOBUFFERS);
 56     !       SET (FULL,0);
 57     !       SET (MUTEX,1);
 58     !       INDEX := 0;
 59     !       QUEUEBEGIN := EMPTY;
 60     !       LOOP
 61     !          INDEX := INDEX + 1;
 62     !          BUFFERPTR(.INDEX.) := 0
 63     !          WHEN INDEX = NOBUFFERS EXIT
 64     !       END;
 65     !       PRODUCER;
 66     !       CONSUMER
 67     !     END.
0 ERRORS FOUND.
```

```
      1900
      1905
      1919
      1922
      1923
      1924
      1946
      1947
      1949
      1950
      1953
      1955
      1958
      1959
      1960
      1961
      1965
      1969
      1970
      1973
      1974
      1975
      1976
      1977
      1978
      1984
      2001
      2976
      3500
      4692
NORMAL TERMINATION.
END OF EXECUTION.
 *PASCAL: NORMAL END-RUN
```

A SEQUENTIAL PROGRAM

```
 COPY FXPROG2.480GMYUL
(*SAMPLE CONCUR PROGRAM USING SEMAPHORE TO MAKE PROCESS LIKE PROCEDURE.*)

CONST ASIZE = 10;  ENDSIG = 9999;

VAR INDATA: INTEGER;
    ARRAYDATA: ARRAY ASIZE OF INTEGER;
    START: INTEGER;  STOP: INTEGER;
    DONE: SEMA;

PROCESS INVERT(START: INTEGER, STOP: INTEGER);
  VAR TEMP: INTEGER;
BEGIN
  LOOP
     WHEN STOP <= START EXIT
     TEMP := ARRAYDATA(.START.);
     ARRAYDATA(.START.) := ARRAYDATA(.STOP.);
     ARRAYDATA(.STOP.) := TEMP;
     START := START + 1;
     STOP := STOP - 1
  END;
  VSIGNAL(DONE)
END INVERT;

BEGIN
  SET(DONE,0);
  START := 1;
  STOP := 0;
  LOOP
     READ(INDATA);
     WHEN INDATA = ENDSIG EXIT
     STOP := STOP + 1;
     ARRAYDATA(.STOP.) := INDATA
  END;
  INVERT(START,STOP);
  PWAIT(DONE);
  LOOP
     WHEN START > STOP EXIT
     WRITE(ARRAYDATA(.START.));
     START := START + 1
  END
END.
1
2
3
4
5
6
8
9999

!
```

```
CONCUR COMPILER VERSION 1 JANUARY-1978                    10:50 MAY 26,'78
OPTIONS: LS...
    1     !    (*SAMPLE CONCUR PROGRAM USING SEMAPHORE TO MAKE PROCESS LIKE PROCEDURE,*)
    2     !
    3     !    CONST ASIZE = 10;  ENDSIG = 9999;
    4     !
    5     !    VAR INDATA: INTEGER;
    6     !        ARRAYDATA: ARRAY /SIZE OF INTEGER;
    7     !        START: INTEGER;  STOP: INTEGER;
    8     !        DONE: SEMA;
    9     !
   10     !    PROCESS INVERT(START: INTEGER, STOP: INTEGER);
   11     !      VAR TEMP: INTEGER;
   12     !    BEGIN
   13     !      LOOP
   14     !          WHEN STOP <= START EXIT
   15     !          TEMP := ARRAYDATA(.START.);
   16     !          ARRAYDATA(.START.) := ARRAYDATA(.STOP.);
   17     !          ARRAYDATA(.STOP.) := TEMP;
   18     !          START := START + 1;
   19     !          STOP := STOP - 1
   20     !      END;
   21     !      VSIGNAL(DONE)
   22     !    END INVERT;
   23     !
   24     !    BEGIN
   25     !      SET(DONE,0);
   26     !      START := 1;
   27     !      STOP := 0;
   28     !      LOOP
   29     !          READ(INDATA);
   30     !          WHEN INDATA = ENDSIG EXIT
   31     !          STOP := STOP + 1;
   32     !          ARRAYDATA(.STOP.) := INDATA
   33     !      END;
   34     !      INVERT(START,STOP);
   35     !      PWAIT(DONE);
   36     !      LOOP
   37     !          WHEN START > STOP EXIT
   38     !          WRITE(ARRAYDATA(.START.));
   39     !          START := START + 1
   40     !      END
   41     !    END.
0 ERRORS FOUND.
```

```
          8
          7
          6
          5
          4
          3
          2
          1
 NORMAL TERMINATION.
 END OF EXECUTION.
  PPASCAL: NORMAL END-RUN
```

CONCURRENT QUICKSORT PROGRAM

```
!COPY EXPROG3.480GMYUL
(* A CONCURRENT QUICKSORT...READS IN A LIST OF NUMBERS, SORTS THEM USING
   A MODIFICATION OF THE QUICKSORT ALGORITHMS PROPOSED BY HOARE, AND THEN
   WRITES OUT THE SORTED LIST. *)

CONST MAXSIZE = 100;  ENDSIG = 9999;

VAR A: ARRAY MAXSIZE OF INTEGER;      (* THE LIST TO BE SORTED *)
    ELEMENTCNT: INTEGER;              (* NUMBER OF ITEMS IN THE LIST *)
    INDEX: INTEGER;                   (* INDEX FOR ACCESSING LIST *)
    QUICKDONE: SEMA;                  (* SEMA TO DELAY PRINT UNTIL SORT ENDS *)
    QUICKSIGS: INTEGER;               (* # OF QUICKSORTS EXPECTED TO SIGNAL *)
    CHECKSIGS: SEMA;                  (* SEMA FOR MUT. EXCL. FOR QUICKSIGS *)


PROCESS QUICKSORT (M: INTEGER, N: INTEGER);

(* FIRST LEVEL PROCESS
   ACTUALLY APPLIES QUICKSORT ALGORITHM TO ELEMENTS M THRU N OF THE LIST.*)

   VAR I: INTEGER;                    (* THE PIVOT FOR THE PARTITION *)
       PARTDONE: SEMA;                (* SEMA TO DELAY UNTIL PARTITION ENDS *)

   PROCESS PARTITION (M: INTEGER, N: INTEGER);

   (* SECOND LEVEL PROCESS
      PARTITIONS THE INTERVAL M - N IN RELATION TO A PIVOT I SO THAT
                  FOR J < I  A(.J.) <= A(.I.)
              AND FOR J > I  A(.J.) >= A(.I.). *)

      VAR X: INTEGER;                 (* HOLD AREA FOR VALUE OF A(.M.) *)
          EXCHDONE: SEMA;             (* SEMA TO DELAY UNTIL EXCHANGE ENDS *)

      PROCESS EXCHANGE (M: INTEGER, N:INTEGER);

      (* THIRD LEVEL PROCESS
         EXCHANGES THE MTH AND NTH ELEMENTS OF THE ARRAY. *)

         VAR TEMP: INTEGER;

      BEGIN (*EXCHANGE*)
         TEMP := A(.M.);
         A(.M.) := A(.N.);
         A(.N.) := TEMP;
         VSIGNAL(EXCHDONE)
      END EXCHANGE;

   BEGIN (*PARTITION*)
      SET (EXCHDONE,0);
      X := A(.M.);
      I := M;
      LOOP
         LOOP
            I := I + 1
```

```
            WHEN A(.I.) >= X EXIT
         END;
         LOOP
            N := N - 1
            WHEN A(.N.) <= X EXIT
         END
         WHEN I >= N EXIT;
         EXCHANGE(I,N);
         PWAIT(EXCHDONE)
      END;
      EXCHANGE(M,N);
      PWAIT(EXCHDONE);
      I := N;
      VSIGNAL(PARTDONE)
   END PARTITION;

BEGIN (*QUICKSORT*)
   SET (PARTDONE,0);
   IF N > M THEN
      PARTITION(M,N+1);
      PWAIT(PARTDONE);
      IF M < I THEN
         PWAIT(CHECKSIGS);
         QUICKSIGS := QUICKSIGS + 1;
         VSIGNAL(CHECKSIGS);
         QUICKSORT(M,I-1)
      END;
      IF I < N THEN
         PWAIT(CHECKSIGS);
         QUICKSIGS := QUICKSIGS + 1;
         VSIGNAL(CHECKSIGS);
         QUICKSORT(I+1,N)
      END
   END;
   PWAIT(CHECKSIGS);
   QUICKSIGS := QUICKSIGS - 1;
   IF QUICKSIGS <= 0 THEN VSIGNAL(QUICKDONE) END;
   VSIGNAL(CHECKSIGS)
END QUICKSORT;


BEGIN (*MAIN PROGRAM*)
   SET (CHECKSIGS,1);
   SET (QUICKDONE,0);
   ELEMENTCNT := 0;
   LOOP
      ELEMENTCNT := ELEMENTCNT + 1;
      READ (A(.ELEMENTCNT.))
      WHEN A(.ELEMENTCNT.) = ENDSIG EXIT
   END;
   ELEMENTCNT := ELEMENTCNT - 1;
   QUICKSIGS := 1;
   QUICKSORT(1,ELEMENTCNT);
   PWAIT(QUICKDONE);
   INDEX := 1;
```

```
   LOOP
      WRITE (A(.INDEX.))$
      INDEX := INDEX + 1
      WHEN INDEX > ELEMENTCNT EXIT
   END
END.
162
215
155
366
228
361
220
198
122
273
197
62
358
159
92
144
45
60
401
153
545
567
703
166
99
74
451
470
0
234
423
50
444
46
349
941
/
351
10
4
135
71
23
418
14
389
563
510
227
```

08:29 04/28/78 607JN39M 16-42 [20] SYSTEM UP TILL 5:00PM....$$$ A C E $$$

448
347
24
389
41
23
550
386
193
465
473
180
667
417
72
698
452
565
247
347
152
507
9999

!

  [21] SYSTEM UP TILL 5:00PM....$$$ A C E $$$

```
!SET M:SI/EXPROG3.480GMYUL

!SET M:LO ME

!SET F:00 NO

!SET F:01 NO

!SET F:02 NO

!CONCUR,
 *PASCAL: START COMPILER
```

```
CONCUR COMPILER VERSION 1 JANUARY,1978                    00:30 APR 28,'78
OPTIONS: LS...
   1    !    (* A CONCURRENT QUICKSORT...READS IN A LIST OF NUMBERS, SORTS THEM USING
   2    !       A MODIFICATION OF THE QUICKSORT ALGORITHMS PROPOSED BY HOARE, AND THEN
   3    !       WRITES OUT THE SORTED LIST. *)
   4    !
   5    !    CONST MAXSIZE = 100;  ENDSIG = 9999;
   6    !
   7    !    VAR A: ARRAY MAXSIZE OF INTEGER;          (* THE LIST TO BE SORTED *)
   8    !        ELEMENTCNT: INTEGER;                  (* NUMBER OF ITEMS IN THE LIST *)
   9    !        INDEX: INTEGER;                       (* INDEX FOR ACCESSING LIST *)
  10    !        QUICKDONE: SEMA;                      (* SEMA TO DELAY PRINT UNTIL SORT ENDS *)
  11    !        QUICKSIGS: INTEGER;                   (* # OF QUICKSORTS EXPECTED TO SIGNAL *)
  12    !        CHECKSIGS: SEMA;                      (* SEMA FOR MUT. EXCL. FOR QUICKSIGS *)
  13    !
  14    !
  15    !    PROCESS QUICKSORT (M: INTEGER, N: INTEGER);
  16    !
  17    !    (* FIRST LEVEL PROCESS
  18    !       ACTUALLY APPLIES QUICKSORT ALGORITHM TO ELEMENTS M THRU N OF THE LIST.*)
  19    !
  20    !       VAR I: INTEGER;                        (* THE PIVOT FOR THE PARTITION *)
  21    !           PARTDONE: SEMA;                    (* SEMA TO DELAY UNTIL PARTITION ENDS *)
  22    !
  23    !       PROCESS PARTITION (M: INTEGER, N: INTEGER);
  24    !
  25    !       (* SECOND LEVEL PROCESS
  26    !          PARTITIONS THE INTERVAL M - N IN RELATION TO A PIVOT I SO THAT
  27    !                       FOR J < I  A(.J.) <= A(.I.)
  28    !                 AND FOR J > I  A(.J.) >= A(.I.). *)
  29    !
  30    !          VAR X: INTEGER;                    (* HOLD AREA FOR VALUE OF A(.M.) *)
  31    !              EXCHDONE: SEMA;                (* SEMA TO DELAY UNTIL EXCHANGE ENDS *)
  32    !
  33    !          PROCESS EXCHANGE (M: INTEGER, N:INTEGER);
  34    !
  35    !          (* THIRD LEVEL PROCESS
  36    !             EXCHANGES THE MTH AND NTH ELEMENTS OF THE ARRAY. *)
  37    !
  38    !             VAR TEMP: INTEGER;
  39    !
  40    !          BEGIN (*EXCHANGE*)
  41    !             TEMP := A(.M.);
  42    !             A(.M.) := A(.N.);
  43    !             A(.N.) := TEMP;
  44    !             VSIGNAL(EXCHDONE)
  45    !          END EXCHANGE;
  46    !
  47    !       BEGIN (*PARTITION*)
  48    !          SET (EXCHDONE,0);
  49    !          X := A(.M.);
  50    !          I := M;
  51    !          LOOP
  52    !             LOOP
```

```
53    !                I := I + 1
54    !                WHEN A(.I.) >= X EXIT
55    !             END;
56    !             LOOP
57    !                N := N - 1
58    !                WHEN A(.N.) <= X EXIT
59    !             END
60    !             WHEN I > N EXIT;
61    !             EXCHANGE(I,N);
62    !             PWAIT(EXCHDONE)
63    !          END;
64    !          EXCHANGE(M,N);
65    !          PWAIT(EXCHDONE);
66    !          I := N;
67    !          VSIGNAL(PARTDONE)
68    !       END PARTITION;
69    !
70    !    BEGIN (*QUICKSORT*)
71    !       SET (PARTDONE,0);
72    !       IF N > M THEN
73    !          PARTITION(M,N+1);
74    !          PWAIT(PARTDONE);
75    !          IF M < I THEN
76    !             PWAIT(CHECKSIGS);
77    !             QUICKSIGS := QUICKSIGS + 1;
78    !             VSIGNAL(CHECKSIGS);
79    !             QUICKSORT(M,I-1)
80    !          END;
81    !          IF I < N THEN
82    !             PWAIT(CHECKSIGS);
83    !             QUICKSIGS := QUICKSIGS + 1;
84    !             VSIGNAL(CHECKSIGS);
85    !             QUICKSORT(I+1,N)
86    !          END
87    !       END;
88    !       PWAIT(CHECKSIGS);
89    !       QUICKSIGS := QUICKSIGS - 1;
90    !       IF QUICKSIGS <= 0 THEN VSIGNAL(QUICKDONE) END;
91    !       VSIGNAL(CHECKSIGS)
92    !    END QUICKSORT;
93    !
94    !
95    !    BEGIN (*MAIN PROGRAM*)
96    !       SET (CHECKSIGS,1);
97    !       SET (QUICKDONE,0);
98    !       ELEMENTCNT := 0;
99    !       LOOP
100   !          ELEMENTCNT := ELEMENTCNT + 1;
101   !          READ (A(.ELEMENTCNT.))
102   !          WHEN A(.ELEMENTCNT.) = ENDSIG EXIT
103   !       END;
104   !       ELEMENTCNT := ELEMENTCNT   1;
105   !       QUICKSIGS := 1;
106   !       QUICKSORT(1,ELEMENTCNT);
107   !       PWAIT(QUICKDONE);
```

```
108     !       INDEX := 1;
109     !       LOOP
110     !          WRITE (A(.INDEX.));
111     !          INDEX := INDEX + 1
112     !          WHEN INDEX > ELEMENTCNT EXIT
113     !       END
114     !    END.
0 ERRORS FOUND.
```

[25] SYSTEM UP TILL 5:00PM....$$$ A C E $$$

4
7
8
10
14
23
23
24
41
45
46
50
60
62
71
72
74
92
99
122
135
144
152
153
155
159
162
166
180
193
197
198
215
220
227
234
247
273
298
331
347
347
351
358
366
306
389
389
401
417
418
423
444
448

```
08:35 04/28/78 607JN39M 16-42    [26] SYSTEM UP TILL 5:00PM....$$$ A C E $$$


        451
        452
        465
        470
        473
        507
        510
        541
        545
        549
        550
        563
        565
        567
        667
        698
        703
NORMAL TERMINATION.
END OF EXECUTION.
 *PASCAL: NORMAL END-RUN

!
```

```
!COPY EXPROG4.480G''YUL
(* A CONCURRENT QUICKSORT...READS IN A LIST OF NUMBERS, SORTS THEM USING
   A MODIFICATION OF THE QUICKSORT ALGORITHMS PROPOSED BY HOARE, AND THEN
   WRITES OUT THE SORTED LIST. *)

CONST MAXSIZE = 100;  ENDSIG = 9999;

VAR A: ARRAY MAXSIZE OF INTEGER;        (* THE LIST TO BE SORTED *)
    ELEMENTCNT: INTEGER;                (* NUMBER OF ITEMS IN THE LIST *)
    INDEX: INTEGER;                     (* INDEX FOR ACCESSING LIST *)
    QUICKDONE: SEMA;                    (* SEMA TO DELAY PRINT UNTIL SORT ENDS *)
    QUICKSIGS: INTEGER;                 (* # OF QUICKSORTS EXPECTED TO SIGNAL *)
    CHECKSIGS: SEMA;                    (* SEMA FOR MUT. EXCL. FOR QUICKSIGS *)


PROCESS QUICKSORT (M: INTEGER, N: INTEGER);

(* FIRST LEVEL PROCESS
   ACTUALLY APPLIES QUICKSORT ALGORITHM TO ELEMENTS M THRU N OF THE LIST.*)

   VAR I: INTEGER;                      (* THE PIVOT FOR THE PARTITION *)
       PARTDONE: SEMA;                  (* SEMA TO DELAY UNTIL PARTITION ENDS *)

   PROCESS PARTITION (M: INTEGER, N: INTEGER);

   (* SECOND LEVEL PROCESS
      PARTITIONS THE INTERVAL M - N IN RELATION TO A PIVOT I SO THAT
                  FOR J < I  A(.J.) <= A(.I.)
              AND FOR J > I  A(.J.) >= A(.I.). *)

      VAR X: INTEGER;                   (* HOLD AREA FOR VALUE OF A(.M.) *)
          EXCHDONE: SEMA;               (* SEMA TO DELAY UNTIL EXCHANGE ENDS *)

      PROCESS EXCHANGE (M: INTEGER, N:INTEGER);

      (* THIRD LEVEL PROCESS
         EXCHANGES THE MTH AND NTH ELEMENTS OF THE ARRAY. *)

         VAR TEMP: INTEGER;

      BEGIN (*EXCHANGE*)
         TEMP := A(.M.);
         A(.M.) := A(.N.);
         A(.N.) := TEMP;
         VSIGNAL(EXCHDONE)
      END EXCHANGE;

   BEGIN (*PARTITION*)
      SET (EXCHDONE,0);
      X := A(.M.);
      I := M;
      LOOP
         LOOP
            I := I + 1
```

```
            WHEN A(.I.) >= X EXIT
         END;
         LOOP
            N := N - 1
            WHEN A(.N.) <= X EXIT
         END
         WHEN I > N EXIT;
         EXCHANGE(I,N);
         PWAIT(EXCHDONE)
      END;
      EXCHANGE(M,N);
      PWAIT(EXCHDONE);
      I := N;
      VSIGNAL(PARTDONE)
   END PARTITION;

BEGIN (*QUICKSORT*)
   SET (PARTDONE,0);
   IF N > M THEN
      PARTITION(M,N+1);
      PWAIT(PARTDONE);
      IF M < I THEN
         PWAIT(CHECKSIGS);
         QUICKSIGS := QUICKSIGS + 1;
         VSIGNAL(CHECKSIGS);
         QUICKSORT(M,I-1)
      END;
      IF I < N THEN
         PWAIT(CHECKSIGS);
         QUICKSIGS := QUICKSIGS + 1;
         VSIGNAL(CHECKSIGS);
         QUICKSORT(I+1,N)
      END
   END;
   PWAIT(CHECKSIGS);
   QUICKSIGS := QUICKSIGS - 1;
   IF QUICKSIGS <= 0 THEN VSIGNAL(QUICKDONE) END;
   VSIGNAL(CHECKSIGS)
END QUICKSORT;


BEGIN (*MAIN PROGRAM*)
   SET (CHECKSIGS,1);
   SET (QUICKDONE,0);
   ELEMENTCNT := 0;
   LOOP
      ELEMENTCNT := ELEMENTCNT + 1;
      READ (A(.ELEMENTCNT.))
      WHEN A(.ELEMENTCNT.) = ENDSIG EXIT
   END;
   ELEMENTCNT := ELEMENTCNT - 1;
   QUICKSIGS := 1;
   QUICKSORT(1,ELEMENTCNT);
   PWAIT(QUICKDONE);
```

```
   LOOP
      WRITE (A(.INDEX.));
      INDEX := INDEX + 1
      WHEN INDEX > ELEMENTCNT EXIT
   END
END.
```

```
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
9999

!
```

```
08:38 04/28/78 607JN39M 16-42     [31] SYSTEM UP TILL 5:00PM....$$$ A C E $$$


!SET M:SI/EXPROG4.480GMYUL

!SET M:LO ME

!SET F:00 NO

!SET F:01 NO

!SET F:02 NO

!CONCUR.
 *PASCAL: START COMPILER
```

```
CONCUR COMPILER VERSION 1 JANUARY,1978                    08:38 APR 28,'78
OPTIONS: LS...
   1    !    (* A CONCURRENT QUICKSORT...READS IN A LIST OF NUMBERS, SORTS THEM USING
   2    !       A MODIFICATION OF THE QUICKSORT ALGORITHMS PROPOSED BY HOARE, AND THEN
   3    !       WRITES OUT THE SORTED LIST. *)
   4    !
   5    !    CONST MAXSIZE = 100;  ENDSIG = 9999;
   6    !
   7    !    VAR A: ARRAY MAXSIZE OF INTEGER;        (* THE LIST TO BE SORTED *)
   8    !        ELEMENTCNT: INTEGER;                (* NUMBER OF ITEMS IN THE LIST *)
   9    !        INDEX: INTEGER;                     (* INDEX FOR ACCESSING LIST *)
  10    !        QUICKDONE: SEMA;                    (* SEMA TO DELAY PRINT UNTIL SORT ENDS *)
  11    !        QUICKSIGS: INTEGER;                 (* # OF QUICKSORTS EXPECTED TO SIGNAL *)
  12    !        CHECKSIGS: SEMA;                    (* SEMA FOR MUT. EXCL. FOR QUICKSIGS *)
  13    !
  14    !
  15    !    PROCESS QUICKSORT (M: INTEGER, N: INTEGER);
  16    !
  17    !    (* FIRST LEVEL PROCESS
  18    !       ACTUALLY APPLIES QUICKSORT ALGORITHM TO ELEMENTS M THRU N OF THE LIST.*)
  19    !
  20    !       VAR I: INTEGER;                      (* THE PIVOT FOR THE PARTITION *)
  21    !           PARTDONE: SEMA;                  (* SEMA TO DELAY UNTIL PARTITION ENDS *)
  22    !
  23    !       PROCESS PARTITION (M: INTEGER, N: INTEGER);
  24    !
  25    !       (* SECOND LEVEL PROCESS
  26    !          PARTITIONS THE INTERVAL M - N IN RELATION TO A PIVOT I SO THAT
  27    !                      FOR J < I  A(.J.) <= A(.I.)
  28    !                  AND FOR J > I  A(.J.) >= A(.I.). *)
  29    !
  30    !          VAR X: INTEGER;                   (* HOLD AREA FOR VALUE OF A(.M.) *)
  31    !              EXCHDONE: SEMA;               (* SEMA TO DELAY UNTIL EXCHANGE ENDS *)
  32    !
  33    !          PROCESS EXCHANGE (M: INTEGER, N:INTEGER);
  34    !
  35    !          (* THIRD LEVEL PROCESS
  36    !             EXCHANGES THE MTH AND NTH ELEMENTS OF THE ARRAY. *)
  37    !
  38    !             VAR TEMP: INTEGER;
  39    !
  40    !          BEGIN (*EXCHANGE*)
  41    !             TEMP := A(.M.);
  42    !             A(.M.) := A(.N.);
  43    !             A(.N.) := TEMP;
  44    !             VSIGNAL(EXCHDONE)
  45    !          END EXCHANGE;
  46    !
  47    !       BEGIN (*PARTITION*)
  48    !          SET (EXCHDONE,0);
  49    !          X := A(.M.);
  50    !          I := M;
  51    !          LOOP
```

```
53    !                  I := I + 1
54    !                  WHEN A(.I.) >= X EXIT
55    !               END;
56    !               LOOP
57    !                  N := N - 1
58    !                  WHEN A(.N.) <= X EXIT
59    !               END
60    !               WHEN I > N EXIT;
61    !               EXCHANGE(I,N);
62    !               PWAIT(EXCHDONE)
63    !            END;
64    !            EXCHANGE(M,N);
65    !            PWAIT(EXCHDONE);
66    !            I := N;
67    !            VSIGNAL(PARTDONE)
68    !         END PARTITION;
69    !
70    !      BEGIN (*QUICKSORT*)
71    !         SET (PARTDONE,0);
72    !         IF N > M THEN
73    !            PARTITION(M,N+1);
74    !            PWAIT(PARTDONE);
75    !            IF M < 1 THEN
76    !               PWAIT(CHECKSIGS);
77    !               QUICKSIGS := QUICKSIGS + 1;
78    !               VSIGNAL(CHECKSIGS);
79    !               QUICKSORT(M,I-1)
80    !            END;
81    !            IF I < N THEN
82    !               PWAIT(CHECKSIGS);
83    !               QUICKSIGS := QUICKSIGS + 1;
84    !               VSIGNAL(CHECKSIGS);
85    !               QUICKSORT(I+1,N)
86    !            END
87    !         END;
88    !         PWAIT(CHECKSIGS);
89    !         QUICKSIGS := QUICKSIGS - 1;
90    !         IF QUICKSIGS <= 0 THEN VSIGNAL(QUICKDONE) END;
91    !         VSIGNAL(CHECKSIGS)
92    !      END QUICKSORT;
93    !
94    !
95    !      BEGIN (*MAIN PROGRAM*)
96    !         SET (CHECKSIGS,1);
97    !         SET (QUICKDONE,0);
98    !         ELEMENTCNT := 0;
99    !         LOOP
100   !            ELEMENTCNT := ELEMENTCNT + 1;
101   !            READ (A(.ELEMENTCNT.))
102   !            WHEN A(.ELEMENTCNT.) = ENDSIG EXIT
103   !         END;
104   !         ELEMENTCNT := ELEMENTCNT - 1;
105   !         QUICKSIGS := 1;
106   !         QUICKSORT(1,ELEMENTCNT);
107   !         PWAIT(QUICKDONE);
```

```
108     !        INDEX := 1;
109     !        LOOP
110     !           WRITE (A(.INDEX.));
111     !           INDEX := INDEX + 1
112     !           WHEN INDEX > ELEMENTCNT EXIT
113     !        END
114     !     END.
0 ERRORS FOUND.
```

[35] SYSTEM UP TILL 5:00PM...$$$ A C E $$$

```
08:48 04/28/78 607JN39M 16-42     [36] SYSTEM UP TILL 5:00PM....$$$ A C E $$$


        55
        56
        57
        58
        59
        60
        61
        62
        63
        64
        65
        66
        67
        68
        69
        70
        71
        72
        73
        74
        75
        76
        77
        78
        79
        80
NORMAL TERMINATION.
END OF EXECUTION.
 *PASCAL: NORMAL END-RUN
```

APPENDIX C

SOURCE LISTING OF CONCUR COMPILER

```
 1  00004 (*************************************************************************)
 2  00004 (*  CONCUR COMPILER        WRITTEN OURING SUMMER QUARTER, 1977            *)
 3  00004 (*                         BY KAREN K. ANOERSON  ICSG-6                   *)
 4  00004 (*                                                                        *)
 5  00004 (*  CURRENT STATUS         AOOING INTERPRETER ANO COOE GENERATION         *)
 6  00004 (*                                                                        *)
 7  00004 (*  LAST MOOIFIEO          JANUARY 21ST, 1978                             *)
 8  00004 (*                                                                        *)
 9  00004 (*************************************************************************)
10  00004
11  00004 PROGRAM COMPILER(INPUT/IN,OUTPUT/OUT,CTRACE/OUT,OLIST/OUT,ETRACE/OUT);
12  00004   CONST PGMSTLIMIT = 500;              (* MAX. SIZE OF PROGRAM STORE *)
13  00004         EMPTY = 9999;                  (* EMPTY LIST SIGNAL *)
14  00004         LONGESTOP = 9;                 (* LENGTH OF LONGEST OP MNEMONIC *)
15  00004
16  00004   TYPE MNEMONIC = (AOOOP,SU8OP,RSU8OP,MULTOP,OIVOP,ROIVOP,EQLOP,NEQOP,LSSOP,
17  00004                    LEQOP,GTROP,GEQOP,ANOOP,OROP,XOROP,NOTOP,LOXOP,STXOP,LOAOP,
18  00004                    STAOP,LOIOP,STAXOP,PWAITOP,VSIGNALOP,8ROP,8RFOP,REAOOP,
19  00004                    WRITEOP,CREATEOP,TERMOP,OUMPOP);
20  00004
21  00004         INSTRUCTIONS = RECORO
22  00004                              CASE OPCOOE: MNEMONIC OF
23  00004
24  00004                           AOOOP,SU8OP,RSU8OP,
25  00004                           MULTOP,OIVOP,ROIVOP,
26  00004                           EQLOP,NEQOP,LSSOP,
27  00004                           LEQOP,GTROP,GEQOP,
28  00004                           ANOOP,OROP,XOROP,
29  00004                           LOXOP,STXOP,LOAOP,
30  00004                           STAOP,PWAITOP,VSIGNALOP:  (INOEXFLAG: 8OOLEAN;
31  00004                                                      LEVEL: INTEGER;
32  00004                                                      OISPLACEMENT: INTEGER);
33  00004
34  00004                           8ROP,8RFOP: (AOORESS: INTEGER);
35  00004                           LOIOP: (CONSTANT: INTEGER);
36  00004                           NOTOP,STAXOP,TERMOP,REAOOP,WRITEOP,OUMPOP: ();
37  00004                           CREATEOP: (INITIALPC: INTEGER;
38  00004                                      SLINK: INTEGER;
39  00004                                      OSIZE: INTEGER;
40  00004                                      PARAMCNT: INTEGER;
41  00004                                      PARAMSTRT: INTEGER)
42  00004                           ENO;
43  00004          ALPHAOPS = PACKEO ARRAY(.1..LONGESTOP.) OF CHAR;
44  00004   VAR PGMSTORE: ARRAY(.1..PGMSTLIMIT.) OF INSTRUCTIONS;   (*PROGRAM STORE*)
45  00C4E       PGMSTAOOR: INTEGER;              (* AOOR OF NEXT AVAILABLE LOC IN PGMSTORE *)
46  00C4F       MAINSTART: INTEGER;              (* START OF THE MAIN PROCESS *)
47  00C50       MAINOATASIZE: INTEGER;           (* SIZE OF MAIN PROCESS OATA AREA *)
48  00C51
49  00C51       CTRACE: TEXT;                    (* COMPILER TRACE OUTPUT FILE *)
50  00C9A       OLIST: TEXT;                     (* OBJECT LISTING OUTPUT FILE *)
51  00CE3       ETRACE: TEXT;                    (* EXECUTION TRACE OUTPUT FILE *)
52  0002C       ERRCNT: INTEGER;                 (* NUMBER OF ERRORS FOUND *)
53  00020       OPLIST: ARRAY(.MNEMONIC.) OF ALPHAOPS;  (* LIST OF MNEMONICS *)
54  0008A (*$FO*)
```

```
 55      00D8A (********
 56      00D8A  PROCEDURE TO COMPILE A CONCUR PROGRAM AND FILL PGMSTORE WITH OBJECT CODE.
 57      00D8A  ********)
 58      00D8A
 59      00D8A PROCEDURE COMPILE;
 60      00004
 61      00004 LABEL 9999;
 62      00004
 63      00004 CONST NDRESWDS = 27;                  (* NUMBER OF RESERVED WORDS *)
 64      00004       NOSINGSYM = 8;                  (* NO OF SPEC SYM NOT PART OF DOUBLE SYM *)
 65      00004       TABLELIMIT = 100;               (* MAX. LENGTH OF SYMBOL TABLE *)
 66      00004       DIGMAX = 10;                    (* MAX. NUMBER OF DIGITS IN INTEGERS *)
 67      00004       IDMAX = 12;                     (* MAX. LENGTH OF IDENTIFIERS *)
 68      00004       NULL = 0;                       (* INTERNAL CODE FOR NULL CHARACTERS *)
 69      00004       BLANK = 20;                     (* INTERNAL CODE FOR BLANK *)
 70      00004       LETTER = 21;                    (* INTERNAL CODE FOR LETTER *)
 71      00004  .    DIGIT = 22;                     (* INTERNAL CODE FOR DIGIT *)
 72      00004
 73      00004 TYPE  SYMBOLS =
 74      00004       (NUL,IDENT,NUMBER,EQL,NEQ,LSS,LEQ,GTR,GEQ,BECOMES,LPAREN,RPAREN,COMMA,
 75      00004        PLUS,MINUS,TIMES,SLASH,SEMICOLON,COLON,LBRACKET,RBRACKET,PERIOD,ANDSYM,
 76      00004        ARRAYSYM,BEGINSYM,CONSTSYM,DOSYM,DUMP,ELSESYM,ELSEIF,ENDSYM,EXIT,IFSYM,
 77      00004        INTSYM,LOOP,NOTSYM,DFSYM,ORSYM,PROCSYM,PWAIT,READSYM,SEMASYM,SETSYM,
 78      00004        THENSYM,VARSYM,VSIGNAL,WHEN,WRITESYM,XOR);
 79      00004
 80      00004       SYMBOLSET = ARRAY (.SYMBOLS.) OF BOOLEAN;
 81      00004       ALPHA = PACKED ARRAY (.1..IDMAX.) OF CHAR;
 82      00004       OBJECT = (CONSTANT,WHOLENO,SEMAPHORE,ANARRAY,PROCESS);
 83      00004       LINKORVAL = RECORD
 84      00004                     CASE DEFINED: BOOLEAN OF
 85      00004                       TRUE:  (VAL: INTEGER);
 86      00004                      FALSE:  (LINK: INTEGER);
 87      00004                     END;
 88      00004       STENTRY = RECORD  NAME: ALPHA;        (* IDENTIFIER NAME *)
 89      00004                         CASE KIND: OBJECT OF  (* TYPE OF IDENTIFIER *)
 90      00004
 91      00004                   CONSTANT:   (VALUE: INTEGER);
 92      00004
 93      00004                   WHOLENO,
 94      00004                   SEMAPHORE:  (WLEVEL: INTEGER;
 95      00004                                WDISP: INTEGER);
 96      00004
 97      00004                   ANARRAY:    (ALEVEL: INTEGER;
 98      00004                                FIRSTPLACE: INTEGER;
 99      00004                                LENGTH: INTEGER);
100      00004
101      00004                   PROCESS:    (PLEVEL: INTEGER;
102      00004                                STARTADDR: LINKORVAL;
103      00004                                DATASIZE: LINKORVAL;
104      00004                                PARAMCNT: INTEGER);
105      00004
106      00004                   END;
107      00004 (*$FO*)
```

```
108  00004 VAR    CH: CHAR;                          (* LAST CHARACTER READ *)
109  00005        CHCLASS: INTEGER;                  (* CLASS OF LAST CHARACTER READ *)
110  00006        SYMBOL: SYMBOLS;                   (* LAST SYMBOL FOUND *)
111  00007        LASTID: ALPHA;                     (* LAST IDENTIFIER FOUND *)
112  0000A        IO: ALPHA;                         (* LAST IDENTIFIER OR RESERVED WORD FOUND *)
113  0000D        NUM: INTEGER;                      (* LAST INTEGER FOUND *)
114  0000E        COUNT: INTEGER;                    (* CHARACTER COUNT *)
115  0000F        LNUMBER: INTEGER;                  (* LINE NUMBER PRINTED ON LISTING *)
116  00010        LLENGTH: INTEGER;                  (* LINE LENGTH *)
117  00011        TEMPMAX: INTEGER;                  (* IOMAX WHICH CAN BE CHANGED *)
118  00012
119  00012        LINE: ARRAY(.1..81.) OF CHAR;      (* OUTPUT LINE *)
120  00063        TEMPIO: ALPHA;                     (* ARRAY IN WHICH IO IS BUILD *)
121  00066        WORD: ARRAY(.1..NORESWDS.) OF ALPHA; (* TABLE OF RESERVED WORDS *)
122  000B7        WORDSYM: ARRAY(.1..NORESWDS.) OF SYMBOLS; (* CORRESPONDING SYMBOLS *)
123  000D2        SINGLESYM: ARRAY(.1..NOSINGSYM.) OF SYMBOLS; (* SYMBOLS FOR SINGSYM *)
124  000DA        TRANSTABLE: ARRAY(.CHAR.) OF INTEGER; (*TRANS EBCDIC TO INTERNAL *)
125  001DA        DECLBEGSYS: SYMBOLSET;             (* LEGAL BEGIN SYMBOLS FOR DECLARATIONS *)
126  00208        STATBEGSYS: SYMBOLSET;             (* LEGAL BEGIN SYMBOLS FOR STATEMENTS *)
127  0023C  .     FACBEGSYS: SYMBOLSET;              (* LEGAL BEGIN SYMBOLS FOR FACTORS *)
128  0026D        FOLLOWSYMBOLS: SYMBOLSET;          (* LEGAL SYMBOLS TO FOLLOW CURRENT SYM *)
129  0029E        SYMBOLTABLE: ARRAY(.0..TABLELIMIT.) OF STENTRY;
130  00690        TODAY: PACKED ARRAY (.1..16.) OF CHAR; (* TODAY'S DATE *)
131  00694        TINDEX: INTEGER;                   (* INDEX FOR SYMBOL TABLE *)
132  00695
133  00695
134  00695 (********
135  00695  PROCEDURE TO PRINT ERROR INDICATION.
136  00695  ********)
137  00695
138  00695 PROCEDURE ERROR(ERRNO: INTEGER);
139  00005 VAR I:INTEGER;
140  00006 BEGIN
141  00006   WRITELN(' ':14,' ':COUNT,'?',ERRNO:2);
142  0001B   ERRCNT := ERRCNT + 1
143  0001B END (*ERROR*);
144  0001E (*$FO*)
```

```
145     0001E (********
146     0001E  PROCEDURE TO GET NEXT SYMBOL FROM INPUT.
147     0001E  ********)
148     0001E
149     0001E PROCEDURE GETASYMBOL;
150     00004   VAR I,J,K: INTEGER;
151     00007       LASTCH: CHAR;
152     00008
153     00008 (********
154     00008  PROCEDURE TO GET NEXT CHARACTER FROM INPUT.
155     00008  ALSO TAKES CARE OF PRINTING LISTING.
156     00008  ********)
157     00008
158     00008 PROCEDURE GETACHAR;
159     00004 BEGIN
160     00005   IF COUNT = LLENGTH THEN
161     0000C *     BEGIN
162     0000C           IF EOF(INPUT) THEN
163     00010               BEGIN
164     00010                   WRITE (' PROGRAM INCOMPLETE');
165     00015                   GOTO 9999
166     00019               END;
167     00019           LLENGTH := 0;
168     00010           COUNT := 0;
169     00021           LNUMBER := LNUMBER + 1;
170     00028           WRITE (' ',LNUMBER:4,' ':4,'!',' ':4);
171     00041           WHILE NOT EOLN(INPUT) DO
172     00046               BEGIN
173     00046                   LLENGTH := LLENGTH + 1;
174     00040                   READ(CH);
175     00054                   WRITE(CH);
176     0005B                   LINE(.LLENGTH.) := CH
177     00061               END;
178     0006C           WRITELN;
179     0006E           LLENGTH := LLENGTH + 1;
180     00075         READLN;
181     00077           LINE(.LLENGTH.) := ' ';
182     00082       END;
183     00082   COUNT := COUNT + 1;
184     00089   CH := LINE(.COUNT.);
185     00099   CHCLASS := TRANSTABLE(.CH.)
186     0009F END (*GETACHAR*);
187     000AE (*$FO*)
```

```
188      000AE BEGIN (*GETASYMBOL*)
189      00005   WHILE CHCLASS = BLANK DO GETACHAR;
190      0000C   IF CHCLASS = LETTER THEN
191      00010      BEGIN (*IDENTIFIER OR RESERVED WORD*)
192      00010         K := 0;
193      00012         REPEAT
194      00012            IF K < IDMAX THEN
195      00015               BEGIN
196      00015                  K := K + 1;
197      00016                  TEMPID(.K.) := CH
198      0001A               END;
199      00022            GETACHAR
200      00022         UNTIL NOT ((CHCLASS = LETTER) OR (CHCLASS = DIGIT));
201      00033         IF K >= TEMPMAX THEN
202      00037            TEMPMAX := K
203      00037         ELSE
204      0003B            REPEAT
205      0003B               TEMPID(.TEMPMAX.) := ' ';
206      00044               TEMPMAX := TEMPMAX - 1
207      00046            UNTIL TEMPMAX = K;
208      0004D         ID := TEMPID;
209      00053         I := 1;
210      00055         J := NORESWDS;
211      00057         REPEAT
212      00057            K := (I + J) DIV 2;
213      0005C            IF ID <= WORD(.K.) THEN J := K - 1;
214      0006D            IF ID >= WORD(.K.) THEN I := K + 1
215      0007B         UNTIL I > J;
216      00081         IF I-1 > J THEN SYMBOL := WORDSYM(.K.) ELSE SYMBOL := IDENT
217      00092      END
218      00095   ELSE
219      00096      IF CHCLASS = DIGIT THEN
220      0009A         BEGIN (*INTEGER*)
221      0009A            K := 0;
222      0009C            NUM := 0;
223      0009F            SYMBOL := NUMBER;
224      000A2            REPEAT
225      000A2               NUM := 10*NUM + (ORD(CH) - ORD('0'));
226      000AC               K:= K + 1;
227      000AD               GETACHAR
228      000AD            UNTIL CHCLASS <> DIGIT;
229      000B3            IF K > DIGMAX THEN ERROR(30)
230      000B7         END
231      000B9   ELSE
232      000BA      IF CH = ':' THEN
233      000BE         BEGIN
234      000BE            GETACHAR;
235      000C0            IF CH = '=' THEN
236      000C4               BEGIN
237      000C4                  SYMBOL := BECOMES;
238      000C7                  GETACHAR
239      000C7               END
240      000C9            ELSE
241      000CA               SYMBOL := COLON
242      000CA         END
243      000CD   ELSE
244      000CE      IF CH = '<' THEN
245      000D2         BEGIN
246      000D2            GETACHAR;
247      000D4            IF CH = '=' THEN
248      000DB               BEGIN
249      000DB                  SYMBOL := LEQ;
250      000DB                  GETACHAR
251      000DB               END
252      000DD            ELSE
253      000DE               IF CH = '>' THEN
254      000E2                  BEGIN
255      000E2                     SYMBOL := NEQ;
256      000E5                     GETACHAR
257      000E5                  END
258      000E7               ELSE
259      000E8                  SYMBOL := LSS
260      000E8         END
261      000E8   ELSE
262      000EC      IF CH = '>' THEN
263      000F0         BEGIN
```

```
265   000F2            IF CH = '=' THEN
266   000F6               BEGIN
267   000F6                  SYMBOL := GEQ;
268   000F9                  GETACHAR
269   000F9               END
270   000F8            ELSE
271   000FC               SYMBOL := GTR
272   000FC         END
273   000FF      ELSE
274   00100         IF CH = '(' THEN
275   00104         BEGIN
276   00104            GETACHAR;
277   00106            IF CH = '*' THEN
278   0010A               BEGIN
279   0010A                  LASTCH := '(';
280   0010C                  WHILE NOT ((LASTCH = '*') AND (CH = ')')) DO
281   0011A                     BEGIN
282   0011A                        LASTCH := CH;
283   00120                        GETACHAR
284   00120 *                   END;
285   00123                  GETACHAR;
286   00125                  GETASYMBOL
287   00125               END
288   00127            ELSE
289   00128               IF CH = '.' THEN
290   0012C                  BEGIN
291   0012C                     SYMBOL := LBRACKET;
292   0012F                     GETACHAR
293   0012F                  END
294   00131               ELSE
295   00132                  SYMBOL := LPAREN
296   00132         END
297   00135      ELSE
298   00136         IF CH = '.' THEN
299   0013A         BEGIN
300   0013A            GETACHAR;
301   0013C            IF CH = ')' THEN
302   00140               BEGIN
303   00140                  SYMBOL := RBRACKET;
304   00143                  GETACHAR
305   00143               END
306   00145            ELSE
307   00146               SYMBOL := PERIOD
308   00146         END
309   00149      ELSE
310   0014A         IF CHCLASS = NULL THEN
311   0014D         BEGIN
312   0014D            SYMBOL := NUL;
313   00150            GETACHAR
314   00150         END
315   00152         ELSE
316   00153         BEGIN
317   00153            SYMBOL := SINGLESYM[.CHCLASS.];
318   00160            GETACHAR
319   00160         END
320   00162   END (*GETASYMBOL*);
321   00165   (*$FO*)
```

```
322  00165 (********
323  00165  PROCEDURE TO GENERATE A MACHINE INSTRUCTION.
324  00165  ********)
325  00165
326  00165 PROCEDURE GEN(OP: MNEMONIC;OPRAND1,OPRAND2,OPRAND3,OPRAND4,OPRAND5: INTEGER);
327  0000A
328  0000A BEGIN
329  00007   IF PGMSTADDR > PGMSTLIMIT THEN
330  0000A     BEGIN
331  0000A       WRITELN(' PROGRAM STORE OVERFLOW');
332  00012       ERRCNT := ERRCNT + 1;
333  00013       GOTO 9999
334  00016     END;
335  00016   WITH PGMSTORE(.PGMSTADDR.) DO
336  00020     BEGIN
337  00020       OPCODE := OP;
338  00026       CASE OPCODE OF
339  0002D
340  0002D   ADDOP,SUBOP,RSUBOP,
341  0002D   MULTOP,DIVOP,RDIVOP,
342  0002D   EQLOP,NEQOP,LSSOP,
343  0002D   LEQOP,GTROP,GEQOP,
344  0002D   ANDOP,OROP,XOROP,
345  0002D   LDXOP,STXOP,LDAOP,
346  0002D   STAOP,PWAITOP,VSIGNALOP: BEGIN
347  0002D                              IF OPRAND1 = 0 THEN
348  0002F                                INDEXFLAG := FALSE
349  00030                              ELSE
350  00033                                INDEXFLAG := TRUE;
351  00036                              LEVEL := OPRAND2;
352  00039                              DISPLACEMENT := OPRAND3
353  0003A                            END;
354  0003D
355  0003D   BROP,BRFOP:                ADDRESS := OPRAND1;
356  00041
357  00041   LDIOP:                     CONSTANT := OPRAND1;
358  00045
359  00045   NOTOP,STAXOP,TERMOP,READOP,WRITEOP,DUMPDP:   ;
360  00046
361  00046
362  00046   CREATEOP:                  BEGIN
363  00046                              INITIALPC := OPRAND1;
364  00049                              SLINK := OPRAND2;
365  0004C                              DSIZE := OPRAND3;
366  0004F                              PARAMCNT := OPRAND4;
367  00052                              PARAMSTRT := OPRAND5
368  00053                            END;
369  00056       END (*CASE*)
370  00075     END;
371  00075   PGMSTADDR := PGMSTADDR + 1
372  00075 END (*GEN*);
373  0007E (*$FO*)
```

```
374      0007E (********
375      0007E  PROCEDURE TO ASSIST IN CHECKING FOR AND RECOVERING FROM SYNTAX ERRORS.
376      0007E  ********)
377      0007E
378      0007E PROCEDURE TESTINEXTSYM8OLS,FOLLOWSYM8OLS: SYM8OLSET;ERRNO: INTEGER);
379      00069   VAR S: SYM8OLS;
380      0006A       TEMPSYMBOLS: SYM8OLSET;
381      00098 8EGIN
382      00017   IF NOT (NEXTSYM8OLS(.SYMBOL.)) THEN
383      00020      8EGIN
384      00020         ERROR(ERRNO);
385      00023         FOR S := NUL TO XOR OD
386      0002A            TEMPSYM8OLS(.S.) := NEXTSYM8OLS(.S.) OR FOLLOWSYM8OLS(.S.);
387      0003F         WHILE NOT (TEMPSYM8OL(.SYM8OL.)) OD GETASYM8OL
388      00048      END
389      00048 END (*TEST*);
390      0004C *(*$F0*)
```

```
391   0004C (********
392   0004C  PROCEDURE TO PARSE A BLOCK.
393   0004C  ********)
394   0004C
395   0004C PROCEDURE BLOCK(LEV,DISP,TINDEX: INTEGER; PASSSYMBOLS: SYMBOLSET;
396   00008                 VAR PSTART,DSIZE: INTEGER);
397   0003B   VAR S: SYMBOLS;
398   0003C       TEMPSYMBOLS: SYMBOLSET;
399   0006D       LPROCESS: INTEGER;              (* SYMBOL TABLE INDEX FOR LATEST PROCESS *)
400   0006E       LSTART: INTEGER;                (* START OF LATEST PROCESS IN PGMSTORE *)
401   0006F       LSIZE: INTEGER;                 (* SIZE OF DATA AREA FOR LATEST PROCESS *)
402   00070       TEMPHIGH: INTEGER;              (* HIGH WATER MARK OF BLOCK'S TEMPORARIES *)
403   00071
404   00071 (********
405   00071  FUNCTION TO FIND THE POSITION OF A SYMBOL IN THE SYMBOL TABLE.  POSITION
406   00071  IS SET TO ZERO IF THE SYMBOL IS NOT CURRENTLY DEFINED.
407   00071  ********)
408   00071
409   00071   FUNCTION POSITION (ID: ALPHA): INTEGER;
410   00008       VAR I: INTEGER;
411   00009   BEGIN
412   0000A       SYMBOLTABLE(.0.).NAME := ID;
413   00010       I := TINDEX;
414   00013       WHILE SYMBOLTABLE(.I.).NAME <> ID DO
415   00023           I := I - 1;
416   00025       POSITION := I
417   00025   END (*POSITION*);
418   00029 (*$FO*)
```

```
419  00029 (********
420  00029  PROCEDURE TO ENTER AN ITEM INTO THE SYMBOL TABLE.
421  00029  ********)
422  00029
423  00029    PROCEDURE ENTER(K: OBJECT;VAR DISP: INTEGER);
424  00006    BEGIN
425  00007       TINDEX := TINDEX + 1;
426  0000C       IF TINDEX > TABLELIMIT THEN
427  00010          BEGIN
428  00010             WRITE ('SYMBOL TABLE OVERFLOW');
429  00015             ERRCNT := ERRCNT + 1;
430  00016             GOTO 9999
431  0001A          END;
432  0001A       WITH SYMBOLTABLE(.TINDEX.) DO
433  00026          BEGIN
434  00026             NAME := LASTID;
435  0002D*            KIND := K;
436  00033             CASE KIND OF
437  0003A    CONSTANT:    VALUE := NUM;
438  00040
439  00040    WHOLEND:     BEGIN
440  00040                    WLEVEL := LEV;
441  00044                    WDISP := DISP;
442  00048                    DISP := DISP + 1
443  00048                END;
444  0004E
445  0004E    SEMAPHORE:   BEGIN
446  0004E                    WLEVEL := LEV;
447  00052                    WDISP := DISP;
448  00056                    DISP := DISP + 2
449  00059                END;
450  0005C
451  0005C    ANARRAY:     BEGIN
452  0005C                    ALEVEL := LEV;
453  00060                    FIRSTPLACE := DISP;
454  00064                    LENGTH := NUM;
455  00069                    DISP := DISP + LENGTH
456  0006D                END;
457  00070
458  00070    PROCESS:     BEGIN
459  00070                    PLEVEL := LEV;
460  00074                    STARTADDR.DEFINED := FALSE;
461  00077                    STARTADDR.LINK := EMPTY;
462  0007A                    DATASIZE.DEFINED := FALSE;
463  00070                    DATASIZE.LINK := EMPTY
464  0007E                END
465  00080             END
466  00086          END
467  00086    END (*ENTER*);
468  0008F (*$FO*)
```

```
00008F (********
00008F  PROCEDURE TO PARSE A NUMERIC CONSTANT.
00008F  ********)
00008F
00008F    PROCEDURE NCONSTANT;
00004     VAR M: BOOLEAN;
00005     BEGIN
00005        IF SYMBOL = MINUS THEN M := TRUE ELSE M := FALSE;
0000F        IF (SYMBOL = PLUS) OR (SYMBOL = MINUS) THEN GETASYMBOL;
00022        IF SYMBOL = NUMBER THEN GETASYMBOL ELSE ERROR(27);
0002F        IF M = TRUE THEN NUM := (- NUM);
00038     END (*NCONSTANT*);
00039
00039 (********
00039  PROCEDURE TO PARSE A CONSTANT.
00039  ********)
00039
00039     PROCEDURE ICONSTANT;
00004        VAR I:INTEGER;
00005   * BEGIN
00005        IF SYMBOL = IDENT THEN
0000A           BEGIN
0000A              I := POSITION(ID);
00013              IF I = 0 THEN
00015                 ERROR(11)
00016              ELSE
0001A                 IF SYMBOLTABLE(.I.).KIND <> CONSTANT THEN
00026                    ERROR(2)
00027                 ELSE
00028                    NUM := SYMBOLTABLE(.I.).VALUE;
00037              GETASYMBOL
00037           END
0003A        ELSE
0003B           NCONSTANT
0003B     END (*ICONSTANT*);
0003F
0003F (********
0003F  PROCEDURE TO PARSE A CONSTANT DECLARATION.
0003F  ********)
0003F
0003F     PROCEDURE CONSTDECLARATION;
00004     BEGIN
00005  WRITELN(CTRACE,' IN CONSTDECLARATION...');
00000        IF SYMBOL = IDENT THEN
00012           BEGIN
00012              LASTID := ID;
0001A              GETASYMBOL;
00010              IF (SYMBOL = EQL) OR (SYMBOL = BECOMES) THEN
00020                 BEGIN
00020                    IF SYMBOL = BECOMES THEN ERROR(1);
00036                    GETASYMBOL;
00039                    NCONSTANT;
0003B                    ENTER(CONSTANT,DISP);
00040                    IF SYMBOL = SEMICOLON THEN GETASYMBOL ELSE ERROR(5)
0004A                 END
00040              ELSE
0004E                 ERROR(3)
0004F           END
00052        ELSE
00053           ERROR(4)
00054     END (*CONSTDECLARATION*);
00061 (*$FO*)
```

```
531      00061 (********
532      00061  PROCEDURE TO PARSE A VARIABLE DECLARATION.
533      00061  ********)
534      00061   PROCEDURE VARDECLARATION;
535      00004   BEGIN
536      00005 WRITELN(CTRACE,' IN VARDECLARATION...');
537      00000       IF SYMBOL = IDENT THEN
538      00012          BEGIN
539      00012             LASTID := ID;
540      0001A             GETASYMBOL;
541      0001D             IF SYMBOL = COLON THEN
542      00022                BEGIN
543      00022                   GETASYMBOL;
544      00025                   IF SYMBOL = INTSYM THEN
545      0002A                      BEGIN
546      0002A                         ENTER(WHOLEND,DISP);
547      0002F                         GETASYMBOL;
548      00032                         IF SYMBOL = SEMICOLON THEN GETASYMBOL ELSE ERROR(5)
549      0003C *                    END
550      0003F                   ELSE
551      00040                      IF SYMBOL = SEMASYM THEN
552      00045                         BEGIN
553      00045                            ENTER(SEMAPHORE,DISP);
554      0004A                            GETASYMBOL;
555      0004D                            IF SYMBOL = SEMICOLON THEN GETASYMBOL ELSE ERROR(5)
556      00057                         END
557      0005A                      ELSE
558      0005B                         IF SYMBOL = ARRAYSYM THEN
559      00060                            BEGIN
560      00060                               GETASYMBOL;
561      00063                               ICONSTANT;
562      00065                               IF SYMBOL = OFSYM THEN
563      0006A                                  BEGIN
564      0006A                                     GETASYMBOL;
565      0006D                                     IF SYMBOL = INTSYM THEN
566      00072                                        BEGIN
567      00072                                           ENTER(ANARRAY,DISP);
568      00077                                           GETASYMBOL;
569      0007A                                           IF SYMBOL = SEMICOLON THEN
570      0007F                                              GETASYMBOL
571      0007F                                           ELSE
572      00083                                              ERROR(5)
573      00084                                        END
574      00087                                     ELSE
575      00088                                        ERROR(14)
576      00089                                  END
577      0008C                               ELSE
578      0008D                                  ERROR(15)
579      0008E                            END
580      00091                         ELSE
581      00092                            ERROR(31)
582      00093                END
583      00096             ELSE
584      00097                ERROR(32)
585      0009B          END
586      0009B       ELSE
587      0009C          ERROR(4)
588      00090   END (*VARDECLARATION*);
589      000AA (*$FO*)
```

```
590   000AA (********
591   000AA  PROCEDURE TO PARSE A PROCESS DECLARATION.
592   000AA  ********)
593   000AA
594   000AA    PROCEDURE PROCDECLARATION;
595   00004       VAR S: SYMBOLS;
596   00005           DISP: INTEGER;
597   00006           PARAMCNT: INTEGER;
598   00007
599   00007 (********
600   00007  PROCEDURE TO PARSE A FORMAL PARAMETER LIST.
601   00007  ********)
602   00007
603   00007       PROCEDURE FPARAMLIST;
604   00004
605   00004 (********
606   00004  PROCEDURE TO PARSE A FORMAL PARAMETER.
607   00004  ********)
608   00004
609   00004 .        PROCEDURE FPARAMETER;
610   00004          BEGIN
611   00005             IF SYMBOL = IDENT THEN
612   0000C                BEGIN
613   0000C                   LASTID := ID;
614   00018                   GETASYMBOL;
615   0001D                   IF SYMBOL = COLON THEN
616   00024                      BEGIN
617   00024                         GETASYMBOL;
618   00029                         IF SYMBOL = INTSYM THEN
619   00030                            BEGIN
620   00030                               ENTER(WHOLEND,DISP);
621   0003B                               GETASYMBOL
622   0003B                            END
623   0003D                         ELSE
624   0003E                            ERROR(7)
625   0003F                      END
626   00044                   ELSE
627   00045                      ERROR(32)
628   00046                END
629   0004B             ELSE
630   0004C                ERROR(4)
631   0004D          END (*FPARAMETER*);
632   00055
633   00055       BEGIN (*PARAMLIST*)
634   00005          FPARAMETER;
635   00007          PARAMCNT := 1;
636   0000A          WHILE SYMBOL = COMMA DO
637   00010             BEGIN
638   00010                GETASYMBOL;
639   00014                FPARAMETER;
640   00016                PARAMCNT := PARAMCNT + 1
641   00018             END;
642   0001C          SYMBOLTABLE(.LPROCESS.).PARAMCNT := PARAMCNT;
643   0002C          IF SYMBOL = RPAREN THEN GETASYMBOL ELSE ERROR(22)
644   00038       END (*FPARAMLIST*);
645   0003D (*$FO*)
```

```
646      0003D (********
647      0003D  PROCEDURE TO FIX UP INSTRUCTIONS NEEDING PROCESS START ADDRESS AND DATA SIZE.
648      0003D  ********)
649      0003D
650      0003D      PROCEDURE FIXUP(LPROCESS,LSTART,LSIZE: INTEGER);
651      00007      VAR SLINK,TLINK: INTEGER;
652      00009 BEGIN
653      00007      WITH SYMBOLTABLE(.LPROCESS.) DO
654      00013          BEGIN
655      00013              TLINK := STARTADDR.LINK;
656      00016              WHILE TLINK <> EMPTY DO
657      00019                  BEGIN
658      00019                      SLINK := TLINK;
659      00018                      TLINK := PGMSTORE(.SLINK.).INITIALPC;
660      00023                      PGMSTORE(.SLINK.).INITIALPC := LSTART
661      0002A                  END;
662      0002E              STARTADDR.DEFINED := TRUE;
663      00031              STARTADDR.VAL := LSTART;
664      00034              TLINK := DATASIZE.LINK;
665      00037              WHILE TLINK <> EMPTY DO
666      0003A                  BEGIN
667      0003A                      SLINK := TLINK;
668      0003C                      TLINK := PGMSTORE(.SLINK.).DSIZE;
669      00044                      PGMSTORE(.SLINK.).DSIZE := LSIZE
670      0004B                  END;
671      0004F              DATASIZE.DEFINED := TRUE;
672      00052              DATASIZE.VAL := LSIZE
673      00053          END
674      00055      END (*FIXUP*);
675      00056 (*$FO*)
```

```
676      00056 (********
677      00056  PROCEDURE TO PARSE A PROCEDURE DECLARATION.
678      00056  ********)
679      00056
680      00056    BEGIN (*PROCDECLARATION*)
681      00005 WRITELN(CTRACE,' IN PROCDECLARATION...');
682      0000D       IF SYMBOL = IDENT THEN
683      00012          BEGIN
684      00012             DISP := 0;
685      00014             LASTID := ID;
686      0001C             ENTER(PROCESS,DISP);
687      00020             LPROCESS := TINDEX;
688      00024             LEV := LEV + 1;
689      00029             PARAMCNT := 0;
690      0002B             GETASYMBOL;
691      0002E             IF SYMBOL = LPAREN THEN
692      00033                BEGIN
693      00033                   GETASYMBOL;
694      00036                   FPARAMLIST
695      00036                END;
696      00038             SYMBOLTABLE(.LPROCESS.).PARAMCNT := PARAMCNT
697      00040          END
698      00045       ELSE
699      00046          BEGIN
700      00046             LEV := LEV + 1;
701      0004B             ERROR(4)
702      0004C          END;
703      0004F       IF SYMBOL = SEMICOLON THEN GETASYMBOL ELSE ERROR(5);
704      0005C       FOR S := NUL TO XOR 00 TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
705      00074       TEMPSYMBOLS(.IDENT.) := TRUE;
706      00077       BLOCK(LEV,DISP,TINDEX,TEMPSYMBOLS,LSTART,LSIZE);
707      00087       IF SYMBOL = IDENT THEN
708      0008C          BEGIN
709      0008C             IF ID <> SYMBOLTABLE(.LPROCESS.).NAME THEN ERROR(6);
710      000A1             GETASYMBOL
711      000A1          END
712      000A4       ELSE
713      000A5          ERROR(4);
714      000A9       LEV := LEV - 1;
715      000AE       TINDEX := LPROCESS;
716      000B2       FIXUP(LPROCESS,LSTART,LSIZE);
717      000BA       IF SYMBOL = SEMICOLON THEN GETASYMBOL ELSE ERROR(5)
718      000C4    END (*PROCDECLARATION*);
719      000D4 (*$FO*)
```

```
720     00004 (********
721     00004  PROCEDURE TO PARSE A STATEMENT SEQUENCE.
722     00004  ********)
723     00004
724     00004   PROCEDURE SSEQUENCE(PASSSYMBOLS:   SYMBOLSET);
725     00036      VAR S: SYMBOLS;
726     00037          TEMPSYMBOLS: SYMBOLSET;
727     00068 (********
728     00068  PROCEDURE TO PARSE A STATEMENT.
729     00068  ********)
730     00068
731     00068   PROCEDURE STATEMENT(PASSSYMBOLS:   SYMBOLSET);
732     00036      VAR I: INTEGER;
733     00037          S: SYMBOLS;
734     00038          TEMPSYMBOLS: SYMBOLSET;
735     00069
736     00069 (********
737     00069  FUNCTION TO RETURN THE INVERSE OF A GIVEN ARITHMETIC OPERATION.
738     00069  ********)
739     00069
740     00069      FUNCTION INVERSEOP(ARITHOP: SYMBOLS): MNEMONIC;
741     00005
742     00005      BEGIN
743     00006         CASE ARITHOP OF
744     0000C      EQL:      INVERSEOP := EQLOP;
745     0000F      NEQ:      INVERSEOP := NEQOP;
746     00012      LSS:      INVERSEOP := GTROP;
747     00015      LEQ:      INVERSEOP := GEQOP;
748     00018      GTR:      INVERSEOP := LSSOP;
749     00018      GEQ:      INVERSEOP := LEQOP;
750     0001E      PLUS:     INVERSEOP := ADDOP;
751     00021      MINUS:    INVERSEOP := RSUBOP;
752     00024      TIMES:    INVERSEOP := MULTOP;
753     00027      SLASH:    INVERSEOP := RDIVOP;
754     0002A      ANDSYM:   INVERSEOP := ANDOP;
755     0002D      ORSYM:    INVERSEOP := OROP;
756     00030      XOR:      INVERSEOP := XOROP;
757     00033         END (*CASE*)
758     00061      END (*INVERSEOP*);
759     00062
760     00062 (********
761     00062  FUNCTION TO INCREMENT PTR TO NEXT TEMPORARY DATA LOCATION.
762     00062  ********)
763     00062
764     00062      FUNCTION NEXTEMP: INTEGER;
765     00004
766     00004      BEGIN
767     00005         NEXTEMP := DISP;
768     0000A         DISP := DISP + 1;
769     00013         IF DISP > TEMPHIGH THEN TEMPHIGH := DISP
770     0001C      END (*NEXTEMP*);
771     00025
772     00025 (********
773     00025  PROCEDURE TO RELEASE TEMPORARY LOCATIONS.
774     00025  ********)
775     00025
776     00025      PROCEDURE RELEASETEMP(RELCNT: INTEGER);
777     00005
778     00005      BEGIN
779     00006         DISP := DISP - RELCNT
780     0000A      END (*RELEASETEMP*);
781     00010 (*$FO*)
```

```
00010 (********
00010  PROCEDURE TO PARSE AN EXPRESSION.
00010  ********)
00010
00010      PROCEDURE EXPRESSION(PASSSYMBOLS:   SYMBOLSET);
00036         VAR S: SYMBOLS;
00037             TEMPSYMBOLS: SYMBOLSET;
00068             NEXTOP: MNEMONIC;
00069             TEMPORARY: INTEGER;
0006A
0006A (********
0006A  PROCEDURE TO PARSE A SIMPLE EXPRESSION.
0006A  ********)
0006A
0006A          PROCEDURE SIMPLEEXPRESSION(PASSSYMBOLS:   SYMBOLSET);
00036             VAR S: SYMBOLS;
00037                 TEMPSYMBOLS: SYMBOLSET;
00068                 NEXTOP: MNEMONIC;
00069                 TEMPORARY: INTEGER;
0006A
0006A (********
0006A  PROCEDURE TO PARSE A TERM.
0006A  ********)
0006A
0006A             PROCEDURE TERM(PASSSYMBOLS:   SYMBOLSET);
00036                VAR S: SYMBOLS;
00037                    TEMPSYMBOLS: SYMBOLSET;
00068                    NEXTOP: MNEMONIC;
00069                    TEMPORARY: INTEGER;
0006A (********
0006A  PROCEDURE TO PARSE A FACTOR.
0006A  ********)
0006A
0006A                PROCEDURE FACTOR(PASSSYMBOLS:   SYMBOLSET);
00036                   VAR I: INTEGER;
00037                       S: SYMBOLS;
00038                       TEMPSYMBOLS: SYMBOLSET;
00069                       M: BOOLEAN;
0006A (*$F0*)
```

```
821   0006A (********
822   0006A  PROCEDURE TO PARSE A VARIABLE.
823   0006A  ********)
824   0006A
825   0006A   PROCEDURE VARIABLE;
826   00004      VAR I: INTEGER;
827   00005          S: SYMBOLS;
828   00006   BEGIN
829   00005 WRITELN(CTRACE,' IN VARIABLE...');
830   0000D      I := POSITION(ID);
831   00022      IF I = 0 THEN
832   00024         ERROR(11)
833   00025      ELSE
834   0002F         WITH SYMBOLTABLE(.I.) DO
835   00040            IF KIND = ANARRAY THEN
836   00044               BEGIN
837   00044                  GETASYMBOL;
838   00040                  IF SYMBOL = LBRACKET THEN
839   00058                     BEGIN
840   00058                        GETASYMBOL;
841   00061                        IF SYMBOL = IDENT THEN
842   0006C                           BEGIN
843   0006C                              FOR S := NUL TO XOR DO
844   00073                                 TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
845   00084                              TEMPSYMBOLS(.RBRACKET.) := TRUE;
846   00087                              EXPRESSION(TEMPSYMBOLS)
847   00089                           END
848   0008F                        ELSE
849   00090                           IF SYMBOL = NUMBER THEN
850   0009B                              BEGIN
851   0009B                                 GEN(LDIOP,NUM,0,0,0,0);
852   000B4                                 GETASYMBOL
853   000B4                              END
854   000BD                           ELSE
855   000BE                              ERROR(26);
856   000C8                        IF SYMBOL = RBRACKET THEN GETASYMBOL ELSE ERROR(29);
857   000E7                        GEN(STAXOP,0,0,0,0,0);
858   000F8                        GEN(LDAOP,1,LEV-ALEVEL,FIRSTPLACE,0,0)
859   0010A                     END
860   00113                  ELSE
861   00114                     ERROR(28)
862   00115               END
863   0011E            ELSE
864   0011F               BEGIN
865   0011F                  IF KIND = WHOLEND THEN
866   00123                     GEN(LDAOP,0,LEV-WLEVEL,WDISP,0,0)
867   00135                  ELSE
868   0013F                     ERROR(10);
869   00149                  GETASYMBOL
870   00149               END
871   00152   END (*VARIABLE*);
872   0015A (*$FO*)
```

```
873  0015A                  BEGIN (*FACTOR*)
874  0000E WRITELN(CTRACE,' IN FACTOR...');
875  00016                  TEST(FACBEGSYS,PASSSYMBOLS,24);
876  00028                  WHILE FACBEGSYS(.SYMBOL.) DO
877  0003D                     BEGIN
878  0003D                        IF SYMBOL = MINUS THEN M := TRUE ELSE M := FALSE;
879  0004C                        IF (SYMBOL = PLUS) OR (SYMBOL = MINUS) THEN GETASYMBOL;
880  0006E                        IF SYMBOL = NUMBER THEN
881  00078                           BEGIN
882  00078                              GEN(LDIOP,NUM,0,0,0,0);
883  0008F                              GETASYMBOL
884  0008F                           END
885  00097                        ELSE
886  00098                           IF SYMBOL = IDENT THEN
887  000A2                              BEGIN
888  000A2                                 I := POSITION(ID);
889  00085                                 IF I = 0 THEN
890  00087                                    BEGIN
891  00087                                       ERROR(11);
892  000C0                                       GETASYMBOL
893  000C0                                    END
894  000C8                                 ELSE
895  000C9                                    IF SYMBOLTABLE(.I.).KIND = CONSTANT THEN
896  000DA                                       BEGIN
897  000DA                                          GEN(LOIOP,
898  000D8                                              SYMBOLTABLE(.I.).VALUE,0,0,0,0);
899  000F7                                          GETASYMBOL
900  000F7                                       END
901  000FF                                    ELSE
902  00100                                       VARIABLE
903  00100                              END
904  00102                           ELSE
905  00103                              IF SYMBOL = LPAREN THEN
906  00100                                 BEGIN
907  00100                                    GETASYMBOL;
908  00115                                    FOR S:= NUL TO XOR DO
909  0011C                                       TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
910  00128                                    TEMPSYMBOLS(.RPAREN.) := TRUE;
911  0012D                                    EXPRESSION(TEMPSYMBOLS);
912  00133                                    IF SYMBOL = RPAREN THEN
913  00130                                       GETASYMBOL
914  0013D                                    ELSE
915  00146                                       ERROR(22)
916  00147                                 END
917  0014F                              ELSE
918  00150                                 IF SYMBOL = NOTSYM THEN
919  0015A                                    BEGIN
920  0015A                                       GETASYMBOL;
921  00162                                       FOR S := NUL TO XOR DO
922  00169                                          TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
923  00178                                       FACTOR(TEMPSYMBOLS);
924  00178                                       GEN(NOTOP,0,0,0,0,0)
925  00183                                    END;
926  00188                        FOR S:= NUL TO XOR DO TEMPSYMBOLS(.S.) := FALSE;
927  0019C                        TEMPSYMBOLS(.LPAREN.) := TRUE;
928  0019E                        TEST(PASSSYMBOLS,TEMPSYMBOLS,17);
929  001A9                        IF M = TRUE THEN
930  001AC                           BEGIN
931  001AC                              TEMPORARY := NEXTEMP;
932  00186                              GEN(STAOP,0,0,TEMPORARY,0,0);
933  001C7                              GEN(LOIOP,0,0,0,0,0);
934  001D7                              GEN(SUBOP,0,0,TEMPORARY,0,0);
935  001E8                              RELEASETEMP(1)
936  001E9                           END
937  001EE                     END
938  001EE                  END (*FACTOR*);
939  001F9 (*$FO*)
```

```
940   001F9            BEGIN (*TERM*)
941   0000E WRITELN(CTRACE,' IN TERM...');
942   00016               FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
943   0002C               TEMPSYMBOLS(.TIMES.) := TRUE;
944   0002E               TEMPSYMBOLS(.SLASH.) := TRUE;
945   00030               TEMPSYMBOLS(.ANDSYM.) := TRUE;
946   00032               FACTOR(TEMPSYMBOLS);
947   00035               WHILE (SYMBOL = TIMES) OR (SYMBOL = SLASH) OR (SYMBOL = ANDSYM) DO
948   0005A                  BEGIN
949   0005A                     TEMPORARY := NEXTEMP;
950   00062                     GEN(STADP,0,0,TEMPORARY,0,0);
951   00071                     NEXTOP := INVERSEOP(SYMBOL);
952   00083                     GETASYMBOL;
953   0008A                     FACTOR(TEMPSYMBOLS);
954   00080                     GEN(NEXTOP,0,0,TEMPORARY,0,0);
955   0009C                     RELEASETEMP(1)
956   0009D                  END
957   000A1            END (*TERM*);
958   000A8
959   000A8         BEGIN (*SIMPLEEXPRESSION*)
960   0000E WRITELN(CTRACE,' IN SIMPLEEXPRESSION...');
961   00016            IF (SYMBOL = PLUS) OR (SYMBOL = MINUS) THEN GETASYMBOL;
962   00032            FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
963   00048            TEMPSYMBOLS(.PLUS.) := TRUE;
964   0004A            TEMPSYMBOLS(.MINUS.) := TRUE;
965   0004C            TEMPSYMBOLS(.ORSYM.) := TRUE;
966   0004E            TEMPSYMBOLS(.XOR.) := TRUE;
967   00050            TERM(TEMPSYMBOLS);
968   00053            WHILE (SYMBOL = PLUS) OR (SYMBOL = MINUS) OR (SYMBOL = ORSYM) OR
969   00074                (SYMBOL = XOR) DO
970   00081               BEGIN
971   00081                  TEMPORARY := NEXTEMP;
972   00088                  GEN(STADP,0,0,TEMPORARY,0,0);
973   00096                  NEXTOP := INVERSEOP(SYMBOL);
974   000A6                  GETASYMBOL;
975   000AC                  TERM(TEMPSYMBOLS);
976   000AF                  GEN(NEXTOP,0,0,TEMPORARY,0,0);
977   000BD                  RELEASETEMP(1)
978   000BE               END
979   000C1         END (*SIMPLEEXPRESSION*);
980   000CB
981   000CB      BEGIN (*EXPRESSION*)
982   0000E WRITELN(CTRACE,' IN EXPRESSION...');
983   00016         FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
984   0002C         FOR S := EQL TO GEQ DO TEMPSYMBOLS(.S.) := TRUE;
985   00030         SIMPLEEXPRESSION(TEMPSYMBOLS);
986   00040         WHILE (SYMBOL = EQL) OR (SYMBOL = NEQ) OR (SYMBOL = LEQ) OR
987   0005E             (SYMBOL = LSS) OR (SYMBOL = GEQ) OR (SYMBOL = GTR) DO
988   00080            BEGIN
989   00080               TEMPORARY := NEXTEMP;
990   00086               GEN(STADP,0,0,TEMPORARY,0,0);
991   00093               NEXTOP := INVERSEOP(SYMBOL);
992   000A1               GETASYMBOL;
993   000A6               SIMPLEEXPRESSION(TEMPSYMBOLS);
994   000A9               GEN(NEXTOP,0,0,TEMPORARY,0,0);
995   000B6               RELEASETEMP(1)
996   000B7            END
997   000B9      END (*EXPRESSION*);
998   000C2 (*$FO*)
```

```
 999      000C2 (********
1000      000C2  PROCEDURE TO PARSE A PROCESS INVOCATION STATEMENT.
1001      000C2  ********)
1002      000C2
1003      000C2       PROCEDURE PROCESSINVOCATION;
1004      00004          VAR S: SYMBOLS;
1005      00005              TEMPORARY: INTEGER;
1006      00006              OP1,OP3: INTEGER;
1007      00008       BEGIN
1008      00005  WRITELN(CTRACE,' IN PROCESS...');
1009      00000          IF SYMBOL = LPAREN THEN
1010      00014             BEGIN
1011      00014                GETASYMBOL;
1012      00019                FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1013      00031                TEMPSYMBOLS(.RPAREN.) := TRUE;
1014      00034                TEMPSYMBOLS(.COMMA.) := TRUE;
1015      00037                EXPRESSION (TEMPSYMBOLS);
1016      00038                TEMPORARY := NEXTEMP;
1017      00041                GEN(STAOP,0,0,TEMPORARY,0,0);
1018      0004E                WHILE SYMBOL = COMMA DO
1019      00055                   BEGIN
1020      00055                      GETASYMBOL;
1021      0005A                      EXPRESSION(TEMPSYMBOLS);
1022      0005E                      GEN(STAOP,0,0,NEXTEMP,0,0)
1023      0006C                   END;
1024      00072                IF SYMBOL = RPAREN THEN GETASYMBOL ELSE ERROR(22)
1025      00080             END;
1026      00085          WITH SYMBOLTABLE(.I.) DO
1027      00093             BEGIN
1028      00093                IF STARTADDR.DEFINED THEN
1029      00096                   OP1 := STARTADDR.VAL
1030      00097                ELSE
1031      0009A                   BEGIN
1032      0009A                      OP1 := STARTADDR.LINK;
1033      00090                      STARTADDR.LINK := PGMSTADDR
1034      0009E                   END;
1035      000A0                IF DATASIZE.DEFINED THEN
1036      000A3                   OP3 := DATASIZE.VAL
1037      000A4                ELSE
1038      000A7                   BEGIN
1039      000A7                      OP3 := DATASIZE.LINK;
1040      000AA                      DATASIZE.LINK := PGMSTADDR
1041      000AB                   END;
1042      000A0                GEN(CREATEOP,OP1,LEV-PLEVEL,OP3,PARAMCNT,TEMPORARY);
1043      000C0                RELEASETEMP(PARAMCNT)
1044      000C2             END
1045      000C4       END (*PROCESSINVOCATION*);
1046      000CB (*$FO*)
```

```
1047  000C8 (********
1048  000C8  PROCEDURE TO PARSE AN ASSIGNMENT STATEMENT.
1049  000C8  ********)
1050  000C8
1051  000C8      PROCEDURE ASSIGNMENTSTATEMENT;
1052  00004
1053  00004         VAR INDEXNEEDED: BOOLEAN;
1054  00005             TEMPORARY: INTEGER;
1055  00006             S: SYMBOLS;
1056  00007      BEGIN
1057  00005 WRITELN(TRACE,' IN ASSIGNMENT...');
1058  00000         IF SYMBOLTABLE(.I.).KIND = ANARRAY THEN
1059  00010            BEGIN
1060  00010               INDEXNEEDED := TRUE;
1061  0001F               IF SYMBOL = LBRACKET THEN
1062  00026                  BEGIN
1063  00026                     GETASYMBOL;
1064  00028                     FOR S := NUL TO XOR 00 TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1065  00043                     TEMPSYMBOLS(.RBRACKET.) := TRUE;
1066  00046                     EXPRESSION(TEMPSYMBOLS);
1067  0004A                     TEMPORARY := NEXTEMP;
1068  00050                     GEN(STAOP,0,0,TEMPORARY,0,0);
1069  00050                     IF SYMBOL = RBRACKET THEN GETASYMBOL ELSE ERROR(29)
1070  00068                  END
1071  00070               ELSE
1072  00071                  ERROR(28)
1073  00072            END
1074  00077         ELSE
1075  00078            IF SYMBOLTABLE(.I.).KIND = WHOLEND THEN
1076  00088               INDEXNEEDED := FALSE
1077  00088            ELSE
1078  00088               ERROR(12);
1079  00091         IF (SYMBOL = EQL) OR (SYMBOL = BECOMES) THEN
1080  000A5            BEGIN
1081  000A5               IF SYMBOL = EQL THEN ERROR(13);
1082  00082               GETASYMBOL;
1083  00087               EXPRESSION(PASSSYMBOLS);
1084  00088               WITH SYMBOLTABLE(.I.) DO
1085  000C9                  IF INDEXNEEDED THEN
1086  000C8                     BEGIN
1087  000C8                        GEN(LOXOP,0,0,TEMPORARY,0,0);
1088  000D8                        RELEASETEMP(1);
1089  000D8                        GEN(STAOP,1,LEV-ALEVEL,FIRSTPLACE,0,0)
1090  000E9                     END
1091  000EE                  ELSE
1092  000EF                     GEN(STAOP,0,LEV-WLEVEL,WOISP,0,0)
1093  000F0            END
1094  00102         ELSE
1095  00103            ERROR(13)
1096  00104      END (*ASSIGNMENTSTATEMENT*);
1097  00112 (*$FO*)
```

```
1098    00112 (********
1099    00112  PROCEDURE TO PARSE A DUMP STATEMENT.
1100    00112  ********)
1101    00112
1102    00112       PROCEDURE DUMPSTATEMENT;
1103    00004
1104    00004       BEGIN
1105    00005 WRITELN(CTRACE,' IN DUMP...');
1106    0000D           GEN(DUMPOP,0,0,0,0,0)
1107    00015       END (*DUMPSTATEMENT*);
1108    0001F (*$FO*)
```

```
1109  0001F (*******
1110  0001F  PROCEDURE TO PARSE AN IF STATEMENT.
1111  0001F  *******)
1112  0001F
1113  0001F      PROCEDURE IFSTATEMENT;
1114  00004         VAR S: SYMBOLS;
1115  00005             BRTOENO: INTEGER;        (* HEAD OF LIST OF BRANCHES NEEDING FIXUP *)
1116  00006             SPOTOFIX: INTEGER;       (* ADDRESS OF OTHER TYPE BRANCH FOR FIXUP *)
1117  00007             K: INTEGER;
1118  00008       BEGIN
1119  00005 WRITELN(CTRACE,' IN IF...');
1120  0000D         BRTOENO := EMPTY;
1121  0000F         FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1122  00027         TEMPSYMBOLS(.THENSYM.) := TRUE;
1123  0002A         EXPRESSION(TEMPSYMBOLS);
1124  0002E         SPOTOFIX := PGMSTADDR;
1125  00030         GEN(BRFOP,0,0,0,0,0);
1126  00030         IF SYMBOL = THENSYM THEN GETASYMBOL ELSE ERROR(16);
1127  00050         FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1128  00068         TEMPSYMBOLS(.ELSEIF.) := TRUE;
1129  0006B         TEMPSYMBOLS(.ELSESYM.) := TRUE;
1130  0006E         TEMPSYMBOLS(.ENDSYM.) := TRUE;
1131  00071         SSEQUENCE(TEMPSYMBOLS);
1132  00077         WHILE SYMBOL = ELSEIF DO
1133  0007E             BEGIN
1134  0007E                 GETASYMBOL;
1135  00083                 K := PGMSTADDR;
1136  00085                 GEN(BROP,BRTOENO,0,0,0,0);
1137  00092                 BRTOENO := K;
1138  00094                 PGMSTORE(.SPOTOFIX.).ADDRESS := PGMSTADDR;
1139  0009E                 FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1140  00086                 TEMPSYMBOLS(.THENSYM.) := TRUE;
1141  000B9                 EXPRESSION(TEMPSYMBOLS);
1142  000BD                 SPOTOFIX := PGMSTADDR;
1143  0008F                 GEN(BRFOP,0,0,0,0,0);
1144  000CC                 IF SYMBOL = THENSYM THEN GETASYMBOL ELSE ERROR(16);
1145  0000F                 FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1146  000F7                 TEMPSYMBOLS(.ELSEIF.) := TRUE;
1147  000FA                 TEMPSYMBOLS(.ELSESYM.) := TRUE;
1148  000FD                 TEMPSYMBOLS(.ENDSYM.) := TRUE;
1149  00100                 SSEQUENCE(TEMPSYMBOLS)
1150  00102             END;
1151  00107         IF SYMBOL = ELSESYM THEN
1152  0010E             BEGIN
1153  0010E                 K := PGMSTADDR;
1154  00110                 GEN(BROP,BRTOENO,0,0,0,0);
1155  00110                 BRTOENO := K;
1156  0011F                 PGMSTORE(.SPOTOFIX.).ADDRESS := PGMSTADDR;
1157  00129                 GETASYMBOL;
1158  0012E                 FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1159  00146                 TEMPSYMBOLS(.ENDSYM.) := TRUE;
1160  00149                 SSEQUENCE(TEMPSYMBOLS)
1161  0014B             END
1162  0014F         ELSE
1163  00150             PGMSTORE(.SPOTOFIX.).ADDRESS := PGMSTADDR;
1164  0015A         IF SYMBOL = ENDSYM THEN GETASYMBOL ELSE ERROR(8);
1165  00160         WHILE BRTOENO <> EMPTY DO
1166  00170             BEGIN
1167  00170                 K := PGMSTORE(.BRTOENO.).ADDRESS;
1168  00178                 PGMSTORE(.BRTOENO.).ADDRESS := PGMSTADDR;
1169  00182                 BRTOENO := K
1170  00182             END
1171  00184       END (*IFSTATEMENT*);
1172  0018B (*$FO*)
```

```
1173    0018B (********
1174    0018B  PROCEDURE TO PARSE A LOOP STATEMENT.
1175    0018B  ********)
1176    0018B
1177    0018B       PROCEDURE LOOPSTATEMENT;
1178    00004          VAR S: SYMBOLS;
1179    00005              LOOPSTART: INTEGER;      (* START ADDRESS OF LOOP *)
1180    00006              BRTOEND: INTEGER;        (* HEAD OF LIST OF BRANCHES NEEDING FIXUP *)
1181    00007              SPOTOFIX: INTEGER;       (* ADDRESS OF OTHER TYPE BRANCH FOR FIXUP *)
1182    00008              K: INTEGER;
1183    00009       BEGIN
1184    00005 WRITELN(CTRACE,' IN LOOP...');
1185    0000D          LOOPSTART := PGMSTADDR;
1186    0000F          BRTOEND := EMPTY;
1187    00011          FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1188    00029          TEMPSYMBOLS(.ENDSYM.) := TRUE;
1189    0002C          TEMPSYMBOLS(.WHEN.) := TRUE;
1190    0002F          SSEQUENCE(TEMPSYMBOLS);
1191    00035          WHILE SYMBOL = WHEN DO
1192    0003C              BEGIN
1193    0003C                 GETASYMBOL;
1194    00041                 FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1195    00059                 TEMPSYMBOLS(.DOSYM.) := TRUE;
1196    0005C                 TEMPSYMBOLS(.EXIT.) := TRUE;
1197    0005F                 EXPRESSION(TEMPSYMBOLS);
1198    00063                 SPOTOFIX := PGMSTADDR;
1199    00065                 GEN(BRFOP,0,0,0,0,0);
1200    00072                 IF SYMBOL = DOSYM THEN
1201    00079                    BEGIN
1202    00079                       GETASYMBOL;
1203    0007E                       FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1204    00096                       TEMPSYMBOLS(.EXIT.) := TRUE;
1205    00099                       SSEQUENCE(TEMPSYMBOLS)
1206    0009B                    END;
1207    0009F                 IF SYMBOL = EXIT THEN
1208    000A6                    BEGIN
1209    000A6                       K := PGMSTADDR;
1210    000A8                       GEN(BROP,BRTOEND,0,0,0,0);
1211    000B5                       BRTOEND := K;
1212    000B7                       PGMSTORE(.SPOTOFIX.).ADDRESS := PGMSTADDR;
1213    000C1                       GETASYMBOL;
1214    000C6                       FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1215    000DE                       TEMPSYMBOLS(.ENDSYM.) := TRUE;
1216    000E1                       TEMPSYMBOLS(.WHEN.) := TRUE;
1217    000E4                       SSEQUENCE(TEMPSYMBOLS)
1218    000E6                    END
1219    000EA                 ELSE
1220    000EB                    ERROR(18)
1221    000EC              END;
1222    000F2              IF SYMBOL = ENDSYM THEN GETASYMBOL ELSE ERROR(8);
1223    00105              GEN(BROP,LOOPSTART,0,0,0,0);
1224    00112              WHILE BRTOEND <> EMPTY DO
1225    00115                 BEGIN
1226    00115                    K := PGMSTORE(.BRTOEND.).ADDRESS;
1227    00110                    PGMSTORE(.BRTOEND.).ADDRESS := PGMSTADDR;
1228    00127                    BRTOEND := K
1229    00127                 END
1230    00129       END (*LOOPSTATEMENT*);
1231    00130 (*$FO*)
```

```
1232    00130 (********
1233    00130  PROCEDURE TO PARSE A PWAIT STATEMENT.
1234    00130  ********)
1235    00130
1236    00130       PROCEDURE PWAITSTATEMENT;
1237    00004
1238    00004       BEGIN
1239    00005 WRITELN(CTRACE,' IN PWAIT...');
1240    0000D          IF SYMBOL = LPAREN THEN
1241    00014             BEGIN
1242    00014                GETASYMBOL;
1243    00019                IF SYMBOL = IDENT THEN
1244    00020                   BEGIN
1245    00020                      I := POSITION(ID);
1246    0002E                      IF I = 0 THEN
1247    00031                         ERROR(11)
1248    00032                      ELSE
1249    00038                         IF SYMBOLTABLE(.I.).KIND <> SEMAPHORE THEN ERROR(25);
1250    0004E                      GETASYMBOL;
1251    00053                      IF SYMBOL = RPAREN THEN GETASYMBOL ELSE ERROR(22)
1252    00061                   END
1253    00066                ELSE
1254    00067                   ERROR(4)
1255    00068             END
1256    0006D          ELSE
1257    0006E             ERROR(23);
1258    00074          WITH SYMBOLTABLE(.I.) DO
1259    00082             GEN(PWAITOP,0,LEV-WLEVEL,WDISP,0,0)
1260    00090       END (*PWAITSTATEMENT*);
1261    00098 (*$FO*)
```

```
1262  00098 (********
1263  00098  PROCEDURE TO PARSE A VSIGNAL STATEMENT.
1264  00098  ********)
1265  00098
1266  00098       PROCEDURE VSIGNALSTATEMENT;
1267  00004
1268  00004       BEGIN
1269  00005 WRITELN(CTRACE,' IN VSIGNAL...');
1270  0000D           IF SYMBOL = LPAREN THEN
1271  00014              BEGIN
1272  00014                 GETASYMBOL;
1273  00019                 IF SYMBOL = IDENT THEN
1274  00020                    BEGIN
1275  00020                       I := POSITION(ID);
1276  0002E                       IF I = 0 THEN
1277  00031                          ERROR(11)
1278  00032                       ELSE
1279  00038                          IF SYMBOLTABLE(.I.).KIND <> SEMAPHORE THEN ERROR(25);
1280  0004E                       GETASYMBOL;
1281  00053 *                     IF SYMBOL = RPAREN THEN GETASYMBOL ELSE ERROR(22)
1282  00061                    END
1283  00066                 ELSE
1284  00067                    ERROR(4)
1285  00068              END
1286  0006D           ELSE
1287  0006E              ERROR(23);
1288  00074           WITH SYMBOLTABLE(.I.) DO
1289  00082              GEN(VSIGNALOP,0,LEV-WLEVEL,WDISP,0,0)
1290  00090       END (*VSIGNALSTATEMENT*);
1291  0009C (*$FO*)
```

```
1292  0009C (********
1293  0009C  PROCEDURE TO PARSE A SET STATEMENT.
1294  0009C  ********)
1295  0009C
1296  0009C      PROCEDURE SETSTATEMENT;
1297  00004        VAR S: SYMBOLS;
1298  00005      BEGIN
1299  00005 WRITELN(CTRACE,' IN SET...');
1300  0000D        IF SYMBOL = LPAREN THEN
1301  00014          BEGIN
1302  00014            GETASYMBOL;
1303  00019            IF SYMBOL = IDENT THEN
1304  00020              BEGIN
1305  00020                I := POSITION(ID);
1306  0002E                IF I = 0 THEN
1307  00031                  ERROR(11)
1308  00032                ELSE
1309  0003B                  IF SYMBOLTABLE(.I.).KIND <> SEMAPHORE THEN ERROR(25);
1310  0004E                GETASYMBOL;
1311  00053                IF SYMBOL = COMMA THEN GETASYMBOL ELSE ERROR(34);
1312  00066                FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1313  0007E                TEMPSYMBOLS(.RPAREN.) := TRUE;
1314  000B1                EXPRESSION(TEMPSYMBOLS);
1315  000B5                WITH SYMBOLTABLE(.I.) DO
1316  00093                  BEGIN
1317  00093                    GEN(STAOP,0,LEV-WLEVEL,WDISP,0,0);
1318  000A6                    GEN(LDIOP,EMPTY,0,0,0,0);
1319  000B3                    GEN(STAOP,0,LEV-WLEVEL,WDISP+1,0,0)
1320  000C2                  END;
1321  000C7                IF SYMBOL = RPAREN THEN GETASYMBOL ELSE ERROR(22)
1322  00005              END
1323  000DA            ELSE
1324  000DB              ERROR(4)
1325  000DC          END
1326  000E1        ELSE
1327  000E2          ERROR(23)
1328  000E3      END (*SETSTATEMENT*);
1329  000EF (*$FD*)
```

```
1330  000EF (********
1331  000EF  PROCEDURE TO PARSE A READ STATEMENT.
1332  000EF  ********)
1333  000EF
1334  000EF       PROCEDURE READSTATEMENT;
1335  00004          VAR S: SYMBOLS;
1336  00005              INDEXNEEDED: BOOLEAN;
1337  00006              TEMPORARY: INTEGER;
1338  00007       BEGIN
1339  00005 WRITELN(CTRACE,' IN READ...');
1340  00000          GEN(READOP,0,0,0,0,0);
1341  0001A          IF SYMBOL = LPAREN THEN
1342  00021             BEGIN
1343  00021                GETASYMBOL;
1344  00026                IF SYMBOL = IDENT THEN
1345  00020                   BEGIN
1346  00020                      I := POSITION(ID);
1347  00038                      IF I = 0 THEN
1348  0003E                         ERROR(11)
1349  0003F                      ELSE
1350  00045                         BEGIN
1351  00045                            IF SYMBOLTABLE[.I.].KIND = ANARRAY THEN
1352  00055                               BEGIN
1353  00055                                  TEMPORARY := NEXTEMP;
1354  00058                                  GEN(STADP,0,0,TEMPORARY,0,0);
1355  00068                                  GETASYMBOL;
1356  0006D                                  IF SYMBOL = LBRACKET THEN
1357  00074                                     BEGIN
1358  00074                                        GETASYMBOL;
1359  00079                                        FOR S := NUL TO XOR 00
1360  00080                                           TEMPSYMBOLS[.S.] := PASSSYMBOLS[.S.];
1361  00091                                        TEMPSYMBOLS[.RBRACKET.] := TRUE;
1362  00094                                        EXPRESSION(TEMPSYMBOLS);
1363  00098                                        GEN(STAXOP,0,0,0,0,0);
1364  000A5                                        INDEXNEEDED := TRUE;
1365  000A7                                        IF SYMBOL = RBRACKET THEN
1366  000AE                                           GETASYMBOL
1367  000AE                                        ELSE
1368  00084                                           ERROR(29)
1369  00085                                     END
1370  0008A                                  ELSE
1371  00088                                     ERROR(28);
1372  000C1                                  GEN(LOADP,0,0,TEMPORARY,0,0);
1373  000CE                                  RELEASETEMP(1)
1374  000CF                               END
1375  000D1                            ELSE
1376  000D2                               IF SYMBOLTABLE[.I.].KIND = WHOLENO THEN
1377  000E2                                  BEGIN
1378  000E2                                     INDEXNEEDED := FALSE;
1379  000E4                                     GETASYMBOL
1380  000E4                                  END
1381  000E9                               ELSE
1382  000EA                                  ERROR(12)
1383  000E8                         END
1384  000F0                   END
1385  000F0                ELSE
1386  000F1                   ERROR(10);
1387  000F7                IF SYMBOL = RPAREN THEN GETASYMBOL ELSE ERROR(22)
1388  00105             END
1389  0010A          ELSE
1390  0010B             ERROR(23);
1391  00111          WITH SYMBOLTABLE[.I.] DO
1392  0011F             IF INDEXNEEDED THEN
1393  00121                GEN(STADP,1,LEV-ALEVEL,FIRSTPLACE,0,0)
1394  0012F             ELSE
1395  00135                GEN(STADP,0,LEV-WLEVEL,WDISP,0,0)
1396  00143       END (*READSTATEMENT*);
1397  0014F (*$FO*)
```

```
1398  0014F (********
1399  0014F  PROCEDURE TO PARSE A WRITE STATEMENT.
1400  0014F  ********)
1401  0014F
1402  0014F       PROCEDURE WRITESTATEMENT;
1403  00004          VAR S: SYMBOLS;
1404  00005       BEGIN
1405  00005 WRITELN(CTRACE,' IN WRITE...');
1406  00000          IF SYMBOL = LPAREN THEN
1407  00014             BEGIN
1408  00014                GETASYMBOL;
1409  00019                FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1410  00031                TEMPSYMBOLS(.RPAREN.) := TRUE;
1411  00034                EXPRESSION(TEMPSYMBOLS);
1412  0003B                IF SYMBOL = RPAREN THEN GETASYMBOL ELSE ERROR(22)
1413  00046             END
1414  00048          ELSE
1415  0004C             ERROR(23);
1416  00052          GEN(WRITEOP,0,0,0,0,0)
1417  0005A       END (*WRITESTATEMENT*);
1418  00065 (*$FO*)
```

```
1419  00065   BEGIN (*STATEMENT*)
1420  0000E WRITELN(CTRACE,' IN STATEMENT...');
1421  00016      CASE SYMBOL OF
1422  0001F
1423  0001F   IDENT:   BEGIN
1424  0001F               I := POSITION(ID);
1425  0002A               IF I = 0 THEN
1426  0002C                  ERROR(11)
1427  0002D               ELSE
1428  00032                  IF SYMBOLTABLE(.I.).KINO = PROCESS THEN
1429  00040                     BEGIN
1430  00040                        GETASYMBOL;
1431  00044                        PROCESSINVOCATION
1432  00044                     END
1433  00046                  ELSE
1434  00047                     BEGIN
1435  00047                        GETASYMBOL;
1436  0004B                        ASSIGNMENTSTATEMENT
1437  0004B                     END
1438  0004D            END;
1439  0004E
1440  0004E   DUMP:    BEGIN
1441  0004E               GETASYMBOL;
1442  00052               DUMPSTATEMENT
1443  00052            END;
1444  00055
1445  00055   IFSYM:   BEGIN
1446  00055               GETASYMBOL;
1447  00059               IFSTATEMENT
1448  00059            END;
1449  0005C
1450  0005C   LOOP:    BEGIN
1451  0005C               GETASYMBOL;
1452  00060               LOOPSTATEMENT
1453  00060            END;
1454  00063
1455  00063   PWAIT:   BEGIN
1456  00063               GETASYMBOL;
1457  00067               PWAITSTATEMENT
1458  00067            END;
1459  0006A
1460  0006A   VSIGNAL: BEGIN
1461  0006A               GETASYMBOL;
1462  0006E               VSIGNALSTATEMENT
1463  0006E            END;
1464  00071
1465  00071   SETSYM:  BEGIN
1466  00071               GETASYMBOL;
1467  00075               SETSTATEMENT
1468  00075            END;
1469  00078
1470  00078   READSYM: BEGIN
1471  00078               GETASYMBOL;
1472  0007C               READSTATEMENT
1473  0007C            END;
1474  0007F
1475  0007F   WRITESYM: BEGIN
1476  0007F               GETASYMBOL;
1477  00083               WRITESTATEMENT
1478  00083            END;
1479  00086      END (*CASE*);
1480  00085
1481  00085      FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := FALSE;
1482  000C6      TEST(PASSSYMBOLS,TEMPSYMBOLS,19)
1483  000C9   END (*STATEMENT*);
1484  000D6 (*$FO*)
```

```
1485    00006   BEGIN (*SSEQUENCE*)
1486    0000E WRITELN(CTRACE,' IN SSEQUENCE...');
1487    00016       STATEMENT(PASSSYMBOLS);
1488    00019       WHILE SYMBOL = SEMICOLON DO
1489    0001E           BEGIN
1490    0001E               GETASYMBOL;
1491    00021               STATEMENT(PASSSYMBOLS)
1492    00022           END
1493    00024   END (*SSEQUENCE*);
1494    0002C
1495    0002C (*$FO*)
```

```
1496    0002C BEGIN (*BLOCK*)
1497    0000F WRITELN(CTRACE,' IN BLOCK...');
1498    00017 REPEAT
1499    00017   IF SYMBOL = CONSTSYM THEN
1500    0001B     BEGIN
1501    0001B       GETASYMBOL;
1502    00010       REPEAT
1503    00010         CONSTDECLARATION
1504    00010       UNTIL SYMBOL <> IDENT
1505    00021     END;
1506    00023   IF SYMBOL = VARSYM THEN
1507    00027     BEGIN
1508    00027       GETASYMBOL;
1509    00029       REPEAT
1510    00029         VARDECLARATION
1511    00029       UNTIL SYMBOL <> IDENT
1512    00020     END;
1513    0002F   TEMPHIGH := DISP;
1514    00031   WHILE SYMBOL = PROCSYM DO
1515    00035     BEGIN
1516    00035 *       GETASYMBOL;
1517    00037       PROCDECLARATION
1518    00037     END
1519    00039 UNTIL NOT ((SYMBOL = CONSTSYM) OR (SYMBOL = VARSYM) OR (SYMBOL = PROCSYM));
1520    00051   IF SYMBOL = BEGINSYM THEN
1521    00055     BEGIN
1522    00055       PSTART := PGMSTADDR;
1523    00058       GETASYMBOL;
1524    0005A       FOR S := NUL TO XOR DO TEMPSYMBOLS(.S.) := PASSSYMBOLS(.S.);
1525    00070       TEMPSYMBOLS(.ENDSYM.) := TRUE;
1526    00072       SSEQUENCE(TEMPSYMBOLS);
1527    00075       DSIZE := TEMPHIGH;
1528    00078       GEN(TERMOP,0,0,0,0,0);
1529    00082       IF SYMBOL = ENDSYM THEN GETASYMBOL ELSE ERROR(8)
1530    0008A     END
1531    0008C   ELSE
1532    0008D     ERROR(33)
1533    0008E END (*BLOCK*);
1534    00096 (*$FD*)
```

```
1535      00096 BEGIN (*COMPILE*)
1536      00005    FOR CH := CHR(0) TO CHR(255) DO TRANSTABLE(.CH.) := NULL;
1537      0001E    FOR CH := 'A' TO 'I' DO TRANSTABLE(.CH.) := LETTER;
1538      0002F    FOR CH := 'J' TO 'R' DO TRANSTABLE(.CH.) := LETTER;
1539      00040    FOR CH := 'S' TO 'Z' DO TRANSTABLE(.CH.) := LETTER;
1540      00051    FOR CH := '0' TO '9' DO TRANSTABLE(.CH.) := DIGIT;
1541      00062    TRANSTABLE(.' '.) := BLANK;
1542      00064    TRANSTABLE(.'='.) := 1;
1543      00066    TRANSTABLE(.')'.) := 2;
1544      00068    TRANSTABLE(.'+'.) := 3;
1545      0006A    TRANSTABLE(.'-'.) := 4;
1546      0006C    TRANSTABLE(.'*'.) := 5;
1547      0006E    TRANSTABLE(.'/'.) := 6;
1548      00070    TRANSTABLE(.';'.) := 7;
1549      00072    TRANSTABLE(.','.) := 8;
1550      00074
1551      00074    WORD(.1.) := 'AND         ';
1552      00078    WORD(.2.) := 'ARRAY       ';
1553      0007C    WORD(.3.) := 'BEGIN       ';
1554      00080    WORD(.4.) := 'CONST       ';
1555      00084    WORD(.5.) := 'DO          ';
1556      00088    WORD(.6.) := 'DUMP        ';
1557      0008C    WORD(.7.) := 'ELSE        ';
1558      00090    WORD(.8.) := 'ELSIF       ';
1559      00094    WORD(.9.) := 'END         ';
1560      00098    WORD(.10.) := 'EXIT       ';
1561      0009C    WORD(.11.) := 'IF         ';
1562      000A0    WORD(.12.) := 'INTEGER    ';
1563      000A4    WORD(.13.) := 'LOOP       ';
1564      000A8    WORD(.14.) := 'NOT        ';
1565      000AC    WORD(.15.) := 'OF         ';
1566      000B0    WORD(.16.) := 'OR         ';
1567      000B4    WORD(.17.) := 'PROCESS    ';
1568      000B8    WORD(.18.) := 'PWAIT      ';
1569      000BC    WORD(.19.) := 'READ       ';
1570      000C0    WORD(.20.) := 'SEMA       ';
1571      000C4    WORD(.21.) := 'SET        ';
1572      000C8    WORD(.22.) := 'THEN       ';
1573      000CC    WORD(.23.) := 'VAR        ';
1574      000D0    WORD(.24.) := 'VSIGNAL    ';
1575      00004    WORD(.25.) := 'WHEN       ';
1576      000D8    WORD(.26.) := 'WRITE      ';
1577      000DC    WORD(.27.) := 'XOR        ';
1578      000E0
1579      000E0    WORDSYM(.1.) := ANDSYM;
1580      000E2    WORDSYM(.2.) := ARRAYSYM;
1581      000E4    WORDSYM(.3.) := BEGINSYM;
1582      000E6    WORDSYM(.4.) := CONSTSYM;
1583      000E8    WORDSYM(.5.) := DOSYM;
1584      000EA    WORDSYM(.6.) := DUMP;
1585      000EC    WORDSYM(.7.) := ELSESYM;
1586      000EE    WORDSYM(.8.) := ELSEIF;
1587      000F0    WORDSYM(.9.) := ENDSYM;
1588      000F2    WORDSYM(.10.) := EXIT;
1589      000F4    WORDSYM(.11.) := IFSYM;
1590      000F6    WORDSYM(.12.) := INTSYM;
1591      000F8    WORDSYM(.13.) := LOOP;
1592      000FA    WORDSYM(.14.) := NOTSYM;
1593      000FC    WORDSYM(.15.) := OFSYM;
1594      000FE    WORDSYM(.16.) := ORSYM;
1595      00100    WORDSYM(.17.) := PROCSYM;
1596      00102    WORDSYM(.18.) := PWAIT;
1597      00104    WORDSYM(.19.) := READSYM;
1598      00106    WORDSYM(.20.) := SEMASYM;
1599      00108    WORDSYM(.21.) := SETSYM;
1600      0010A    WORDSYM(.22.) := THENSYM;
1601      0010C    WORDSYM(.23.) := VARSYM;
1602      0010E    WORDSYM(.24.) := VSIGNAL;
1603      00110    WORDSYM(.25.) := WHEN;
1604      00112    WORDSYM(.26.) := WRITESYM;
1605      00114    WORDSYM(.27.) := XOR;
1606      00116
1607      00116    SINGLESYM(.1.) := EQL;
1608      00118    SINGLESYM(.2.) := RPAREN;
1609      0011A    SINGLESYM(.3.) := PLUS;
1610      0011C    SINGLESYM(.4.) := MINUS;
```

```
1612    00120    SINGLESYM(.6.) := SLASH;
1613    00122    SINGLESYM(.7.) := SEMICOLON;
1614    00124    SINGLESYM(.8.) := COMMA;
1615    00126
1616    00126    FOR SYMBOL := NUL TO XOR OD DECLBEGSYS(.SYMBOL.) := FALSE;
1617    00137    DECLBEGSYS(.CONSTSYM.) := TRUE;
1618    00139    DECLBEGSYS(.PROCSYM.) := TRUE;
1619    0013B    DECLBEGSYS(.VARSYM.) := TRUE;
1620    0013D    FOR SYMBOL := NUL TO XOR OD STATBEGSYS(.SYMBOL.) := FALSE;
1621    0014E    STATBEGSYS(.IDENT.) := TRUE;
1622    00150    STATBEGSYS(.DUMP.) := TRUE;
1623    00152    STATBEGSYS(.IFSYM.) := TRUE;
1624    00154    STATBEGSYS(.LOOP.) := TRUE;
1625    00156    STATBEGSYS(.PWAIT.) := TRUE;
1626    00158    STATBEGSYS(.READSYM.) := TRUE;
1627    0015A    STATBEGSYS(.SETSYM.) := TRUE;
1628    0015C    STATBEGSYS(.VSIGNAL.) := TRUE;
1629    0015E    STATBEGSYS(.WRITESYM.) := TRUE;
1630    00160    FOR SYMBOL := NUL TO XOR OD FACBEGSYS(.SYMBOL.) := FALSE;
1631    00171    FACBEGSYS(.IDENT.) := TRUE;
1632    00173    FACBEGSYS(.LPAREN.) := TRUE;
1633    00175    FACBEGSYS(.NOTSYM.) := TRUE;
1634    00177    FACBEGSYS(.NUMBER.) := TRUE;
1635    00179
1636    00179    PAGE(OUTPUT);
1637    00180    DATE(TODAY);
1638    00182    WRITELN(' CONCUR COMPILER VERSION 1 JANUARY,1978',' ':20,TODAY:16);
1639    00193    WRITELN(' OPTIONS: LS...');
1640    0019B    COUNT := 0;
1641    0019D    LLENGTH := 0;
1642    0019F    LNUMBER := 0;
1643    001A1    CHCLASS := BLANK;
1644    001A3    TEMPMAX := IOMAX;
1645    001A5    FOR SYMBOL := NUL TO XOR OD FOLLOWSYMBOLS(.SYMBOL.) := FALSE;
1646    001B6    FOR SYMBOL := NUL TO XOR OD
1647    001B0       FOLLOWSYMBOLS(.SYMBOL.) := FOLLOWSYMBOLS(.SYMBOL.) OR
1648    001C7                                  DECLBEGSYS(.SYMBOL.) OR
1649    001C0                                  STATBEGSYS(.SYMBOL.);
1650    001D8    FOLLOWSYMBOLS(.SEMICOLON.) := TRUE;
1651    001DA    FOLLOWSYMBOLS(.PERIOD.) := TRUE;
1652    001DC    GETASYMBOL;
1653    001DE    BLOCK(0,0,0,FOLLOWSYMBOLS,MAINSTART,MAINDATASIZE);
1654    001E8    IF SYMBOL <> PERIOD THEN ERROR(9);
1655    001EE    WRITE(ERRCNT:3,' ERRORS FOUND.');
1656    001F8 9999:
1657    001F8    WRITELN
1658    001F8 END (*COMPILE*);
1659    00263 (*$FO*)
```

```
1660  00263 (********
1661  00263  PROCEDURE TO LIST THE CODE GENERATED BY THE COMPILER.
1662  00263  ********)
1663  00263
1664  00263    PROCEDURE LISTCODE;
1665  00004       VAR INDEX: INTEGER;
1666  00005    BEGIN
1667  00005       OPLIST(.ADDOP.) := 'ADD      ';
1668  00009       OPLIST(.SUBOP.) := 'SUB      ';
1669  0000D       OPLIST(.RSUBOP.) := 'RSUB     ';
1670  00011       OPLIST(.MULTOP.) := 'MULT     ';
1671  00015       OPLIST(.DIVOP.) := 'DIV      ';
1672  00019       OPLIST(.RDIVOP.) := 'RDIV     ';
1673  0001D       OPLIST(.EQLOP.) := 'EQL      ';
1674  00021       OPLIST(.NEQOP.) := 'NEQ      ';
1675  00025       OPLIST(.LSSOP.) := 'LSS      ';
1676  00029       OPLIST(.LEQOP.) := 'LEQ      ';
1677  0002D       OPLIST(.GTROP.) := 'GTR      ';
1678  00031       OPLIST(.GEQOP.) := 'GEQ      ';
1679  00035       OPLIST(.ANDOP.) := 'AND      ';
1680  00039       OPLIST(.OROP.) := 'OR       ';
1681  0003D       OPLIST(.XOROP.) := 'XOR      ';
1682  00041       OPLIST(.NOTOP.) := 'NOT      ';
1683  00045       OPLIST(.LDXOP.) := 'LDX      ';
1684  00049       OPLIST(.STXOP.) := 'STX      ';
1685  0004D       OPLIST(.LOADP.) := 'LDA      ';
1686  00051       OPLIST(.STAOP.) := 'STA      ';
1687  00055       OPLIST(.LDIOP.) := 'LDI      ';
1688  00059       OPLIST(.STAXOP.) := 'STAX     ';
1689  00050       OPLIST(.PWAITOP.) := 'PWAIT    ';
1690  00061       OPLIST(.VSIGNALOP.) := 'VSIGNAL  ';
1691  00065       OPLIST(.BROP.) := 'BR       ';
1692  00069       OPLIST(.BRFOP.) := 'BRF      ';
1693  0006D       OPLIST(.READOP.) := 'READ     ';
1694  00071       OPLIST(.WRITEOP.) := 'WRITE    ';
1695  00075       OPLIST(.CREATEOP.) := 'CREATE   ';
1696  00079       OPLIST(.TERMOP.) := 'TERMINATE';
1697  0007D       OPLIST(.DUMPOP.) := 'DUMP     ';
1698  00081
1699  00081       PAGE(OLIST);
1700  00088       WRITELN(OLIST,'  GENERATED CODE');
1701  00090       WRITELN(OLIST,'0 INDEX     OPERATION          OPERANDS');
1702  00098       INDEX := 1;
1703  0009A       WHILE INDEX < PGMSTADDR DO
1704  0009D          BEGIN
1705  0009D             WITH PGMSTORE(.INDEX.) DO
1706  000A7                BEGIN
1707  000A7                   WRITE(OLIST,'   ',INDEX:3,' ':6,OPLIST(.OPCODE.):9,' ':5);
1708  000C5                   CASE OPCODE OF
1709  000CC             ADDOP,SUBOP,RSUBOP,
1710  000CC             MULTOP,DIVOP,RDIVOP,
1711  000CC             EQLOP,NEQOP,LSSOP,
1712  000CC             LEQOP,GTROP,GEQOP,
1713  000CC             ANDOP,OROP,XOROP,
1714  000CC             LDXOP,STXOP,LOADP,
1715  000CC             STAOP,PWAITOP,VSIGNALOP:
1716  000CC                            WRITELN(OLIST,ORD(INDEXFLAG),'  ':3,LEVEL,' ':3,DISPLACEMENT);
1717  000E8
1718  000E8             BROP,BRFOP: WRITELN(OLIST,ADDRESS);
1719  000F1
1720  000F1             LDIOP:      WRITELN(OLIST,CONSTANT);
1721  000FA
1722  000FA             NOTOP,STAXOP,TERMOP,
1723  000FA             READOP,WRITEOP,DUMPOP:
1724  000FA                            WRITELN(OLIST);
1725  000FD
1726  000FD             CREATEOP:  WRITELN(OLIST,INITIALPC,'  ':3,SLINK,' ':3,DSIZE,' ':3,
1727  00118                                PARAMCNT,' ':3,PARAMSTRT);
1728  00128                   END (*CASE*);
1729  0014A
1730  0014A                   INDEX := INDEX + 1
1731  0014A                END
1732  00148          END
1733  00148    END (*LISTCODE*);
1734  00188 (*$FO*)
```

```
1735  00188 (********
1736  00188  PROCEDURE TO INTERPRET CODE GENERATED BY CONCUR COMPILER.
1737  00188  ********)
1738  00188
1739  00188   PROCEDURE INTERPRET;
1740  00004
1741  00004      LABEL 999;
1742  00004
1743  00004      CONST DESCSTLIMIT = 40;        (* MAX. SIZE OF DESCRIPTOR STORE *)
1744  00004            DATASTLIMIT = 300;       (* MAX. SIZE OF DATA STORE *)
1745  00004
1746  00004      TYPE STATUS = (FREE,READY,WAITING,TERMINATED);
1747  00004           DESCRIPTORS = RECORD
1748  00004                           STATUSCODE: STATUS;  (*CURRENT STATUS OF PROCESS*)
1749  00004                           DESCENDENTCNT: INTEGER; (*NO OF DIRECT DESCENDENTS*)
1750  00004                           STATICLINK: INTEGER;  (*STATIC LINK*)
1751  00004                           RWFLINK: INTEGER;  (*FORWARD LINK FOR READY OR WAIT*)
1752  00004                           RWBLINK: INTEGER;  (*BACKWARD LINK FOR SAME LISTS*)
1753  00004                           DATASTART: INTEGER;  (*PTR TO START OF DATA AREA*)
1754  00004                           DATASIZE: INTEGER;  (*SIZE OF DATA AREA*)
1755  00004                           PGMCNTR: INTEGER;  (*PROGRAM COUNTER*)
1756  00004                           XREG: INTEGER;  (*INDEX REGISTER*)
1757  00004                           ACC: INTEGER   (*ACCUMULATOR*)
1758  00004                         END;
1759  00004
1760  00004      VAR CURRPRO: INTEGER;    (*PTR TO CURRENT PROCESS DESCRIPTOR*)
1761  00005          CURRPC: INTEGER;     (*PTR TO CURRENT INSTRUCTION*)
1762  00006          FREELIST: INTEGER;   (*HEAD OF FREE STORAGE LIST*)
1763  00007          STAR: INTEGER;       (*STORAGE ACCESS REGISTER*)
1764  00008          READYLIST: INTEGER;  (*"HEAD" OF READY LIST*)
1765  00009          TEMP: INTEGER;       (*TEMPORARY VARIABLE*)
1766  0000A          NEXTREADY: INTEGER;  (*HOLD AREA FOR NEXT READY PROCESS*)
1767  0000B          DESCSTORE: ARRAY(.1..DESCSTLIMIT.) OF DESCRIPTORS; (*DESCRIPTOR STORE*)
1768  0019B          DATASTORE: ARRAY(.1..DATASTLIMIT.) OF INTEGER;  (*DATA STORE*)
1769  002C7          I: INTEGER;
1770  002C8          ALPHASTAT: ARRAY(.STATUS.) OF CHAR;
1771  002CC          COUNT: INTEGER; (* NUMBER OF INSTRUC. TO EXECUTE FOR CURR. PROCESS *)
1772  002CD
1773  002CD (********
1774  002CD  PROCEDURE TO CALCULATE THE ACTUAL ADDRESS OF A TYPICAL OPERAND.
1775  002CD  ********)
1776  002CD
1777  002CD      PROCEDURE SETSTAR;
1778  00004          VAR I,J: INTEGER;
1779  00006      BEGIN
1780  00005         WITH DESCSTORE(.CURRPRO.) DO
1781  00010         WITH PGMSTORE(.CURRPC.)  DO
1782  0001B            BEGIN
1783  0001B               IF INDEXFLAG = FALSE THEN STAR := 0 ELSE STAR := XREG - 1;
1784  00027               STAR := STAR + DISPLACEMENT;
1785  0002D               I := LEVEL;
1786  00030               J := CURRPRO;
1787  00033               WHILE I>0 DO
1788  00035                  BEGIN
1789  00035                     J := DESCSTORE(.J.).STATICLINK;
1790  0003E                     I := I - 1
1791  0003E                  END;
1792  00040               STAR := STAR + DESCSTORE(.J.).DATASTART
1793  00049            END
1794  0004E      END (*SETSTAR*);
1795  0004F (*$F0*)
```

```
1796  0004F (********
1797  0004F  PROCEDURE TO DELETE A DESCRIPTOR FROM A LIST.
1798  0004F  ********)
1799  0004F
1800  0004F       PROCEDURE DELETE(PROPTR: INTEGER; VAR LISTHEAD: INTEGER);
1801  00006       BEGIN
1802  00007          WITH DESCSTORE(.PROPTR.) DO
1803  00011             IF PROPTR = RWFLINK THEN
1804  00015                LISTHEAD := EMPTY
1805  00016             ELSE
1806  00019                BEGIN
1807  00019                   DESCSTORE(.RWBLINK.).RWFLINK := RWFLINK;
1808  00026                   DESCSTORE(.RWFLINK.).RWBLINK := RWBLINK;
1809  00033                   IF LISTHEAD = PROPTR THEN LISTHEAD := RWFLINK
1810  00039                END
1811  0003B       END (*DELETE*);
1812  0003C
1813  0003C (********
1814  0003C  PROCESS TO INSERT A DESCRIPTOR INTO A LIST.
1815  0003C  ********)
1816  0003C
1817  0003C       PROCEDURE INSERT(PROPTR: INTEGER; VAR LISTHEAD: INTEGER);
1818  00006       BEGIN
1819  00007          WITH DESCSTORE(.PROPTR.) DO
1820  00011             IF LISTHEAD = EMPTY THEN
1821  00015                BEGIN
1822  00015                   LISTHEAD := PROPTR;
1823  00018                   RWFLINK := PROPTR;
1824  0001B                   RWBLINK := PROPTR
1825  0001C                END
1826  0001E             ELSE
1827  0001F                BEGIN
1828  0001F                   RWBLINK := DESCSTORE(.LISTHEAD.).RWBLINK;
1829  0002A                   DESCSTORE(.LISTHEAD.).RWBLINK := PROPTR;
1830  00036                   RWFLINK := DESCSTORE(.RWBLINK.).RWFLINK;
1831  00041                   DESCSTORE(.RWBLINK.).RWFLINK := PROPTR
1832  00049                END
1833  00040       END (*INSERT*);
1834  0004E (*$FO*)
```

```
1835      0004E (********
1836      0004E  PROCEDURE TO DUMP "REGISTERS", DESCSTORE, AND DATASTORE.
1837      0004E  ********)
1838      0004E
1839      0004E       PROCEDURE DUMPDSTORES;
1840      00004          VAR I,J,K: INTEGER;
1841      00007              DATAEND: INTEGER;
1842      00008       BEGIN
1843      00005          PAGE(OUTPUT);
1844      0000C          WRITELN(' CURRENT PROCESS =',CURRPRD:3,'   CURRENT PC =',CURRPC:4);
1845      00023          WRITELN(' READY LIST HEAD =',READYLIST:3,'   FREE LIST HEAD =',FREELIST:4);
1846      0003A          WRITE('0PRDCESS                NUMBER OF    STATIC    FORWARD    8ACKWARD');
1847      0003F          WRITELN('    DATA STORE    PROGRAM    INDEX');
1848      00047          WRITE('  NUMBER   STATUS   DESCENDENTS    LINK      LINK       LINK');
1849      0004C          WRITELN('     START   SIZE   COUNTER   REGISTER   ACCUMULATOR');
1850      00054          J := 0;
1851      00056          FOR I := 1 TO DESCSTLIMIT DO
1852      0005A             WITH DESCSTORE(.I.) DO
1853      00064                IF DRD(STATUSCODE) <> 0 THEN
1854      00067                   BEGIN
1855      00067                      WRITELN(' ',I:5,'        ',
1856      00077                                 ALPHASTAT(.STATUSCODE.),
1857      00083                                 DESCENDENTCNT:11,
1858      00088                                 STATICLINK:12,
1859      0008D                                 RWFLINK:9,
1860      00092                                 RWBLINK:11,
1861      00097                                 DATASTART:10,
1862      0009C                                 DATASIZE:7,
1863      000A1                                 PGMCNTR:9,
1864      000A6                                 XREG:10,
1865      000A8                                 ACC:16);
1866      000B1                      J := J + 1
1867      00081                   END;
1868      00085          IF J = 0 THEN
1869      00087             WRITELN('0ND ACTIVE PROCESSES!')
1870      0008E          ELSE
1871      000C0             FOR I := 1 TO DESCSTLIMIT DO
1872      000C4                WITH DESCSTORE(.I.) DO
1873      000CE                   IF DRD(STATUSCODE) <> 0 THEN
1874      000D1                      BEGIN
1875      000D1                         WRITELN('0DATA STORE FOR PROCESS NUMBER',I:3,':');
1876      000E2                         J := DATASTART;
1877      000E5                         DATAEND := DATASTART + DATASIZE - 1;
1878      000E8                         WHILE J <= DATAEND DO
1879      000EE                            BEGIN
1880      000EE                               K := 0;
1881      000F0                               WHILE (K < 10) AND (J <= DATAEND) DO
1882      000FC                                  BEGIN
1883      000FC                                     WRITE(' ',DATASTORE(.J.));
1884      0010C                                     K := K + 1;
1885      0010D                                     J := J + 1
1886      0010D                                  END;
1887      0010F                               WRITELN
1888      0010F                            END
1889      00111                      END
1890      00112       END (*DUMPDSTORES*);
1891      0016F (*$FD*)
```

```
1892  0016F (********
1893  0016F  PROCEDURE TO FIND A FREE DESCRIPTOR AND ALLOCATE DATA STORE.
1894  0016F  ********)
1895  0016F
1896  0016F   PROCEDURE ALLOCATE(VAR DSIZE,NEWDESC,NEWDATA: INTEGER);
1897  00007      CONST LINKSPACE = 2;   (*# OF WORDS NEEDED TO LINK FREE BLOCK*)
1898  00007
1899  00007      VAR I,J,K: INTEGER;
1900  0000A   BEGIN
1901  00007      NEWDESC := EMPTY;
1902  0000A      I := 1;
1903  0000C      WHILE (NEWDESC = EMPTY) AND (I <= DESCSTLIMIT) DO
1904  00019         BEGIN
1905  00019            IF DESCSTORE(.I.).STATUSCODE = FREE THEN NEWDESC := I;
1906  00027            I := I + 1;
1907  00028         END;
1908  00029      IF NEWDESC = EMPTY THEN
1909  00020         BEGIN
1910  0002D            WRITELN(' ABNORMAL TERMINATION--DESCRIPTOR STORE OVERFLOW.');
1911  00035            DUMPDSTORES;
1912  00037            GOTO 999
1913  0003A         END;
1914  0003A      NEWDATA := EMPTY;
1915  0003D      I := FREELIST;
1916  00040      J := 0;
1917  00042      WHILE (NEWDATA = EMPTY) AND (I<>EMPTY) DO
1918  0004F         IF DATASTORE(.I+1.) >= DSIZE THEN
1919  0005A            BEGIN
1920  0005A               K := DATASTORE(.I + 1.) - DSIZE;
1921  00065               IF K <= LINKSPACE THEN
1922  00068                  BEGIN
1923  00068                     IF J = 0 THEN
1924  0006A                        FREELIST := DATASTORE(.I.)
1925  0006E                     ELSE
1926  00074                        DATASTORE(.J.) := DATASTORE(.I.);
1927  00082                     DSIZE := DSIZE + K;
1928  00087                     NEWDATA := I
1929  00088                  END
1930  0008A               ELSE
1931  0008B                  BEGIN
1932  0008B                     DATASTORE(.I + 1.) := K;
1933  00094                     NEWDATA := I + K
1934  00095                  END
1935  00098            END
1936  00098         ELSE
1937  00099            BEGIN
1938  00099               J := I;
1939  0009B               I := DATASTORE(.I.)
1940  0009F            END;
1941  000A4      IF NEWDATA = EMPTY THEN
1942  000A8         BEGIN
1943  000A8            WRITELN(' ABNORMAL TERMINATION--DATA STORE OVERFLOW.');
1944  000B0            DUMPDSTORES;
1945  000B2            GOTO 999
1946  000B5         END
1947  000B5   END (*ALLOCATE*);
1948  000CF (*$FO*)
```

```
1949  000CF (********
1950  000CF  PROCEDURE TO RELEASE A DESCRIPTOR AND RETURN ITS DATA AREA TO THE FREE LIST.
1951  000CF  ********)
1952  000CF
1953  000CF   PROCEDURE DEALLOCATE(PROPTR: INTEGER);
1954  00005      VAR I,J: INTEGER;
1955  00007   BEGIN
1956  00006      WITH DESCSTORE(.PROPTR.) DO
1957  00010         BEGIN
1958  00010            STATUSCODE := FREE;
1959  00013            J := 0;
1960  00015            I := FREELIST;
1961  00018            WHILE DATASTART > I DO
1962  0001C               BEGIN
1963  0001C                  J := I;
1964  0001E                  I := DATASTORE(.I.)
1965  00022               END;
1966  00027            IF J = 0 THEN
1967  00029               BEGIN
1968  00029                  IF DATASTART + DATASIZE = FREELIST THEN
1969  00030                     BEGIN
1970  00030                        DATASTORE(.DATASTART.) := DATASTORE(.FREELIST.);
1971  00040                        DATASTORE(.DATASTART + 1.) := DATASTORE(.FREELIST + 1.)
1972  0004C                                                   + DATASIZE
1973  00050                     END
1974  00054                  ELSE
1975  00055                     BEGIN
1976  00055                        DATASTORE(.DATASTART.) := FREELIST;
1977  0005F                        DATASTORE(.DATASTART + 1.) := DATASIZE
1978  00066                     END;
1979  0006A                  FREELIST := DATASTART
1980  00068               END
1981  0006E            ELSE
1982  0006F               IF (DATASTART + DATASIZE = I) AND
1983  00077                  (J + DATASTORE(.J + 1.) = DATASTART) THEN
1984  00087                  BEGIN
1985  00087                     DATASTORE(.J.) := DATASTORE(.I.);
1986  00095                     DATASTORE(.J + 1.) := DATASTORE(.J + 1.) + DATASIZE +
1987  000A4                                           DATASTORE(.I + 1.)
1988  000A9                  END
1989  000AF               ELSE
1990  00080                  IF DATASTART + DATASIZE = I THEN
1991  00086                     BEGIN
1992  00086                        DATASTORE(.DATASTART.) := DATASTORE(.I.);
1993  000C5                        DATASTORE(.DATASTART + 1.) := DATASIZE +
1994  000C0                                                      DATASTORE(.I + 1.);
1995  000D8                        DATASTORE(.J.) := DATASTART
1996  00000                     END
1997  000E1                  ELSE
1998  000E2                     IF J + DATASTORE(.J + 1.) = DATASTART THEN
1999  000EE                        DATASTORE(.J + 1.) := DATASTORE(.J + 1.) + DATASIZE
2000  000FC                     ELSE
2001  00101                        BEGIN
2002  00101                           DATASTORE(.J.) := DATASTART;
2003  0010A                           DATASTORE(.DATASTART.) := I;
2004  00113                           DATASTORE(.DATASTART + 1.) := DATASIZE
2005  0011A                        END
2006  0011E         END
2007  0011E   END (*DEALLOCATE*);
2008  0011F (*$FO*)
```

```
2009  0011F (********
2010  0011F  PROCEDURE TO CREATE A PROCESS DESCRIPTOR.
2011  0011F  ********)
2012  0011F
2013  0011F   PROCEDURE CREATEDESCRIPTOR(INITIALPC,SLINK,DSIZE,PARAMCNT,PARAMSTRT: INTEGER);
2014  00009      VAR NEWDESC: INTEGER;          (* PTR TO NEWLY CREATED DESCRIPTOR *)
2015  0000A          NEWDATA: INTEGER;          (* PTR TO NEWLY ALLOCATED DATA AREA *)
2016  00008          I: INTEGER;
2017  0000C      BEGIN
2018  00007          ALLOCATE(DSIZE,NEWDESC,NEWDATA);
2019  0000C          WITH DESCSTORE(.NEWDESC.) DO
2020  00016              BEGIN
2021  00016                  INSERT(NEWDESC,READYLIST);
2022  00018                  STATUSCODE := READY;
2023  0001E                  DESCENDENTCNT := 0;
2024  00021                  DATASTART := NEWDATA;
2025  00024                  DATASIZE := DSIZE;
2026  00027                  PGMCNTR := INITIALPC;
2027  0002A                  XREG := 0;
2028  0002D                  ACC := 0;
2029  00030                  IF SLINK = EMPTY THEN
2030  00033                      STATICLINK := EMPTY
2031  00034                  ELSE
2032  00037                      BEGIN
2033  00037                          I := CURRPRD;
2034  0003A                          WHILE SLINK > 0 DO
2035  0003C                              BEGIN
2036  0003C                                  I := DESCSTORE(.I.).STATICLINK;
2037  00045                                  SLINK := SLINK -1
2038  00045                              END;
2039  00047                          STATICLINK :=I;
2040  0004A                          DESCSTORE(.I.).DESCENDENTCNT :=
2041  00051                              DESCSTORE(.I.).DESCENDENTCNT + 1
2042  00059                      END;
2043  0005D                                  IF PARAMCNT > 0 THEN
2044  0005F                                      STAR := DESCSTORE(.CURRPRD.).DATASTART +
2045  00068                                              PARAMSTRT;
2046  00068                  FOR I := 1 TO PARAMCNT DO
2047  0006F                      DATASTORE(.DATASTART + I - 1.) := DATASTORE(.STAR + I - 1.)
2048  0007D              END
2049  00086   END (*CREATEDESCRIPTOR*);
2050  00088  (*$FO*)
```

```
2051     00088   BEGIN (*INTERPRET*)
2052     00005      PAGE(OUTPUT);
2053     0000C      READYLIST := EMPTY;
2054     0000E      FREELIST := 1;
2055     00010      DATASTORE(.1.) := EMPTY;
2056     00012      DATASTORE(.2.) := DATASTLIMIT;
2057     00014      FOR I := 1 TO DESCSTLIMIT DO DESCSTORE(.I.).STATUSCODE := FREE;
2058     00025      CREATEDESCRIPTOR(MAINSTART,EMPTY,MAINDATASIZE,0,0);
2059     0002D      ALPHASTAT(.FREE.) := 'F';
2060     0002F      ALPHASTAT(.READY.) := 'R';
2061     00031      ALPHASTAT(.WAITING.) := 'W';
2062     00033      ALPHASTAT(.TERMINATED.) := 'T';
2063     00035      WHILE READYLIST <> EMPTY DO
2064     00038         BEGIN
2065     00038            CURRPRO := READYLIST;
2066     0003A            NEXTREADY := DESCSTORE(.CURRPRO.).RWFLINK;
2067     00042            COUNT := ((CLOCK DIV 10) MOD 10) + 1;
2068     0004D WRITELN(ETRACE,' *** PROCESS #',CURRPRO:3,' ALLOTTED',COUNT:3,' INSTRUCTIONS');
2069     00067            WITH DESCSTORE(.CURRPRO.) DO
2070     00071               WHILE (COUNT > 0) AND (STATUSCODE = READY) DO
2071     0007D                  BEGIN
2072     0007D   WRITELN(ETRACE,' PC = ',PGMCNTR:3,'      OPERATION IS ',
2073     0008E           OPLIST(.PGMSTORE(.PGMCNTR.).OPCODE.));
2074     000A1                  CURRPC := PGMCNTR;
2075     000A4                  PGMCNTR := PGMCNTR + 1;
2076     000A9                  CASE PGMSTORE(.CURRPC.).OPCODE  OF
2077     00087
2078     00087           ADDOP:    BEGIN
2079     00087                        SETSTAR;
2080     00089                        ACC := ACC + DATASTORE(.STAR.)
2081     000C0                     END;
2082     000C4
2083     000C4           SUBOP:    BEGIN
2084     000C4                        SETSTAR;
2085     000C6                        ACC := ACC - DATASTORE(.STAR.)
2086     000CD                     END;
2087     000D1
2088     000D1           RSUBOP:   BEGIN
2089     000D1                        SETSTAR;
2090     000D3                        ACC := DATASTORE(.STAR.) - ACC
2091     00008                     END;
2092     000DE
2093     000DE           MULTOP:   BEGIN
2094     000DE                        SETSTAR;
2095     000E0                        ACC := ACC * DATASTORE(.STAR.)
2096     000E7                     END;
2097     000EC
2098     000EC           DIVOP:    BEGIN
2099     000EC                        SETSTAR;
2100     000EE                        ACC := ACC DIV DATASTORE(.STAR.)
2101     000F5                     END;
2102     000F9
2103     000F9           RDIVOP:   BEGIN
2104     000F9                        SETSTAR;
2105     000F8                        ACC := DATASTORE(.STAR.) DIV ACC
2106     00103                     END;
2107     00106 (*$FO*)
```

```
2108    00106          EQLOP:   BEGIN
2109    00106                      SETSTAR;
2110    00108                      ACC := ORD(ACC = DATASTORE(.STAR.))
2111    00112                   END;
2112    00117
2113    00117          NEQOP:   BEGIN
2114    00117                      SETSTAR;
2115    00119                      ACC := ORD(ACC <> DATASTORE(.STAR.))
2116    00123                   END;
2117    00128
2118    00128          LSSOP:   BEGIN
2119    00128                      SETSTAR;
2120    0012A                      ACC := ORD(ACC < DATASTORE(.STAR.))
2121    00134                   END;
2122    00139
2123    00139          LEQOP:   BEGIN
2124    00139                      SETSTAR;
2125    00138                      ACC := ORD(ACC <= DATASTORE(.STAR.))
2126    00145                   END;
2127    0014A
2128    0014A          GTROP:   BEGIN
2129    0014A                      SETSTAR;
2130    0014C                      ACC := ORD(ACC > DATASTORE(.STAR.))
2131    00156                   END;
2132    00158
2133    0015B          GEQOP:   BEGIN
2134    00158                      SETSTAR;
2135    0015D                      ACC := ORD(ACC >= DATASTORE(.STAR.))
2136    00167                   END;
2137    0016C
2138    0016C          ANDOP:   BEGIN
2139    0016C                      SETSTAR;
2140    0016E                      ACC := ORD((ACC <> 0) AND (DATASTORE(.STAR.) <> 0))
2141    0017E                   END;
2142    00180
2143    00180          OROP:    BEGIN
2144    00180                      SETSTAR;
2145    00182                      ACC := ORD((ACC <> 0) OR (DATASTORE(.STAR.) <> 0 ))
2146    00192                   END;
2147    00194
2148    00194          XOROP:   BEGIN
2149    00194                      SETSTAR;
2150    00196                      ACC := ORD(ACC <> DATASTORE(.STAR.))
2151    001A0                   END;
2152    001A5
2153    001A5          NOTOP:   ACC := ORD(ACC = 0);
2154    001AE (*$F0*)
```

```
2155    001AE          LDXOP:   BEGIN
2156    001AE                      SETSTAR;
2157    001B0                      XREG := DATASTORE(.STAR.)
2158    001B5                   END;
2159    001B9
2160    001B9          STXOP:   BEGIN
2161    001B9                      SETSTAR;
2162    001BB                      DATASTORE(.STAR.) := XREG
2163    001C0                   END;
2164    001C4
2165    001C4          LDAOP:   BEGIN
2166    001C4                      SETSTAR;
2167    001C6                      ACC := DATASTORE(.STAR.)
2168    001CB                   END;
2169    001CF
2170    001CF          STAOP:   BEGIN
2171    001CF                      SETSTAR;
2172    001D1                      DATASTORE(.STAR.) := ACC
2173    00106                   END;
2174    001DA
2175    001DA          STAXOP:  XREG := ACC;
2176    0010F
2177    0010F          LDIOP:   ACC := PGMSTORE(.CURRPC.).CONSTANT;
2178    001E9
2179    001E9          BROP:    PGMCNTR := PGMSTORE(.CURRPC.).ADDRESS;
2180    001F3
2181    001F3          BRFOP:   IF ACC = 0 THEN PGMCNTR := PGMSTORE(.CURRPC.).ADDRESS;
2182    00200 (*$FO*)
```

```
2183    00200          PWAITOP: BEGIN
2184    00200                      SETSTAR;
2185    00202                      DATASTORE(.STAR.) := DATASTORE(.STAR.) - 1;
2186    0020F                      IF DATASTORE(.STAR.) < 0 THEN
2187    00216                         BEGIN
2188    00216                            DELETE(CURRPRO,READYLIST);
2189    0021A                            INSERT(CURRPRO,DATASTORE(.STAR + 1.));
2190    00224                            STATUSCODE := WAITING
2191    00225                         END
2192    00227                   END;
2193    00228
2194    00228          VSIGNALOP:
2195    00228                   BEGIN
2196    00228                      SETSTAR;
2197    0022A                      DATASTORE(.STAR.) := DATASTORE(.STAR.) + 1;
2198    00237                      IF DATASTORE(.STAR.) <= 0 THEN
2199    0023E                         BEGIN
2200    0023E                            TEMP := DATASTORE(.STAR + 1.);
2201    00246*                           DELETE(TEMP,DATASTORE(.STAR + 1.));
2202    00250                            INSERT(TEMP,READYLIST);
2203    00254                            DESCSTORE(.TEMP.).STATUSCODE := READY;
2204    0025E                            IF NEXTREADY = CURRPRO THEN READYLIST := RWFLINK
2205    00262                         END
2206    00264                   END;
2207    00265
2208    00265          READOP:  READLN(ACC);
2209    0026E
2210    0026E          WRITEOP: WRITELN(' ',ACC);
2211    0027C
2212    0027C          DUMPOP:  DUMPDSTORES;
2213    0027F (*$FO*)
```

```
2214  0027F          CREATEOP:
2215  0027F                   WITH PGMSTORE(.CURRPC.) DO
2216  002B9                   BEGIN
2217  00289                      CREATEDESCRIPTOR(INITIALPC,SLINK,OSIZE,PARAMCNT,PARAMSTRT);
221B  00296                      IF NEXTREADY = CURRPRO THEN READYLIST := RWFLINK
2219  0029A                   END;
2220  00290
2221  00290          TERMOP:  BEGIN
2222  00290                      STATUSCODE := TERMINATED;
2223  002A0                      DELETE(CURRPRO,READYLIST);
2224  002A4                      TEMP := CURRPRO;
2225  002A6                      WHILE (DESCSTORE(.TEMP.).STATUSCODE = TERMINATED)
2226  002B1                        AND (DESCSTORE(.TEMP.).DESCENDENTCNT = 0)
2227  002B0                        AND (DESCSTORE(.TEMP.).STATICLINK <> EMPTY) DO
2228  002CE                         BEGIN
2229  002CE                            DEALLOCATE(TEMP);
2230  002D1                            TEMP := DESCSTORE(.TEMP.).STATICLINK;
2231  002D9                            DESCSTORE(.TEMP.).DESCENDENTCNT :=
2232  002E0                              DESCSTORE(.TEMP.).DESCENDENTCNT - 1
2233  002E7                         END;
2234  002EB                      IF (DESCSTORE(.TEMP.).STATUSCODE = TERMINATED) AND
2235  002FB                         (DESCSTORE(.TEMP.).DESCENDENTCNT = 0) AND
2236  00305                         (DESCSTORE(.TEMP.).STATICLINK = EMPTY) THEN
2237  00313                         BEGIN
223B  00313                            WRITELN(' NORMAL TERMINATION.');
2239  0031B                            GOTO 999
2240  0031C                         END
2241  0031C                   END
2242  0031C                   END (*CASE*);
2243  0033C                COUNT := COUNT - 1;
2244  00330             END (*WITH*);
2245  0033E             IF (DESCSTORE(.CURRPRO.).STATUSCODE = READY) OR
2246  0034B                (NEXTREADY <> CURRPRO) THEN
2247  00352                READYLIST := NEXTREADY
224B  00352       END (*WHILE*);
2249  00355       WRITELN(' ABNORMAL TERMINATION--NO PROCESSES READY.');
2250  00350       DUMPSTORES;
2251  0035F 999:  WRITELN(' END OF EXECUTION.')
2252  00366 END (*INTERPRET*);
2253  00393 (*$FO*)
```

```
2254    00393 BEGIN (*DRIVER*)
2255    0039B
2256    0039B   ERRCNT := 0;
2257    0039D   PGMSTADDR := 1;
2258    0039F   RESET(INPUT);
2259    003A1   REWRITE(CTRACE);
2260    003A3   REWRITE(OUTPUT);
2261    003A5   COMPILE;
2262    003A6   IF (ERRCNT = 0) AND (PGMSTADDR > 1) THEN
2263    003B1      BEGIN
2264    003B1         REWRITE(DLIST);
2265    003B3         LISTCODE;
2266    003B4         REWRITE(ETRACE);
2267    003B6         REWRITE(OUTPUT);
2268    003B8         INTERPRET
2269    003B8      END
2270    003B9   ELSE
2271    003BA      WRITELN(' EXECUTION SUPPRESSED.')
2272    003C1 END.
```

***** S U M M A R Y *****
NO ERRORS DETECTED

NONE

TOTAL PROGRAM SIZE IS 9.90 K (9252 WORDS).

*PASCAL: NORMAL END-RUN

```
LOAD (GO),(EF,(:JO,:SYS)),(UNSAT,(PASCALIB)),(MAP),(PERM),(LMN,CONCUR,
TIC:L OF :LIB.PASCALIB ASSOCIATED FOR P@CLOCK
CONTX:L OF :LIB.PASCALIB ASSOCIATED FOR ONLINE
KERID:L OF :LIB.PASCALIB ASSOCIATED FOR P@CLOSE
KERNL:L OF :LIB.PASCALIB ASSOCIATED FOR BIN@HEX
ERRX:L OF :LIB.PASCALIB ASSOCIATED FOR P@ERRXIT
CIO:L OF :LIB.PASCALIB ASSOCIATED FOR P@GET
RDI:L OF :LIB.PASCALIB ASSOCIATED FOR RDI
GETFF:L OF :LIB.PASCALIB ASSOCIATED FOR GETFF
WRC:L OF :LIB.PASCALIB ASSOCIATED FOR WRC
WRI:L OF :LIB.PASCALIB ASSOCIATED FOR WRI
PUTFF:L OF :LIB.PASCALIB ASSOCIATED FOR PUTFF
WRS:L OF :LIB.PASCALIB ASSOCIATED FOR WRS

   * * ALLOCATION SUMMARY * *

PROTECTION       LOCATION   PAGES

DATA (00)          A000       1
PROCEDURE (01)     A400      16
DCB (10)           A200       1
```

```
**********  SIGMA 5/6/7/9  LOAD MODULE MAP  **********
********** ACN= 607JN39M  <ROOT> START= C7D5  **********
**********  LMN= CONCUR          BIAS= A000   **********
**********  SGN= CONCUR          SIZE=011.4K  **********
          "+"=DBLE DEF "-"=LIB DEF "*"=UNUSED DEF "@"=UNUSED LIB DEF

********** PROTECTION TYPES **********
     00 DATA   01 PROCEDURE   10 STATIC
     -------   ------------   ---------
HIGH:  A0FD          CEC1          A3FF
LOW :  A000          A400          A200
SIZE:    FE          2AC2           200

*******  SECT - PROGRAM SECTIONS MAP              ******

A064 CSECT 0
A090 CSECT 0                                                                    -   2C WDS
A21A LDCB  2     A214 0  M:*                                                    -   6E WDS
A244 LDCB  2     A244 0  M:DO                                                   -   2A WDS
A278 LDCB  2     A278 0  M:LO                                                   -   34 WDS
A2AC LDCB  2     A2AC 0  M:SI                                                   -   34 WDS
A2E0 LDCB  2     A2E0 0  M:EI                                                   -   34 WDS
A314 LDCB  2     A314 0  F:00                                                   -   34 WDS
A348 LDCB  2     A348 0  F:01                                                   -   34 WDS
A37C LDCB  2     A37C 0  F:02                                                   -   34 WDS
A40C CSECT 1     A40F 0  ERRDR                                                  -   1E WDS
A42A CSECT 1     A42D 0  GETACHAR                                               -   AE WDS
A4D8 CSECT 1     A4D8 0  GETASYMB                                               -  166 WDS
A63E CSECT 1     A641 0  GEN                                                    -   7E WDS
A68C CSECT 1     A68F 0  TEST                                                   -   4C WDS
A708 CSECT 1     A708 0  POSITION                                               -   2A WDS
A732 CSECT 1     A735 0  ENTER                                                  -   90 WDS
A7C2 CSECT 1     A7C5 0  NCONSTAN                                               -   3A WDS
A7FC CSECT 1     A7FF 0  ICONSTAN                                               -   40 WDS
A83C CSECT 1     A83F 0  CONSTDEC                                               -   62 WDS
A89E CSECT 1     A8A1 0  VARDECLA                                               -   AA WDS
A948 CSECT 1     A948 0  FPARAMET                                               -   56 WDS
A99E CSECT 1     A9A1 0  FPARAMLI                                               -   3E WDS
A9DC CSECT 1     A9DF 0  FIXUP                                                  -   56 WDS
AA32 CSECT 1     AA35 0  PROCDECL                                               -   D4 WDS
AB06 CSECT 1     AB09 0  INVERSEO                                               -   62 WDS
AB68 CSECT 1     AB68 0  NEXTEMP                                                -   26 WDS
AB8E CSECT 1     AB91 0  RELEASET                                               -   10 WDS
AB9E CSECT 1     ABA1 0  VARIABLE                                               -  15A WDS
ACF8 CSECT 1     ACF8 0  FACTOR                                                 -  1FA WDS
AEF2 CSECT 1     AEF5 0  TERM                                                   -   AB WDS
AF9A CSECT 1     AF9D 0  SIMPLEEX                                               -   CC WDS
B066 CSECT 1     B069 0  EXPRESSI                                               -   C2 WDS
B128 CSECT 1     B128 0  PROCESSI                                               -   CC WDS
B1F4 CSECT 1     B1F7 0  ASSIGNME                                               -  112 WDS
B306 CSECT 1     B309 0  DUMPSTAT                                               -   20 WDS
B326 CSECT 1     B329 0  IFSTATEM                                               -  18C WDS
B482 CSECT 1     B485 0  LOOPSTAT                                               -  130 WDS
B5E2 CSECT 1     B5E5 0  PWAITSTA                                               -   9C WDS
B67E CSECT 1     B681 0  VSIGNALS                                               -   9C WDS
B71A CSECT 1     B71D 0  SETSTATE                                               -   F0 WDS
B80A CSECT 1     B80D 0  READSTAT                                               -  150 WDS
B95A CSECT 1     B95D 0  WRITESTA                                               -   66 WDS
B9C0 CSECT 1     B9C3 0  STATEMEN                                               -   D6 WDS
BA96 CSECT 1     BA99 0  SSEQUENC                                               -   2C WDS
BAC2 CSECT 1     BAC5 0  BLOCK                                                  -   96 WDS
BB58 CSECT 1     BB58 0  COMPILE                                                -  264 WDS
BD8C CSECT 1     BD8F 0  LISTCODE                                               -  18C WDS
BF78 CSECT 1     BF78 0  SETSTAR                                                -   50 WDS
BFC8 CSECT 1     BFCB 0  DELETE                                                 -   3C WDS
C004 CSECT 1     C007 0  INSERT                                                 -   4E WDS
C052 CSECT 1     C055 0  DUMPDSTO                                               -  170 WDS
C1C2 CSECT 1     C1C5 0  ALLOCATE                                               -   D0 WDS
C292 CSECT 1     C295 0  DEALLOCA                                               -  120 WDS
C382 CSECT 1     C385 0  CREATEDE                                               -   8C WDS
C43E CSECT 1     C441 0  INTERPRE                                               -  394 WDS
C7D2 CSECT 1     C7D5 0 *COMPILER                                               -   78 WDS
C84A CSECT 1                                                                    -    2 WDS
C84C CSECT 1                                                                    -   16 WDS
C862 CSECT 1                                                                    -   16 WDS
C878 CSECT 1                                                                    -  168 WDS
C9E0 CSECT 1                                                                    -   18 WDS
C9F8 CSECT 1                                                                    -  14E WDS
CB46 CSECT 1                                                                    -   1A WDS
CB60 CSECT 1                                                                    -  16C WDS
                                                                                -    C WDS
```

```
CCDB CSECT 1                                                                    -   A2 WDS
CD7A CSECT 1                                                                    -   94 WDS
CE0E CSECT 1                                                                    -   10 WDS
CE1E CSECT 1                                                                    -   1A WDS
CE38 CSECT 1                                                                    -   4C WDS
CE84 CSECT 1                                                                    -    E WDS
CE92 CSECT 1                                                                    -   2E WDS

*******  RELOCATABLE DEFINITIONS SORTED BY NAME  ******

              C1C5 0  ALLOCATE     B1F7 0  ASSIGNME     BAC5 0  BLOCK        B858 0  COMPILE
              C7D5 0 *COMPILER     A83F 0  CONSTDEC     C385 0  CREATEDE     C295 0  DEALLOCA
              BFCB 0  DELETE       C055 0  DUMPDSTD     B309 0  DUMPSTAT     A735 0  ENTER
              A40F 0  ERROR        B069 0  EXPRESSI     A314 0  F:00         A34B 0  F:01
              A37C 0  F:02         ACFB 0  FACTOR       A9DF 0  FIXUP        A94B 0  FPARAMET
              A9A1 0  FPARAMLI     A641 0  GEN          A42D 0  GETACHAR     A4DB 0  GETASYMB
              A7FF 0  ICONSTAN     B329 0  IFSTATEM     C007 0  INSERT       C441 0  INTERPRE
              A809 0  INVERSED     C79D 0  LAB-0999     8D50 0  LAB-9999     8D8F 0  LISTCODE
              B4B5 0  LODPSTAT     A21A 0  M:*          A244 0  M:DD         A2E0 0  M:EI
              A27B 0  M:LD         A2AC 0  M:SI         A7C5 0  NCONSTAN     A868 0  NEXTEMP
              A708 0  POSITION     AA35 0  PROCDECL     B128 0  PROCESSI     85E5 0  PWAITSTA
              B80D 0  READSTAT     AB91 0  RELEASET     BF7B 0  SETSTAR      B71D 0  SETSTATE
              AF9D 0  SIMPLEEX     BA99 0  SSEQUENC     B9C3 0  STATEMEN     AEF5 0  TERM
              A68F 0  TEST         ABA1 0  VARDECLA     ABA1 0  VARIABLE     B681 0  VSIGNALS
              B950 0  WRITESTA
```

FOD

APPENDIX D

COMPILER OPERATING INSTRUCTIONS

Input to the compiler is read from M:SI. The source listing and execution output are produced on M:LØ. The compiler trace is produced on F:00. The object code is listed on F:01. The execution trace is produced on F:02. To run the compiler, these files must be assigned. Unwanted output can be avoided by assigning the file to NØ.

EXECUTION VIA TIME-SHARING:

1. Assign the files using the SET command. For example:

```
!SET M:SI/EXPROG1    (source code is in a data file called EXPROG1)
!SET M:LØ ME         (listing and execution output will appear on
                      the terminal)
!SET F:00 NØ         (compilation trace suppressed)
!SET F:01 ME         (object code will be listed on the terminal)
!SET F:02 NØ         (execution trace suppressed)
```

2. Run the compiler.

```
!CONCUR.60764DYT
```

3. The compilation will begin with the following message.

```
*PASCAL:  START COMPILER
```

EXECUTION VIA BATCH:

1. Assign the files using the ASSIGN command. For example:

```
!ASSIGN M:SI(FILE,EXPRØG1)        (as above..except that output
                                   will appear on the line printer)
!ASSIGN M:LØ(DEVICE,LP),(VFC)

!ASSIGN F:00(DEVICE,NØ)

!ASSIGN F:01(DEVICE,LP),(VFC)

!ASSIGN F:02(DEVICE,NØ)
```

2. Run the compiler.

```
!RUN(LMN,CONCUR,60764DYT)
```

APPENDIX E

COMPILATION ERRORS

When it detects an error in the source program, the compiler signals the error by skipping to a new line and printing a question mark beneath the offending symbol. The question mark is followed by a number indicating the type of error that has occurred. The compiler counts the errors detected and prints this count at the end of the source listing. Any compiler-detected error is fatal; that is, if the error count is non-zero at the end of compilation, execution is suppressed. The errors which the compiler detects are listed below with their corresponding numbers. A sample program with errors follows.

1. Use = instead of :=?
2. Constant expected.
3. Symbol = expected.
4. Identifier expected.
5. Semicolon expected.
6. Process name expected.
7. Integer expected...parameters must be integer.
8. END expected.
9. Period expected.
10. Variable expected.
11. Undeclared identifier.
12. Assignment to constant, process, or semaphore not allowed.
13. Assignment operator := expected.
14. Arrays must be integer.
15. OF expected.
16. THEN expected.
17. The preceding factor cannot be followed by this symbol.

18. DO or EXIT expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain process identifier.
22. Right parenthesis expected.
23. Left parenthesis expected.
24. An expression cannot begin with this symbol.
25. Semaphore expected.
26. Subscript expected...neither process nor semaphore allowed.
27. Number expected.
28. Left bracket expected.
29. Right bracket expected.
30. Number is too large.
31. Variable must be INTEGER, SEMA, or ARRAY.
32. Colon expected.
33. Block expected...beginning with CONST, VAR, PROCESS, or BEGIN.
34. Comma expected.

10:11 04/28/78 607JN39M 3F-48 [3] SYSTEM UP TILL 5:00PM....$$$ A C E $$$

```
!SET M:SI/EXPROG5.480GMYUL

!SET M:LO ME

!SET F:00 NO

!SET F:01 NO

!SE) F:02 NO

!CONCUR.
 *PASCAL: START COMPILER
```

```
CONCUR COMPILER VERSION 1 JANUARY,1978                          10:11 APR 28,'78
OPTIONS: LS...
   1    !    (* SIMPLE PRODUCER-CONSUMER PROBLEM...PAGE 37 TSICHRITZIS AND BERNSTEIN *)
   2    !
   3    !    (*THIS VERSION INCLUDES ERRORS TO ILLUSTRATE THE ERROR HANDLING
   4    !      OF THE CONCUR COMPILER...*)
   5    !
   6    !
   7    !    CONST NOBUFFERS = 20; EMPTY = 99;
   8    !    VAR   BUFFER: ARRAY NOBUFFERS OF INTEGER;
   9    !          BUFFERPTR: ARRAY NOBUFFERS OF INTEGER;
  10    !          INDEX : INTEGER;  QUEUEBEGIN: INTEGER;  QUEUEND: INTEGER;
  11    !          AVAIL: SEMA;  FULL: SEMA;   MUTEX: SEMA;
  12    !
  13    !    PROCESS PRODUCER;
  14    !      VAR INPUTDATA: INTEGER;
  15    !    BEGIN
  16    !      LOOP
  17    !         READ (INPUTDATA);
  18    !         PWAIT(AVAIL);
  19    !         PWAIT(MUTEX);
  20    !         INDEX := 0;
  21    !         LOOP
  22    !            INDEX := INDEX + 1
  23    !            WHEN BUFFERPTR(.INDEX.) = 0 DO
  24    !               IF QUEUEBEGIN = EMPTY
  25    !                  QUEUEBEGIN := INDEX;
                                       ?17
                                             ?16
  26    !                  QUEUEND := INDEX
  27    !               END;
  28    !               BUFFERPTR(.QUEUEND.) := INDEX;
  29    !               BUFFER(.INDEX.) := INPUTDATA;
  30    !               BUFFERPTR(.INDEX.) := EMPTY;
  31    !               QUEUEND := INDEX
  32    !            EXIT
  33    !         END;
  34    !         VSIGNAL(MUTEX);
  35    !         VSIGNAL(FULL)
  36    !         WHEN INPUTDATA = ENDSIGNAL EXIT
                                         ?11
  37    !      END
  38    !    END PRODUCER;
  39    !
  40    !    PROCESS CONSUMER;
  41    !      VAR OUTPUTDATA: INTEGER;
  42    !    BEGIN
  43    !      LOOP
  44    !         PWAIT(FULL);
  45    !         PWAIT(MUTEX);
  46    !         INDEX = QUEUEBEGIN;
                        ?13
  47    !         OUTPUTDATA := BUFFER(.INDEX.);
  48    !         QUEUEBEGIN := BUFFERPTR(.INDEX.);
```

```
 49     !          BUFFERPTR(.INDEX.) := 0;
 50     !          VSIGNAL(MUTEX);
 51     !          VSIGNAL(AVAIL);
 52     !          WHEN OUTPUTDATA = ENDSIGNAL EXIT
                                              ?11
 53     !          WRITE (OUTPUTDATA)
 54     !        END
 55     !      END CONSUMER;
 56     !
 57     !
 58     !      BEGIN (* MAIN PROGRAM *)
 59     !        SET (AVAIL,NOBUFFERS);
 60     !        FULL := 0;
                     ?12
 61     !        SET (MUTEX,1);
 62     !        INDEX := 0;
 63     !        QUEUEBEGIN := EMPTY;
 64     !        LOOP
 65     !           INDEX := INDEX + 1;
 66     !           BUFFERPTR(.INDEX.) := 0
 67     !           WHEN INDEX = NOBUFFERS EXIT
 68     !        END;
 69     !        PRODUCER;
 70     !        CONSUMER
 71     !      END.
  6 ERRORS FOUND.
EXECUTION SUPPRESSED.
 *PASCAL: NORMAL END-RUN

!
```

APPENDIX F

SOURCE PROGRAM
WITH OBJECT CODE LISTING

```
!COPY EXPROG2.480GMYUL
(*SAMPLE CONCUR PROGRAM USING SEMAPHORE TO MAKE PROCESS LIKE PROCEDURE.*)

CONST ASIZE = 10;  ENDSIG = 9999;

VAR INDATA: INTEGER;
    ARRAYDATA: ARRAY ASIZE OF INTEGER;
    START: INTEGER;  STOP: INTEGER;
    DONE: SEMA;

PROCESS INVERT(START: INTEGER, STOP: INTEGER);
  VAR TEMP: INTEGER;
BEGIN
  LOOP
     WHEN STOP <= START EXIT
     TEMP := ARRAYDATA(.START.);
     ARRAYDATA(.START.) := ARRAYDATA(.STOP.);
     ARRAYDATA(.STOP.) := TEMP;
     START := START + 1;
     STOP := STOP - 1
  END;
  VSIGNAL(DONE)
END INVERT;

BEGIN
  SET(DONE,0);
  START := 1;
  STOP := 0;
  LOOP
     READ(INDATA);
     WHEN INDATA = ENDSIG EXIT
     STOP := STOP + 1;
     ARRAYDATA(.STOP.) := INDATA
  END;
  INVERT(START,STOP);
  PWAIT(DONE);
  LOOP
     WHEN START > STOP EXIT
     WRITE(ARRAYDATA(.START.));
     START := START + 1
  END
END.
1
2
3
4
5
6
7
8
9999

!
```

08:22 04/28/78 607JN39M 16-42 [12] SYSTEM UP TILL 5:00PM....$$$ A C E $$$

```
!SET M:SI/EXPROG?.480GMYUL

!SET M:LO ME

!SET F:00 NO

!SET F:01 ME

!SET F:02 NO

!CONCUR.
 *PASCAL: START COMPILER
```

```
CONCUR COMPILER VERSION 1 JANUARY,1978                    08:22 APR 28,'78
OPTIONS: LS...
   1    !    (*SAMPLE CONCUR PROGRAM USING SEMAPHORE TO MAKE PROCESS LIKE PROCEDURE.*)
   2    !
   3    !    CONST ASIZE = 10;  ENDSIG = 9999;
   4    !
   5    !    VAR INDATA: INTEGER;
   6    !        ARRAYDATA: ARRAY ASIZE OF INTEGER;
   7    !        START: INTEGER;  STOP: INTEGER;
   8    !        DONE: SEMA;
   9    !
  10    !    PROCESS INVERT(START: INTEGER, STOP: INTEGER);
  11    !      VAR TEMP: INTEGER;
  12    !    BEGIN
  13    !      LOOP
  14    !          WHEN STOP <= START EXIT
  15    !          TEMP := ARRAYDATA(.START.);
  16    !          ARRAYDATA(.START.) := ARRAYDATA(.STOP.);
  17    !          ARRAYDATA(.STOP.) := TEMP;
  18    !          START := START + 1;
  19    !          STOP := STOP - 1
  20    !      END;
  21    !      VSIGNAL(DONE)
  22    !    END INVERT;
  23    !
  24    !    BEGIN
  25    !      SET(DONE,0);
  26    !      START := 1;
  27    !      STOP := 0;
  28    !      LOOP
  29    !          READ(INDATA);
  30    !          WHEN INDATA = ENDSIG EXIT
  31    !          STOP := STOP + 1;
  32    !          ARRAYDATA(.STOP.) := INDATA
  33    !      END;
  34    !      INVERT(START,STOP);
  35    !      PWAIT(DONE);
  36    !      LOOP
  37    !          WHEN START > STOP EXIT
  38    !          WRITE(ARRAYDATA(.START.));
  39    !          START := START + 1
  40    !      END
  41    !    END.
 0 ERRORS FOUND.
```

GENERATED CODE

INDEX	OPERATION	OPERANDS		
1	LDA	0	0	1
2	STA	0	0	3
3	LDA	0	0	0
4	GEQ	0	0	3
5	BRF	7		
6	BR	34		
7	LDA	0	0	0
8	STAX			
9	LDA	1	1	1
10	STA	0	0	2
11	LDA	0	0	0
12	STA	0	0	3
13	LDA	0	0	1
14	STAX			
15	LDA	1	1	1
16	LDX	0	0	3
17	STA	1	1	1
18	LDA	0	0	1
19	STA	0	0	3
20	LDA	0	0	2
21	LDX	0	0	3
22	STA	1	1	1
23	LDA	0	0	0
24	STA	0	0	3
25	LDI	1		
26	ADD	0	0	3
27	STA	0	0	0
28	LDA	0	0	1
29	STA	0	0	3
30	LDI	1		
31	RSUB	0	0	3
32	STA	0	0	1
33	BR	1		
34	VSIGNAL	0	1	13
35	TERMINATE			
36	LDI	0		
37	STA	0	0	13
38	LDI	9999		
39	STA	0	0	14
40	LDI	1		
41	STA	0	0	11
42	LDI	0		
43	STA	0	0	12
44	READ			
45	STA	0	0	0
46	LDA	0	0	0
47	STA	0	0	15
48	LDI	9999		
49	EQL	0	0	15
50	BRF	52		

51	BR	63				
52	LDA	0	0	12		
53	STA	0	0	15		
54	LDI	1				
55	ADD	0	0	15		
56	STA	0	0	12		
57	LDA	0	0	12		
58	STA	0	0	15		
59	LDA	0	0	0		
60	LDX	0	0	15		
61	STA	1	0	1		
62	BR -	44				
63	LDA	0	0	11		
64	STA	0	0	15		
65	LDA	0	0	12		
66	STA	0	0	16		
67	CREATE	1	0	4	2	15
68	PWAIT	0	0	13		
69	LDA	0	0	11		
70	STA	0	0	15		
71	LDA	0	0	12		
72	LSS	0	0	15		
73	BRF	75				
74	BR	85				
75	LDA	0	0	11		
76	STAX					
77	LDA	1	0	1		
78	WRITE					
79	LDA	0	0	11		
80	STA	0	0	15		
81	LDI	1				
82	ADD	0	0	15		
83	STA	0	0	11		
84	BR	69				
85	TERMINATE					

```
08:26 04/28/78 607JN391 16-42     [16] SYSTEM UP TILL 5:00PM....$$$ A C E $$$


          8
          7
          6
          5
          4
          3
          2
          1
NORMAL TERMINATION.
END OF EXECUTION.
 *PASCAL: NORMAL END-RUN

!
```

REFERENCES

1. Tsichritzis, D.C., and Bernstein, P.A. Operating Systems, Academic Press. New York, New York, 1974.

2. Brinch Hansen, P. The Architecture of Concurrent Programs, Prentice-Hall. Englewood Cliffs, New Jersey, 1977.

3. Wirth, N. "Towards A Discipline of Real-Time Programming", Proceedings from the conference on Language Design For Reliable Software, 1977.

4. Wirth, N. "Modula: a Language for Modular Multiprogramming", Software -- Practice and Experience, Vol. 7 (January, 1977), pp. 3 - 35.

5. Wirth, N. "The Use of Modula", Software -- Practice and Experience, Vol. 7 (January, 1977), pp. 37 - 65.

6. Wirth, N. "Design and Implementation of Modula", Software -- Practice and Experience, Vol. 7 (January, 1977) pp. 67 - 84.

7. Dijkstra, E. W. "Cooperating Sequential Processes", pp. 43 - 112 in Programming Languages (F. Genuys, editor), Academic Press. New York, New York, 1968.

8. Hoare, C.A.R. "Algorithm 63 Partition", "Algorithm 64 Quicksort", Communications of the ACM, Vol. 4 No. 7 (July, 1961), p. 321

9. Lorin, H., Sorting and Sort Systems, Addison-Wesley, Reading, Mass., 1975.

10. Richards, M., "The Portability of the BCPL Compiler", Software - Practice and Experience, Vol. 1 (1971), pp. 135-146.

11. Waite, W. M., Implementing Software for Non-Numeric Applications, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

12. Knuth, D. E., The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Mass., 1973 (second edition).

13. Bobrow, D. G., and Wegbreit, B. "A Model and Stack Implementation of Multiple Environments", Communications of the ACM, Vol. 16 No. 10 (October, 1973), pp. 591 - 603.

14. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17 No. 10 (October, 1974), pp. 549 - 557.

15. Owicki, S. and Gries, D. "Verifying Properties of Parallel Programs: An Axiomatic Approach", Communications of the ACM, Vol. 19 No. 5 (May, 1976), pp. 279 - 285.

16. Gries, D. "An Exercise in Proving Parallel Programs Correct", Communications of the ACM, Vol. 20 No 12 (December, 1977), pp. 921 - 930.

17. Wirth, N. Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.