1986

Automatic Ordering of Program Units for Execution

Ronald D. Williams
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/gradreports

Part of the Computer Sciences Commons

Recommended Citation
Williams, Ronald D., "Automatic Ordering of Program Units for Execution" (1986). All Graduate Plan B and other Reports. 1169.
https://digitalcommons.usu.edu/gradreports/1169
AUTOMATIC ORDERING OF PROGRAM
UNITS FOR EXECUTION

by

Ronald D. Williams

A report submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
Computer Science

UTAH STATE UNIVERSITY
Logan, Utah
1986
ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Wendell L. Pope, Department of Computer Science, Utah State University, for the help given to me as I have proceeded to obtain my master's degree in computer science. His help and suggestions have been very helpful. I would also like to thank my other committee members, Dr. Donald Cooley, Computer Science, and Dr. David Luthy, Accounting, for their time and help.

A special word of appreciation is extended to my parents, Joseph R. and Helen P. Williams of Ogden, Utah, for the considerable support and encouragement they have given me as I have worked towards this degree.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td><strong>1. INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2. A NON-PROCEDURAL PROGRAMMING SYSTEM</strong></td>
<td>3</td>
</tr>
<tr>
<td>2.1 System Definition</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Completeness Test</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Feasibility Test</td>
<td>4</td>
</tr>
<tr>
<td>2.4 Order Constraints</td>
<td>4</td>
</tr>
<tr>
<td>2.5 Imposed Ordering</td>
<td>5</td>
</tr>
<tr>
<td><strong>3. SEMAPHORES</strong></td>
<td>6</td>
</tr>
<tr>
<td>3.1 Semaphore Definition</td>
<td>6</td>
</tr>
<tr>
<td>3.2 Semaphore Use</td>
<td>9</td>
</tr>
<tr>
<td><strong>4. ORDERING PROGRAM OVERVIEW</strong></td>
<td>12</td>
</tr>
<tr>
<td>4.1 Language and System</td>
<td>12</td>
</tr>
<tr>
<td>4.2 Graph Data Structure</td>
<td>12</td>
</tr>
<tr>
<td>4.3 Graph Construction</td>
<td>14</td>
</tr>
<tr>
<td>4.4 Completeness Test</td>
<td>15</td>
</tr>
<tr>
<td>4.5 Feasibility Test</td>
<td>16</td>
</tr>
<tr>
<td>4.6 Semaphore Placement</td>
<td>17</td>
</tr>
<tr>
<td><strong>5. USE OF THE ORDERING PROGRAM</strong></td>
<td>19</td>
</tr>
<tr>
<td>5.1 Source Program</td>
<td>19</td>
</tr>
<tr>
<td>5.2 Flagging</td>
<td>20</td>
</tr>
<tr>
<td>5.3 Instructions for Use</td>
<td>22</td>
</tr>
<tr>
<td>5.4 Sample Input and Output</td>
<td>24</td>
</tr>
<tr>
<td><strong>6. SINGLE AND MULTIPROCESSOR ENVIRONMENTS</strong></td>
<td>28</td>
</tr>
<tr>
<td>6.1 Single Processor Environment</td>
<td>28</td>
</tr>
<tr>
<td>6.2 Multiprocessor Environment</td>
<td>29</td>
</tr>
<tr>
<td>6.3 Multiprocessor Synchronization</td>
<td>30</td>
</tr>
<tr>
<td>6.4 Conclusion</td>
<td>31</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

REFERENCES ................................................. 33
APPENDIX ...................................................... 34
  Appendix A.  Order Program Listing ...................... 35
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Directed Graph</td>
<td>13</td>
</tr>
<tr>
<td>2.</td>
<td>Graph Data Structure Representation</td>
<td>13</td>
</tr>
<tr>
<td>3.</td>
<td>Pre-existing inputs</td>
<td>24</td>
</tr>
<tr>
<td>4.</td>
<td>Source Program Input</td>
<td>24</td>
</tr>
<tr>
<td>5.</td>
<td>Directed Graph Representation of Source Program</td>
<td>25</td>
</tr>
<tr>
<td>6.</td>
<td>Output with Semaphores Inserted</td>
<td>27</td>
</tr>
<tr>
<td>7.</td>
<td>Dual Processor Execution Times</td>
<td>29</td>
</tr>
</tbody>
</table>
ABSTRACT

Automatic Ordering of Program Units for Execution

by

Ronald D. Williams, Master of Science
Utah State University, 1986

A program written in today's sequential programming languages must be written according to a rule which states that source instructions must be written in their exact order of execution. A better rule would be to let the programmer write the instructions in any order he wants--then let a program figure out the proper order of execution. Such a system applies not only to individual instructions in a procedure or program, but to procedures in a program and to programs in a job stream.

This paper and its associated automatic ordering program introduce a method by which instructions can be written in any order. The ordering program analyzes the source instructions and determines their order of execution. Semaphores are utilized by the ordering program to control the order of execution of the source instructions. Were this system to be used in conjunction with a compiler, the user of such a compiler would no longer be forced to worry as much about the order of his source instructions. Thus, the programmer would be able to
concentrate more on the "what" of programming rather than so much on the "how" of programming. The programmer, then, would be writing programs at a higher level than is possible with current higher level languages.

(57 pages)
1. INTRODUCTION

In the current sequential programming environment, a programmer must pay explicit attention to the order of statements in a procedure, the order of procedures in a program, and/or the order of programs in a job stream. This is necessitated by the fact that the input to a program unit (where a program unit can be a statement in a procedure, a procedure in a program, or a program in a job stream) often must have been previously created by another program unit. In other words, if program A creates file X as output and program B uses file X as input, then program A must finish executing before program B can start executing.

This project is an exercise in implementing an automatic ordering technique which will determine when all inputs have become available to a particular program unit. The potential benefits of automatic ordering of program units extend into two areas. First, the use of automatic program unit ordering allows the programmer to concentrate on programming at a higher level than is possible in the current sequential programming environment. The programmer can concentrate more on the "what" of programming rather than so much on the "how" of programming. Second, the use of automatic program unit ordering allows different program units to execute concurrently on different processors in a multiprocessor environment. The problem of scheduling the program units on two or more processors thus becomes easier.

This project has attempted to address the above issues. Needless to say, more research remains to be done. However, a basic algorithm
to implement automatic program unit ordering is presented in this project.

Definitions of three terms used extensively by this project are given at this point.

Program unit: A program unit consists of a distinct and separable element of a program. Thus, a program unit can theoretically be an individual statement in a procedure, a procedure in a program, or a program in a job stream. The actual syntax of the program units as used by this project is given later in this paper.

Source program: A source program is a program which is analyzed by the automatic ordering program. In following the convention described under the definition of "program units," a source program can theoretically consist of a procedure of statements, a program of procedures, or a job stream of programs. The actual syntax of the source program as used by this project is given later in this paper.

Ordering program: The ordering program is the program which analyzes the source program and automatically determines the order of program unit execution.

2.1 System Definition

A formal definition of the project is given in this section. A less formal and more intuitive description is given in section 4.

In any multiprogramming or multiprocessing environment where a number of programs units are to be executed, the question of their order of execution arises. That is, given a set of program units, how can they be ordered for execution?

Let $Q_1, Q_2, \ldots, Q_n$ be the program units to be ordered. Let $\text{IN}_i$ be the input set for $Q_i$, and $\text{OUT}_i$ be the output set for $Q_i$. That is, $\text{IN}_i = (X_{i1}, X_{i2}, \ldots, X_{ik})$ and $\text{OUT}_i = (Y_{i1}, Y_{i2}, \ldots, Y_{im})$.

The $Q_i$ can be a collection of statements, procedures, or tasks in one program, a collection of interdependent programs, or any combination thereof. The $\text{IN}_i$ and $\text{OUT}_i$ can be sets of files, variables, or parameters.

Assume that the $Q_i$ are all submitted to be run, but the order is unspecified. We classify the $Q_i$ as follows:

1) Any $Q_i$ with a null input set may begin executing.
2) Any $Q_i$ whose input set consists entirely of existing resident files may begin executing.
3) All remaining $Q_i$ must be interdependent, i.e., the output from one must be produced before another can begin.

In order to determine the dependencies of the program units as classified above, we form a directed graph as follows: Search the output sets of all $Q_i$ for inputs for $Q_j$. If an input for $Q_j$ is an output of $Q_i$, then $Q_i$ must precede $Q_j$ and $<i,j>$ is a directed edge in
the graph, and $Q_i$ and $Q_j$ are vertices in the graph. Repeat the search for all $j$.

2.2 Completeness Test

The system is complete and the graph is finished when inputs for all $Q_j$ have been found. Failure to find one or more inputs means the system can not run and the user must be notified to remove the deficiency.

2.3 Feasibility Test

If the graph has a topological ordering of its vertices, then it has no cycles and the system is feasible and can be run. If the system is not feasible, then the cycles must be reported to the user so they can be removed.

2.4 Order Constraints

An ordering of the $Q_i$ is established by the topological ordering described above. To run the system in that order, establish a semaphore, $S_{ij}$, for each directed edge, $<i,j>$, in the graph. Initialize all the $S_{ij}$ to zero. Prefix $Q_j$ with $P(S_{ij})$ and suffix $Q_i$ with $V(S_{ij})$ (the $P$ and $V$ operations are described in detail in section 3). $P(S_{ij})$ causes $Q_j$ to be blocked from executing when $S_{ij} = 0$, and if $S_{ij} = 1$, $Q_j$ passes its $P$ and may begin executing. $V(S_{ij})$ sets $S_{ij}$ equal to one and causes a search to see if $Q_j$ is amongst the set of all blocked program units. If it is, $Q_j$ is moved to the ready state so that it can begin executing. In a multiprocessing environment, any of potentially several program units in the ready state can be selected for execution.
2.5 Imposed Ordering

Several methods suggest themselves for allowing users to establish an order between any pair of program units not ordered by input-output constraints. One method would be for the user to establish an artificial input-output dependency between the two units. Another method would be to allow the user an opportunity to add an edge to the graph after the completeness test is done. Yet another way would be to give the user the capability to define a semaphore, but this has the disadvantage of having to be done after the feasibility test, and may cause the system to be infeasible. The feasibility test would have to be done again. The issue of imposed ordering has not been addressed by this project. It could, however, be addressed at some future time.
3. SEMAPHORES

3.1 Semaphore Definition

Since the mechanism used by this system to control the order of program execution is the semaphore, a definition and description of the semaphore is given in this chapter. The semaphore was introduced by Dijkstra in 1965 and is commonly used as a process control mechanism. [1]

A semaphore is nothing more than an integer variable with access constraints. Other than at initialization, a semaphore can only be accessed by two atomic operations: P and V. The definitions of the P and V operations are:

```plaintext
Procedure P(S_{ij});
begin
A: if FLAG then goto A;
FLAG := true;
if S_{ij} = 0 then BLOCK(Q_j);
FLAG := false;
end;

Procedure V(S_{ij});
begin
A: if FLAG then goto A;
FLAG := true;
S_{ij} := S_{ij} + 1;
if Q_j is blocked then MOVE(Q_j);
FLAG := false;
end;
```

where $S_{ij}$ is a semaphore and $Q_j$ is a program unit. [2]

When semaphores are used, we must consider the problem of mutual exclusion. That is, if a P operation is executing on a given semaphore, we cannot allow a V operation (or visa versa) to begin executing on the same semaphore. The use of the FLAG will prevent this. When a P or V operation is entered, the value of the FLAG is checked. If it
is equal to TRUE, then the corresponding V or P operation on this semaphore is currently executing and the IF statement will continue to execute until the value of FLAG becomes equal to FALSE. Once the FLAG is set to FALSE, the IF statement is passed and the FLAG is once again set to TRUE. Mutual exclusion is now ensured.

Another problem encountered when semaphores are used is that of indivisibility of P and V. That is, if a P (or a V) operation is currently executing, we cannot allow it to be interrupted by another instance of execution of the P (or the V) operation. Such a problem can be handled by disabling the hardware interrupt capability upon entering a P or V, and re-enabling it upon exit. This will ensure that a P (or a V) operation which is executing on a given semaphore will not be interrupted and the integrity of that P (or that V) operation will not be threatened.

The P and V operations manage two queues of program units. P puts program units on the BLOCK queue. V moves programs units from the BLOCK queue to the READY queue. [2] Such control is performed as follows: Qj invokes P(Sij). As the P operation executes, the value of the semaphore $S_{ij}$ is checked. If $S_{ij}$ is equal to zero, then program unit $Q_j$ is placed on the BLOCK queue where it will remain until it is removed by a V operation (to be described shortly) and control given to the CPU scheduler. If $S_{ij}$ is not equal to zero (meaning the associated V operation has already executed), then control returns to $Q_j$ as it has passed its P. After passing all the P's preceding it, $Q_j$ can begin executing. Scheduling the execution of $Q_j$ is given to the CPU scheduler.
When \( Q_i \) finishes executing, all the \( V(S_{ij}) \) following it are invoked. When the \( V \) operation executes, the value of the semaphore \( S_{ij} \) is incremented by one. The BLOCK queue is then searched for \( Q_j \) (which was placed there by a \( P \) operation). If \( Q_j \) is found, then the MOVE operation will transfer \( Q_j \) from the BLOCK queue to the READY queue and control will then be returned to the calling process. We must be careful when searching the BLOCK queue for a particular \( Q_j \) since a given \( Q_j \) can be placed on the BLOCK queue more than once. Such a situation will occur if a \( Q_j \) is preceded by more than one \( P \) operation. Two possible solutions to this problem exist.

The first solution allows a \( Q_j \) to exist on the BLOCK queue more than once, but a flag is associated with each occurrence of \( Q_j \). When a \( P \) operation on an \( S_{ij} \) is executed, the \( Q_j \) is simply placed at the end of the BLOCK queue. Then, when searching the BLOCK queue for a \( Q_j \) we must see if it exists on the queue more than once and if so, we can not move \( Q_j \) to the READY queue until all instances of \( Q_j \) on the BLOCK queue have been given authority to execute. Such authority can be given by setting the flag on each instance of \( Q_j \) when that instance has been given authority to execute. Once all instances of \( Q_j \) have had their flag set, then \( Q_j \) can be moved to the READY queue and all \( Q_j \) on the BLOCK queue can be removed.

The second and probably better solution is to associate a counter with each \( Q_j \). Each time a \( P \) operation on an \( S_{ij} \) is executed, a search is made for \( Q_j \) on the BLOCK queue. If \( Q_j \) is found, its counter is incremented by one. If \( Q_j \) is not found, it is added to the BLOCK queue and its counter is set to one. Then, each time a \( V \) operation
is executed on $S_{ij}$, $Q_j$ is found on the BLOCK queue and its counter is decremented by one. When the counter reaches zero, $Q_j$ is ready to execute and can be moved to the READY queue.

As already mentioned, the second solution is probably the better solution. By allowing only one occurrence of $Q_j$ to exist on the BLOCK queue, the system is both easier to comprehend and easier to implement. The searching algorithm is easier since only one occurrence of $Q_j$ has to be found on the BLOCK queue. Another advantage is that the length of the queue is shorter. With the first solution, the maximum length of the queue is equal to $n$ where $n$ is the number of semaphores in a source program. With the second solution, the maximum length of the queue is equal to $m$ where $m$ is the number of program units in a source program.

We note here that the $P$ and $V$ operations must interface with the operating system. Such an interface can not be covered by the limited scope of this project. However, the implementor of such a system must be aware that such considerations must be taken into account.

3.2 Semaphore Use

Assume that all the $Q_i$ are compilable program units in a given language. All $Q_i$ will be embedded in a source program of the same language. The $S_{ij}$ are declared as global variables and are initialized to zero. Each $Q_i$ is preceded by the appropriate calls to the $P$ operation, and suffixed by the appropriate calls to the $V$ operation. [2]

A specific example to show how semaphores work will now be given. Three semaphores, $S_{12}$, $S_{13}$, and $S_{23}$ are initialized to the integer
value zero. The calls to the P and V operations are then inserted into a sample source program with program units Q1, Q2, and Q3 as shown.

\[
\begin{align*}
S_{12} &:= 0; \\
S_{13} &:= 0; \\
S_{23} &:= 0; \\
Q_1 &\text{ V(S}_{12}\text{) V(S}_{13}\text{)} \\
P(S_{12}) &\text{ Q2 V(S}_{23}\text{)} \\
P(S_{13}) P(S_{23}) &\text{ Q3.}
\end{align*}
\]

The inputs and outputs to these program units have been intentionally left off in order to simplify this illustration.

In the traditional procedural processing environment, Q1 would execute first, Q2 would execute next, and finally, Q3 would execute. The programmer would have to explicitly order these program units. In a non-procedural environment, the program units can theoretically be listed in any order such as Q3, Q2, Q1. Also, in a multiprocessor environment, some or all the program units can execute concurrently depending on which program units are dependent on the completion of other program units. These problems (non-procedurality and concurrency) will be controlled by P and V operations on semaphores.

Since program unit Q1 does not have a P operation preceding it, execution of Q1 can begin immediately. Program unit Q2 has a P operation on semaphore S_{12} preceding it. Since S_{12} was initialized to zero, Q2 is placed on the BLOCK queue and must wait there until the V operation on semaphore S_{12} following Q1 is executed. Only after the V operation following Q1 is executed can Q2 can begin executing. Likewise, since Q3 has P operations on semaphores S_{13} and S_{23} preceding it, Q3 will be placed on the BLOCK queue and will remain there until the V operations on semaphores S_{13} and S_{23} following both Q1 and
Q2 are executed. In other words, Q2 is dependent on the completion of Q1, and Q3 is dependent on the completion of both Q1 and Q2.

As has been shown, the use of semaphores is a convenient way to control the order of program execution. This convenience, however, is totally dependent on the way in which semaphores are inserted into the source program. If they are inserted manually by the programmer, then no advantage is gained by their use since the programmer is still explicitly defining the order in which program units can execute. The benefits are gained when program unit ordering is done by an automatic ordering technique and the semaphores are inserted automatically into the source program.
4. ORDERING PROGRAM OVERVIEW

We have thus far discussed the needs and reasons for implementing an automatic ordering technique of program units. We have also discussed the use of semaphores as a mechanism for controlling the execution of these program units. We shall now consider the program used to perform the ordering of the program units.

4.1 Language and System

The ordering program is written in Pascal on a VAX 11/780 running VMS/4.2 operating system.

4.2 Graph Data Structure

The main data structure used by the ordering program is a directed graph. A set of program units with inputs and outputs such as

\[
\begin{array}{cccc}
Q1 & Y1 & Y2 & \\
X1 & Q2 & Y3 & Y4 \\
X2 & Q3 & Y5 & \\
X3 & Q4 & Y6 & \\
X4 & Q5 & Y7 & \\
X5 & Q6 & Y8 & \\
X6 & X7 & X8 & Q7
\end{array}
\]

can be represented by a directed graph such as shown in figure 1 at the top of page 13. The directed graph of figure 1 can then be implemented as a data structure as shown in figure 2 also on page 13.
The graph data structure consists of a table of length $n$ where $n$ is the number of vertices in the graph. Each program unit in a source program corresponds directly to a vertex in the graph. In our ordering program, $n$ (or the size of the table) has been set equal to 100 since we do not know in advance the number of program units in a source program. The table contains four columns. DESPTR is a pointer to a description or name of the vertex. The name of the vertex is the program unit. PREDCOUNT shows how many predecessors a vertex has. TAG is a boolean value which will be used during topological sorting.
and cycle reporting. SUCCLIST is a pointer to a list containing each immediate successor of a vertex. The successor list can be of infinite length.

4.3 Graph Construction

Before the graph can be constructed, the source program must first be analyzed by a lexical analyzer. The lexical analyzer will examine the source program and find each input of a program unit, each output of a program unit, and the name of the program unit. The inputs are placed in an input list structure in the same order in which they are found in the source program, and the outputs are placed in an output list structure in the same order in which they are found in the source program. The program unit names are placed directly in the graph data structure in the same order in which they are found in the source program. The inputs and outputs can now be easily compared as will be described below. In addition to the input and output lists just described, a list is also created which contains all pre-existing inputs. Once the input, output, and pre-existing input lists are constructed, the relationships between the inputs and the outputs of the program units can be analyzed.

One of three conditions will exist if the source program is written correctly. (1) If a program unit does not have any inputs other than screen input or real-time input, then this program unit is not dependent on any other program unit in the source program. This program unit can be inserted into the graph with its PREDCOUNT set equal to zero. (2) If a program unit has inputs that are all pre-existing, such as resident files, then this program unit is not
dependent on any other program unit in the source program. This program unit can also be added to the graph with its PREDCOUNT set equal to zero. (3) If a program unit \( Q_j \) has inputs which are created as the outputs of another program unit \( Q_i \), then program unit \( Q_j \) is dependent on program unit \( Q_i \) since \( Q_i \)'s output is used as input by \( Q_j \). Since there is a dependency between these two program units, an edge is inserted into the graph. This edge consists of a successor record being linked to the SUCCLIST list of graph record \( i \) (since \( i \) is the predecessor). This successor record contains the value \( j \). Since \( j \) now has a predecessor, the PREDCOUNT field of \( j \) is incremented by one. Since one program unit can produce more than one output, and since another program unit can use one or more of these outputs as its inputs, care must be taken to ensure that an edge between two vertices is not created more than once. In other words, if an edge between two vertices already exists, then an attempt to insert another identical edge into the graph will be flagged. If the edge already exists, the identical edge will not be created.

4.4 Completeness Test

At the same time the graph is being constructed, the completeness test is being performed. The completeness test fails if a program unit uses an input, but this input is neither an output from another program unit nor a pre-existing input. If a completeness error is found, a message indicating the error and the location in the source program where the error occurred is reported. Construction of the graph continues with the next input. However, processing will terminate after all inputs in the source file have been checked for completeness.
4.5 Feasibility Test

Once the graph is constructed, the feasibility test is performed. The feasibility test consists of performing a topological sort of the graph. If the sort is successful, then no cycles exist in the source program and the source program is feasible. If the topological sort fails, then at least one cycle exists in the source program and the system is not feasible. A cycle reporting routine is then called and the first cycle encountered is reported. Should more than one cycle exist, only the first cycle encountered is reported. The cycle must be removed and the graph rebuilt. If another cycle exists, it will be reported at this time.

Topological sorting is performed by first checking the value of each vertex's PREDCOUNT. If the value is equal to zero, then this vertex has no predecessors and can be ordered. This vertex is inserted into an eligibility queue. Vertices are removed from the eligibility queue one at a time and are, at that time, considered ordered. After a vertex is removed from the eligibility queue, the PREDCOUNT of this vertex's successors can be decremented by one. When a vertex's PREDCOUNT reaches zero, it is eligible for ordering and is inserted into the eligibility queue. This process continues until (1) all PREDCOUNTs become equal to zero in which case no cycles exist, or (2) until there are some PREDCOUNTs not equal to zero, and, at the same time, no vertices exist in the eligibility queue in which case at least one cycle exists. Cycle reporting is performed only when the topological sort routine determines that a cycle exists in the graph. If a cycle exists, the vertex on which the topological sort halted is
the first vertex in the cycle. All successors of this vertex are examined to determine if they are involved in the cycle. If they are, the TAG field associated with this vertex is set to TRUE. When all vertices involved in the cycle are found, the cycle is reported. The cycle is reported by listing the first vertex in the cycle, all subsequent vertices in the cycle, and finally, to complete the cycle, the first vertex in the cycle is listed again. [3]

4.6 Semaphore Placement

When the graph has been found to contain no cycles, the feasibility test is successful and the semaphores can be inserted into the source program. This is a very simple matter. Each vertex of the graph is examined for its successors. The edge between a predecessor and its successor represents a precedence relation. A semaphore is generated to represent the edge. The program unit representing the predecessor vertex in the graph and the program unit representing the successor vertex in the graph are then found in the source program. A P operation on the semaphore is inserted before the successor program unit and a V operation on the semaphore is inserted after the predecessor program unit.

A semaphore is generated by prefixing an "S" to two three-digit integers. The "S" is used to satisfy the requirement that identifier names begin with a character. The first three-digit integer following the "S" represents the location (or line number) in the source program where the predecessor program unit is found. The second three digit integer following the "S" represents the location (or line number) in the source program where the successor program unit is found.
When all semaphores have been inserted, the original source program is replaced by the new source program. The new source program can now be used in conjunction with a compiler which implements semaphores. The actual interface between the ordering program and a compiler is not addressed by this project.
5. USE OF THE ORDERING PROGRAM

5.1 Source Program

To use the ordering program, a set of compilable program units together with the inputs and outputs for each of the program units, is submitted to the ordering program. Any name can be given to the set of program units. An example of such a set is

X1 X2  Q1  Y3 Y4
X3  Q2  Y5
Q3  Y6
X4 X6  Q4  Y7

where X1 and X2 are inputs for program unit Q1, and Y3 and Y4 are outputs for program unit Q1. Since the ordering program will automatically order the program units in the correct order of execution, the order in which inputs and outputs for a given program unit appears is immaterial, except that the inputs and outputs for a given program unit must be associated with that program unit by appearing on the same line. In other words, inputs and outputs can precede the program unit, the program unit can precede the inputs and outputs, and so forth. In our example, inputs precede, and outputs follow the program unit. Such an order is not mandatory, but is done here for purposes of illustration. Syntax rules are as follows: A line of the source program may begin in any column of a source record. One or more blanks can separate entries in a source record. The maximum line length is 80 characters. All inputs must begin with the letter "X". An integer value must follow the "X". All program unit names must begin with the letter "Q". Up to 49 characters can follow the "Q". All outputs must begin with the letter "Y". An integer value must follow the "Y". If
an "X" and a "Y" are both followed by the same integer value, then this input and output represent the same data and an edge will be created in the graph to indicate the dependency between these associated program units. The inputs and outputs are optional, but a program unit name is mandatory. An error message will be generated if one does not exist on every line of the source program. The "X"'s, "Y"'s, and "Q"'s can be either upper or lower case letters. A source program can be up to 100 lines in length.

In addition to the source program, a file which contains pre-existing inputs must be created. As with the source program, the names of pre-existing inputs must begin with an "X" and be followed by an integer. At least one space must separate inputs. Input records are a maximum of 80 characters in length.

We should note here that this project is an exercise in the ordering of program units and is not an exercise in parsing. While the ordering program does perform some syntax checking, the amount done is minimal and the user must be aware that his input needs to be syntactically correct.

5.2 Flagging

As discussed in the previous section, each input, each output, and each program unit name is specifically flagged as such. That is, every input must have an "X" preceding it, every output must have a "Y" preceding it, and every program unit name must have a "Q" preceding it. When the lexical analyzer examines the source code, it specifically looks for an "X", a "Y", or a "Q". When it finds one of these three characters, the lexical analyzer knows it has found either an
input, an output, or a program unit name respectively. After finding an identifying flag, the lexical analyzer looks for the remaining characters associated with the just found flag. Blanks delimit the inputs, outputs, and program unit names.

This method of lexical analysis has costs associated with it. These costs come from two areas. (1) The space needed in the source program file to store each of the flags. (2) The time required by the lexical analyzer (and hence the CPU) to search for these flags. However, such a method of lexical analysis does have its benefits. Locations of inputs, outputs, and program unit names can be arbitrary. In other words, outputs can precede inputs, program unit names can follow inputs and outputs, and so forth. This does, in fact, raise the level of programming to a higher level (restrained only by the fact that inputs and outputs must remain on the same line of the source program as their associated program units).

If the costs associated with this method of lexical analysis are considered to be too high, an alternative method does exist for performing the lexical analysis. We can place all inputs into an input set and all outputs into an output set. Such sets can be specified as \( \text{IN}_i = (X_{i1}, X_{i2}, \ldots, X_{ik}) \) for the input set and \( \text{OUT}_i = (Y_{i1}, Y_{i2}, \ldots, Y_{im}) \) for the output set. We can now require that all inputs (or the input set) precede the program unit name, and all outputs (or the output set) follow the program unit name. Then, when we perform the lexical analysis, we will begin looking only for inputs and will continue to do so until we find a program unit name. The individual inputs will, in this case, be identified only by the
fact that a delimiter will precede and follow each input. No flags will be associated with the input. Once the program unit name is found, we will then look only for outputs and will continue to do so until we find the end of the source line. As with the inputs, the outputs will be identified only by a delimiter preceding and following each output.

We now have a simpler lexical analyzer. We also require less secondary storage to store the source program and we spend less CPU time to perform the lexical analysis. We have, however, lost the flexibility associated with the flagging method. Which method to use can be determined only by the intended use of the ordering program. We note that this project uses the first method just described—the flagging method. Such a decision was made for reasons of simplicity and flexibility. By requiring all inputs, outputs, and program unit names to be flagged as such, recognizing which of the three constructs (input, output, or program unit name) we have found becomes much easier. In using the first method, we do not claim it is any better than the second method—only easier and more flexible. Each method does, however, have the advantages and disadvantages just mentioned. Which ones are given the most weight is a decision which must be made by the implementors of a program such as our ordering program.

5.3 Instructions for Use

Once the input file has been created, the ordering program can be run. Entering "R ORDER" will invoke the ordering program. The following prompt will appear: "Program Source File:". The name of the source program file is entered. The following prompt will then
appear: "Pre-existing Parameters File:". The name of the pre-existing inputs file is entered. If a pre-existing inputs file does not exist for a given source program, the ENTER key can simply be pressed without entering the name of a file. The format of these files is given in the next part of this section. The ordering program will now run to completion. When it finishes, a message will be displayed indicating how many program units were processed.

The following conditions will cause errors—each of which will cause an error message to be generated. (1) A syntax error in the source program or in the pre-existing inputs file will cause an error message to be generated. All syntax errors found in the entire source program will be displayed. Remember, however, that only minimal syntax checking is performed. (2) A completeness error, as discussed in sections 2 and 4, will cause an error message to be generated. All completeness errors will be displayed for an entire source program. (3) A feasibility error, as discussed in sections 2 and 4, will cause an error message to be generated. Only the first cycle encountered will be reported. This cycle must be removed and the ordering program run again to check for additional cycles. (4) Output errors will be reported. If the insertion of semaphores into a source program line causes that line to exceed 132 characters, an error message will be generated and the output file will not be created.

When the ordering program successfully completes execution, an output file containing the inserted semaphores is created. The file will have the same name as the input file with an extension of ".SEM".
5.4 Sample Input and Output

A sample input file consisting of 26 program units has been generated. A pre-existing inputs file also was generated and is shown in figure 3.

x1 x2 x3 x4 x5 x6 x7

Figure 3. Pre-existing Inputs

The source program input file is shown in figure 4. As described above, the inputs begin with an "X", the outputs begin with a "Y", and the program unit names begin with a "Q".

```
Q1 y8 y9
x2 x3 Q2 y10
x4 x5 x6 Q3 y11 y12 y13
x7 Q4 y14
x8 x10 Q5 y15 y16
x8 x9 Q6 y17
x10 x11 x12 Q7 y18
x14 Q8 y19 y20
x13 x14 Q9 y21 y22 y23 y24
x15 Q10 y25
x17 x18 Q11 y28
x19 x21 x22 x23 x24 Q12 y29 y30
x16 x25 Q13 y26 y27
x28 x26 x27 Q14 y41 y42
x29 Q15 y31
x30 Q16 y32
x32 Q17 y35 y36
x32 Q18 y37
x31 x28 Q19 y33
x31 Q20 y34
x33 Q21 y38
x25 x34 Q22 y39
x35 x36 x37 Q23 y40
x38 x41 Q24 y43
x42 x43 Q25 y44 y45 y46 y47
x39 x40 Q26 y48
```

Figure 4. Source Program Input
The ordering program is run, the name of the source program file and the name of the pre-existing inputs file are entered when their respective prompts are displayed. A graph representing the source program is shown in figure 5.

Figure 5. Directed Graph Representation of Source Program
The output produced by the ordering program is shown in figure 6. Note that the semaphores are initialized to zero at the beginning of the program. Also, the insertion of the semaphores has drastically altered the format of the original source program. While this is not necessarily appealing to the eye, the context of the source program has not been altered. One final note, the order of execution of the program units is given at the bottom of the listing. This order of execution applies to both a single processor environment and to a multiprocessor environment. Comparison of these two environments will be made in section 6.

S001005 := 0;
S001006 := 0;
S002005 := 0;
S002007 := 0;
S003007 := 0;
S003009 := 0;
S004008 := 0;
S004009 := 0;
S005010 := 0;
S005013 := 0;
S006011 := 0;
S007011 := 0;
S008012 := 0;
S009012 := 0;
S010013 := 0;
S010022 := 0;
S011014 := 0;
S011019 := 0;
S012015 := 0;
S012016 := 0;
S013014 := 0;
S014024 := 0;
S014025 := 0;
S015019 := 0;
S015020 := 0;
S016017 := 0;
S016018 := 0;
S017023 := 0;

Figure 6. Output with Semaphores Inserted
(continued on next page)
S018023 := 0;
S019021 := 0;
S020022 := 0;
S021024 := 0;
S022026 := 0;
S023026 := 0;
S024025 := 0;
Q1 y8 y9 V(S001005) V(S001006)
x2 x3 Q2 y10 V(S002005) V(S002007)
x4 x5 x6 Q3 y11 y12 y13 V(S003007) V(S003009)
x7 Q4 y14 V(S004008) V(S004009)
P(S002005) P(S001005) x8 x10 Q5 y15 y16 V(S005010)
V(S005013)
P(S001006) x8 x9 Q6 y17 V(S006011)
P(S003007) P(S002007) x10 x11 x12 Q7 y18 V(S007011)
P(S004008) x14 Q8 y19 y20 V(S008012)
P(S004009) P(S003009) x13 x14 Q9 y21 y22 y23 y24 V(S009012)
P(S005010) x15 Q10 y25 V(S010013) V(S010022)
P(S007011) P(S006011) x17 x18 Q11 y28 V(S011014) V(S011019)
P(S009012) P(S008012) x19 x21 x22 x23 x24 Q12 y29 y30 V(S012015)
V(S012016)
P(S010013) P(S005013) x16 x25 Q13 y26 y27 V(S013014)
P(S013014) P(S011014) x28 x26 x27 Q14 y41 y42 V(S014024)
V(S014025)
P(S012015) x29 Q15 y31 V(S015019) V(S015020)
P(S012016) x30 Q16 y32 V(S016017) V(S016018)
P(S016017) x32 Q17 y35 y36 V(S017023)
P(S016018) x32 Q18 y37 V(S018023)
P(S015019) P(S011019) x31 x28 Q19 y33 V(S019021)
P(S015020) x31 Q20 y34 V(S020022)
P(S019021) x33 Q21 y38 V(S021024)
P(S020022) P(S010022) x25 x34 Q22 y39 V(S022026)
P(S018023) P(S017023) x35 x36 x37 Q23 y40 V(S023026)
P(S021024) P(S014024) x38 x41 Q24 y43 V(S024025)
P(S024025) P(S014025) x42 x43 Q25 y44 y45 y46 y47
P(S023026) P(S022026) x39 x40 Q26 y48

Order of program execution:
Q1 Q2 Q3 Q4 Q6 Q5 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q15 Q16
Q14 Q19 Q20 Q17 Q18 Q22 Q23 Q24 Q26 Q25

Figure 6. Output with Semaphores Inserted (cont.)
6. SINGLE AND MULTIPROCESSOR ENVIRONMENTS

A comparison of the ordering of program units in both a single processor and a multiprocessor environment shall be considered in this section.

6.1 Single Processor Environment

The main benefit of automatic program unit ordering in a single processor environment is the relief the programmer will have of the worry of arranging the program units in the correct execution order. In the end, the result of automatic ordering and manual ordering are the same. However, the increase in productivity of performing the automatic method over the manual method is potentially quite different. We shall now consider the execution time for the program units in a single processor environment. For purposes of illustration, arbitrary execution times have been assigned to each program unit in the sample program initially considered in section 5. These execution times are contained in parenthesis following each program unit. Note that these program units are listed in the same order in which they will be executed:

Q1(6)  Q2(5)  Q3(10)  Q4(4)  Q6(4)  Q5(5)  Q7(10)  Q8(8)
Q9(5)  Q10(9)  Q11(5)  Q12(4)  Q13(8)  Q15(6)  Q16(4)  Q14(5)
Q19(7)  Q20(6)  Q17(7)  Q18(5)  Q21(8)  Q22(7)  Q23(5)  Q24(10)
Q26(6)  Q25(9).

Total execution time is derived simply by adding the execution times of each individual program unit. Given the sample execution times, the total execution time for all the program units is 168 time units.
6.2 Multiprocessor Environment

The benefits of automatic program unit ordering in a multiprocessor environment are similar to the benefits of the single processor environment. There are, however, some additional benefits. As with the single processor system, the programmer does not need to worry about the order in which he places the program units. Another benefit of automatic program unit ordering is that the ordering of programs, including the placement of semaphores in the source code, also determines which programs can execute concurrently. In a multiprocessor environment, this benefit would aid greatly in the scheduling process. Using the same individual program unit execution times as used in the previous section, figure 7 shows the total execution time for all program units on a two processor system.

Processor #1

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q8</th>
<th>Q10</th>
<th>Q13</th>
<th>Q14</th>
<th>Q17</th>
<th>Q20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>19</td>
<td>27</td>
<td>36</td>
<td>44</td>
<td>49</td>
<td>56</td>
</tr>
<tr>
<td>0</td>
<td>22</td>
<td>023</td>
<td>026</td>
<td>62</td>
<td>69</td>
<td>74</td>
<td>80</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Processor #2

<table>
<thead>
<tr>
<th>Q2</th>
<th>Q3</th>
<th>Q7</th>
<th>Q9</th>
<th>Q11</th>
<th>Q12</th>
<th>Q15</th>
<th>Q16</th>
<th>Q18</th>
<th>Q19</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>15</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>39</td>
<td>45</td>
<td>49</td>
<td>54</td>
</tr>
<tr>
<td>0</td>
<td>21</td>
<td>024</td>
<td>025</td>
<td>61</td>
<td>69</td>
<td>79</td>
<td>88</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7. Dual Processor Execution Times

As can be seen, total execution time in a two processor environment is 88 time units. This is just slightly more than half of the 168 time units required on the single processor system. Needless to say,
the reduced execution time is a great benefit to the user. The automatic ordering of the program units just compounds the benefits. Although not addressed here, similar benefits can also be obtained on a multiprocessor system where more than two processors exist.

6.3 Multiprocessor Synchronization

One significant problem exists when using semaphores in a multiprocessor environment. This problem is the synchronization of the different processors. For example, if two processors begin executing a P operation at the same time, then the integrity of the P operation is threatened. Likewise, if one processor begins executing a P operation on a given semaphore and another processor begins executing a V operation on the same semaphore, once again the integrity of the operation (P or V) is threatened. These problems are the same problems of mutual exclusion and indivisibility considered in 3.1, only now, the use of multiple processors complicates the problem even more. Steps must be taken to guard against such loss of integrity.

Synchronization of processors will alleviate this integrity problem. Such synchronization must be performed at the operating system level since it is at this level that the machines actually communicate with each other. One method of synchronization involves the use of a coordinator process residing on one of the processors. The coordinator will ensure that mutual exclusion and indivisibility exists among critical processes. In our case, the critical processes are the P and the V operations. When a process (P or V) wants to invoke mutual exclusion or indivisibility, it sends a request message to the coordinator. When the process receives a reply message from the coordi-
nator, it can begin executing. After the process is finished executing, it sends a release message back to the coordinator.

When a request message is received, the coordinator checks to see if some other critical process is executing. If no such process is executing, then the coordinator sends a reply message back to the requesting process. Otherwise, the request is queued and will later be serviced when a release message is received from some other process. [1]

Other methods of multiprocessor synchronization exist. Such methods include the distributed approach and the token passing approach. See [1] for additional information on these two methods.

6.4 Conclusion

We have attempted, through the implementation of this project, to demonstrate that the automatic ordering of program units is not only feasible from a technological standpoint, but is also desirable from a programmer efficiency standpoint. In a single processor environment, the programmer becomes more efficient since he is not required to worry about the order in which he places program units. In a multiprocessor environment, the programmer becomes even more efficient since he does not have to worry about either program unit ordering nor program unit synchronization. The synchronization of programs in a multiprocessor system offers, by far, the most promising future for automatic program unit ordering.

We should note that there are costs associated with automatic program unit ordering. These costs include the additional overhead of the ordering algorithms which in turn increases computer time used, the
additional complexity of compiling a program (program unit ordering would be closely associated with the compilation process), and the reduced control the programmer will have over his style and method of programming. Historically, when new assemblers and compilers have been introduced, these same questions of increased cost and computer inefficiencies have been raised. Even so, the introduction of these assemblers and compilers have resulted in benefits which have far outweighed any cost and computer inefficiency considerations. Although automatic program unit ordering does have additional costs associated with it, additional research into the subject would surely provide the information needed to efficiently implement such a system.
REFERENCES


Appendix A. Order Program Listing

program order(input,output);

(******************************************************************************)
(*)
(* Program name: ORDER *)
(*)
(* This program performs automatic ordering of program units for execution *)
(* We can see that P1 must execute before P2. P3 can execute any time *)
(* any other program's output. Therefore, P1 and P2 can execute concurrently *)
(* currently with P3. This program, given input like that shown above, *)
(* will calculate which program units can execute concurrently. *)
(* Semaphores will be used to control the scheduling of the programs. *)
(*)
(* The following modules exist in this system: *)
(* LEXICAL: Reads input parameters, output parameters, and *)
(* enters them into in_list and out_list linked *)
(* lists, respectively. The program names are also *)
(* read and are entered directly into the graph *)
(* data structure (GRAPH_REC). This module also *)
(* reads the names of parameters that are pre-existing *)
(* parameters and stores them in a linked list called *)
(* PRE_LIST. *)
(* BUILD: This module takes the input parameter linked list *)
(* where the pre-existing parameters linked list (pre_list), *)
(* and builds the graph data structure (graph_rec) using *)
(* the above mentioned linked lists. *)
(* CYCLES: This module takes the graph data structure (graph_rec) *)
(* and determines if any cycles exist in the graph. If one *)
(* or more cycles do exist, a message is sent to the user *)
(* indicating such and the nodes of the graph in the cycle *)
(* are then reported to the user. *)
(* SEMAS: This module takes the graph record and inserts semaphores *)
(* between the proper programs. This is done by inserting *)
(* a "P" operation before of successor programs and *)
(* inserting a "V" operation after predecessor programs. *)
(* The output file is then written containing the new program with the semaphores inserted. *)
(*)
(******************************************************************************)

type
descrptor = varying[50] of char; (* Descrip. of a row of GRAPH_REC *)
descrptor_pointer = descrptor; (* Pointer to DESCRIPTOR *)
node_pointer = ^node_record; (* Points to nodes of the graph *)
node_record = record (* Defines a node of graph *)
  value: integer; (* Designates which node in graph *)
  next: node_pointer; (* Pointer to next node *)
end;

graph_record = record (* Defines structure of graph *)
  desptr: desc_pointer; (* Points to descrip. of graph *)
  precount: integer; (* No. of predecessors of node *)
  tag: boolean; (* Used in cycle detection *)
  succlist: node_pointer; (* List of successors of node *)
end;

record_132 = packed array[l .. 132] of char; (* 80 character record *)
top_sort_str = varying[50] of char; (* save the topological sort *)

var
  description: [global] desc_pointer; (* Desc. of a row of GRAPH_REC *)
  graph_rec: [global] array[l .. 100] of graph_record; (* Main graph *)
  pre, source, out: [global] text;
  prefile,
  infile, outfile: [global] varying[25] of char; (* holds file names *)
  lex_error: [global] boolean; (* True if syntax errors exit *)
  in_list, out_list: [global] array[l .. 100] of node_pointer;
  test, pre_list: [global] node_pointer; (* Input, output params *)
  p: [global] integer; (* Pre-existing parameters *)
  record_hold: [global] array[l .. 100] of record_132;(* Holding for input recs*)
  total_rows: [global] integer; (* Total number of program rows *)
  cycles_exist: [global] boolean; (* True is cycles found in graph *)
  top_sort_save: [global] array[l .. 100] of top_sort_str;
  a: char; (* save the topological sort *)

procedure build_lists; extern;
procedure pre_exist; extern;
procedure build_graph; extern;
procedure top_sort; extern;
procedure report_cycles; extern;
procedure insert_semaphores; extern;

begin (* main program *)
  write('Program Source File: '). (* read the file name character *)
  while (a <> '.') and (not eoln(input)) do (* by character and place the *)
    begin (* name into the input file name*)
      read(a);
      infile := infile + a;
      outfile := outfile + a;
    end;
  if (eoln(input)) and (a <> '.') then (* add extension to output file *)
    begin
      outfile := outfile + '.sem'
    end
  else
    begin (* add extension to output file *)
      outfile := outfile + 'sem';
    end;
end;
while not eoln(input) do
  begin
    read(a);
    infile := infile + a;
  end;
end;
open(source,infile,history := old);
reset(source);
read(a);
write('Pre-existing Parameters File: ');
readln(prefile);
if length(prefile) <> 0 then
  begin
    open(pre,prefile,history := old);
    reset(pre);
  end;
for p := 1 to 100 do
  begin
    with graph_rec[p] do
      begin
        desptr := nil;
        predcount := 0;
        succlist := nil;
      end;
in_list[p] := nil;
out_list[p] := nil;
end;
pre_list := nil;
lex_error := false;
cycles_exist := false;
build_lists;
if length(prefile) <> 0 then
  pre_exist;
if lex_error then
  begin
    writeln(' ');
    writeln('Lex error exists--program execution halted. ');
  end
else
  begin
    build_graph;
top_sort;
    if cycles_exist then
      report_cycles
    else
      insert_semaphores;
  end;
writeln(' ');
writeln('Total program rows: ',total_rows : l);
end. (* end main program *)
module lexical(input, output);

(*Module- LEXICAL*)
(*This module contains the BUILD_LISTS and the PRE_EXIST procedures.*)
(*These procedures are responsible for building lists to contain the*)
(*input, output, and pre-existing program parameters. The program*)
(*names are also added to the graph.*)

(*)

(type

descriptor = varying[50] of char; (* Descrip. of a row of GRAPH_REC *)
desc_pointer = ^descriptor; (* Pointer to DESCRIPTOR *)
node_pointer = ^node_record; (* Points to nodes of the graph *)
node_record = record
  value: integer; (* Defines a node of graph *)
  next: node_pointer; (* Designates which node in graph *)
end;
graph_record = record
  desptr: desc_pointer; (* Points to descrip. of graph *)
  predcount: integer; (* No. of predecessors of node *)
  tag: boolean; (* Used in cycle detection *)
  succlist: node_pointer; (* List of successors of node *)
end;
record_132 = packed array[1..132] of char; (* 80 character record *)
top_sort_str = varying[50] of char; (* save the topological sort *)

var
description: [external] desc_pointer; (* Desc. of a row of GRAPH_REC *)
graph_rec: [external] array[1..100] of graph_record; (* Main graph *)
pre, source, out: [external] text; (* Input and output file names *)
prefile, (* pre-existing parameters *)
infile, outfile: [external] varying[25] of char; (* holds the file names *)
lex_error: [external] boolean; (* True if syntax errors exit *)
in_list, out_list: [external] array[1..100] of node_pointer; (* Input, output params *)
test, pre_list: [external] node_pointer; (* Pre-existing parameters *)
p: [external] integer;
record_hold: [external] array[1..100] of record_132;(* Hold for input rec *)
total_rows: [external] integer; (* Total number of program rows*)
cycles_exist: [external] boolean; (* true if cycles found in graph*)
top_sort_save: [external] array[1..100] of top_sort_str; (* save the topological sort *)

[global] procedure build_lists;

**********************************************************************
**********************************************************************

**********************************************************************
(* Procedure BUILD LISTS: *)
(* Reads each of the input parameters, each of the output parameters, *)
(* and each of the program names. If the value is an input parameter, *)
(* the integer part is placed into IN_LIST. Rows of IN_LIST correspond *)
(* to rows of the program. Input parameters must begin with the letter*)
(* "X". If the value is an output parameter, the integer part is *)
(* placed into OUT_LIST. Rows of OUT_LIST correspond to rows of the *)
(* program. Output parameters must begin with the letter "Y". If the *)
(* value is a program name, it will be placed the graph. Only minimal *)
(* syntax checking is performed on the program. *)

var
    p, i,
    number: integer;
    c: char;
    inrecord: record_132;
    box,
    in_rear,
    out_rear: node_pointer;
    p_found,
    valid_num: boolean;

begin
    p := 0;
    while not eof(source) do begin
        begin
            readln(source,inrecord);
            p := p + 1;
            record_hold[p] := inrecord;
            i := 1;
            p_found := false;
            while i <= 80 do
                begin
                    if inrecord[i] in ['X','x'] then begin
                        valid_num := true;
                        i := i + 1;
                        if inrecord[i] = ' ' then
                            begin
                                writeln('** Invalid input parameter on line ',p:1);
                                lex_error := true;
                                valid_num := false;
                            end;
                        number := 0;
                        while inrecord[i] <> ' ' do
                            begin
                                (* Read first input record *)
                                (* Next row of program *)
                                (* Keep for later use *)
                                (* First position of record *)
                                (* No program name found yet *)
                                (* Check each position of rec *)
                        end;
                    end;
                end;
            end;
        end;
    end;
c := inrecord[i];  (* Get character from input rec*)
if (c >= '0') and (c <= '9') then (* Valid digit? *)
  begin
    number := number * 10 + ord(c) - ord('0'); (*Alpha to int*)
    i := i + 1;  (* Look at next position *)
  end
else  (* Not a valid digit *)
  begin
    writeln('** Invalid input parameter on line ',p:',l);
    lex_error := true;  (* An error has been found *)
    valid_num := false;  (* Number is not valid *)
    while inrecord[i] <> ' ' do (* Find the next blank *)
      i := i + 1;  (* Look at next position *)
  end;
end;
if valid_num then (* Not unless number is valid *)
  begin
    new(box);  (* Get a new node *)
    boxA.value := number;  (* Put parm. number in node *)
    boxA.next := nil;  (* node will go at end of list *)
    if in_list[p] = nil then (* If this is the first node *)
      begin
        in_list[p] := box;  (* set pointer in IN_LIST *)
        in_rear := box;  (* set rear pointer *)
      end
    else  (* If not the first node *)
      begin
        in_rearA.next := box;  (* Place at rear of list *)
        in_rear := box;  (* Move rear pointer *)
      end;
  end
else
  if inrecord[i] in ['Y','y'] then (* Output parameters *)
    begin
      valid_num := true;  (* Valid unless told otherwise *)
      i := i + 1;  (* Look at next position *)
      if inrecord[i] = ' ' then (* Error if not a digit *)
        begin
          writeln('** Invalid output parameter on line ',p:',l);
          lex_error := true;  (* An error has been found *)
          valid_num := false;  (* Number is not valid *)
        end;
      number := 0;  (* Initialize to zero *)
      while inrecord[i] <> ' ' do (* Scan for digits til blank *)
        begin
          c := inrecord[i];  (* Get character from input rec*)
          if (c >= '0') and (c <= '9') then (* Valid digit? *)
            begin
              number := number * 10 + ord(c) - ord('0');  (* Convert alpha to integer *)
            end
        end
    end
  else
    ...
i := i + 1; (* look at next position *)
end
else (* Not a valid digit *)
begin
writeln('** Invalid output parameter on line ',p:l);
lex_error := true; (* An error has been found *)
valid_num := false; (* Number is not valid *)
while inrecord[i] <> ' ' do (* Find the next blank*)
i := i + 1; (* Look at next position *)
end;
end;
if valid_num then (* Not unless number is valid *)
begin
new(box);
box^value := number; (* Put param. number in node *)
box^next := nil; (* Node will go at end of list *)
if out_list[p] = nil then (* If this is the first node*)
begin
out_list[p] := box; (* Set pointer in OUT_LIST *)
out_rear := box; (* Set rear pointer *)
end
else (* If not the first node *)
begin
out_rear^.next := box; (* Place at rear of list *)
out_rear := box; (* Move rear pointer *)
end;
end
else if inrecord[i] in ['Q','q'] then (* Program name *)
begin
valid_num := true; (* Valid unless told otherwise *)
p_found := true; (* A program name is found *)
new(description); (* Get a new description *)
while inrecord[i] <> ' ' do (* Scan for digits til blank*)
begin
description^ := description^ + inrecord[i]; (* Add character to DESCRIPTION*)
i := i + 1; (* Look at next position *)
end;
graph_rec[p].desptr := description; (* Link description into graph *)
i := i + 1; (* Look at next position *)
end;
if not p_found then
begin
writeln('** Program name was not found on line ',p:l);
lex_error := true; (* An error has been found *)
end;
end;
total_rows := p; (* Remember how many prog lines*)
begin
  while not eof(pre) do
  begin
    readln(pre, inrecord);
    i := 1;
    while i <= 80 do
    begin
      if inrecord[i] in ['X', 'x'] then (* First char. must be a "Y" *)
      begin
        valid_num := true;
        i := i + 1;
        if inrecord[i] = ' ' then (* If blank, then error *)
        begin
          writeln('** Invalid pre-existing parameter name');
          lex_error := true;
          valid_num := false;
        end;
        number := 0;
        while inrecord[i] <> ' ' do (* Scan for digits only *)
        begin
          c := inrecord[i]; (* Get char. from input rec *)
          if (c >= '0') and (c <= '9') then (* Valid digit? *)
          begin
            number := number * 10 + ord(c) - ord('0'); (* Convert from alpha to integer *)
          end;
        end;
      end;
    end;
  end;
end; (* build_lists *)

[global] procedure pre_exist;

(* Procedure PRE_EXIST: *)
(* This procedure reads the file EXISTING.DAT. This file contains *)
(* names of parameters that exist before the program begins execution. *)
(* These parameters must be present in order to determine if the *)
(* program is complete. *)
(* )
(* Position in input record *)
(* Temp. holds a character *)
(* Points to rear of list *)
(* True if valid param. number *)
(* Alpha to integer conversion *)
(* Temp. pointer to node rec *)
(* Pre-existing parameters *)

var
  i: integer;
  c: char;
  rear: node_pointer;
  valid_num: boolean;
  number: integer;
  box: node_pointer;
  inrecord: record_l32;

begin
  while not eof(pre) do
  begin
    readln(pre, inrecord);
    i := 1;
    while i <= 80 do
    begin
      if inrecord[i] in ['X', 'x'] then (* First char. must be a "Y" *)
      begin
        valid_num := true;
        i := i + 1;
        if inrecord[i] = ' ' then (* If blank, then error *)
        begin
          writeln('** Invalid pre-existing parameter name');
          lex_error := true;
          valid_num := false;
        end;
        number := 0;
        while inrecord[i] <> ' ' do (* Scan for digits only *)
        begin
          c := inrecord[i]; (* Get char. from input rec *)
          if (c >= '0') and (c <= '9') then (* Valid digit? *)
          begin
            number := number * 10 + ord(c) - ord('0'); (* Convert from alpha to integer *)
          end;
        end;
      end;
    end;
  end;
end;
i := i + 1;       (* Look at next position *)
end
else
begin
  writeln('** Invalid pre_existing parameter name');
  lex_error := true;  (* An error was found *)
  valid_num := false; (* Not a valid number *)
  while inrecord[i] <> ' ' do
    i := i + 1;       (* Scan until a blank is found *)
  end;
end;
if valid_num then
begin
  new(box);
  boxA.value := number; (* Get a new node *)
  boxA.next := nil;     (* Put param. number in node *)
  if pre_list = nil then
begin
    pre_list := box;    (* Node will go at end of list *)
    rear := box;       (* PRE_LIST points at node *)
  end
else
begin
  rearA.next := box;  (* Place at rear of list *)
  rear := box;       (* Point to rear of list *)
end;
end;
  i := i + 1;       (* Look at next character *)
end;
end;
end;
module build(input, output);

(* **********************************************************************)
(* Module: BUILD *)
(* **********************************************************************)
(* This module contains two procedures; BUILD_GRAPH, INSERT_NODE. *)
(* Build_graph reads each element of in_list. Each element of out_list*)
(* is then read. If any of the elements are equal, an edge is then *)
(* inserted into the graph data structure. This insertion is done by *)
(* the insert_node procedure. Before the insertion is done, the graph *)
(* is checked to see if the edge already exists. If it does exist, *)
(* the insertion is not done. *)
(* If no output elements exist for a given input element, then pre-list*)
(* is checked to see if these elements are pre-existing parameters. *)
(* No edges are inserted into the graph for matches between these *)
(* elements. If a match is not found in pre_list, then an error has *)
(* occurred because this means that a given input parameter will not *)
(* be in existence when the corresponding program begins execution. *)
(* This error is reported to the user. *)
(* **********************************************************************)

type
  descriptor = varying[50] of char; (* Descrip. of a row of GRAPH_REC *)
  desc_pointer = ^descriptor; (* Pointer to DESCRIPTOR *)
  node_pointer = ^node_record; (* Points to nodes of the graph *)
  node_record = record
    value: integer; (* Designates which node in graph *)
    next: node_pointer; (* Pointer to next node *)
  end;
  graph_record = record
    desptr: desc_pointer; (* Points to descrip. of graph *)
    predcount: integer; (* No. of predecessors of node *)
    tag: boolean; (* Used in cycle detection *)
    succlist: node_pointer; (* List of successors of node *)
  end;
  record_132 = packed array[1..132] of char;(* 80 character record *)
  top_sort_str = varying[50] of char; (* save the topological sort *)

var
  description: [external] desc_pointer; (* Desc. of a row of GRAPH_REC *)
  graph_rec: [external] array[1..100] of graph_record; (* Main graph *)
  pre, source, out: [external] text; (* Input and output file names *)
  prefile, (* pre-existing parameters *)
  infile, outfile: [external] varying[25] of char; (* holds file names *)
  lex_error: [external] boolean; (* True if syntax errors exit *)
  in_list, out_list: [external] array[1..100] of node_pointer; (* Input, output params *)
  test, pre_list: [external] node_pointer; (* Pre-existing parameters *)
  p: [external] integer;
  record_hold: [external] array[1..100] of record_132;(* Hold for input rec *)
total_rows: [external] integer; (* Total number of program rows *)
cycles_exist: [external] boolean; (* true if cycles found in graph *)
top_sort_save: [external] array[1..100] of top_sort_str;
(* save the topological sort *)

(global) procedure build_graph;

var
  in_value, (* Value of an input node *)
  out_value: integer; (* Value of an output node *)
  in_p, (* line number of the org. prog. *)
  out_p: integer; (* line number of the org. prog. *)
  in_ptr, (* points to nodes of IN_LIST *)
  out_ptr, (* points to nodes of OUT_LIST *)
  pre_ptr: node_pointer; (* points to nodes of PRE_LIST *)
  match_found: boolean; (* T if elements are equal *)

(var)

procedure insert_node(succ, pred: integer);

var
  box, (* temp. pointer to a node *)
  ptr, (* temp. point. to graph edges *)
  last_ptr: node_pointer; (* points to next to last node *)
  already_exists: boolean; (* an edge already exists *)

begin
  already_exists := false; (* assume edge doesn't exist *)
  ptr := graph_rec[pred].succlist; (* set to first edge in graph *)
  while (ptr <> nil) and (not already_exists) do (* find end of edge list *)
    if succ = ptr^.value then (* does edge already exist? *)
      already_exists := true (* if it does, set to true *)
    else
      begin
        last_ptr := ptr; (* doesn't exist, rem. this node *)
        ptr := ptr^.next; (* get the next node *)
      end;
  if not already_exists then (* if the edge doesn't exist, *)
    begin
      graph_rec[succ].predcount := graph_rec[succ].predcount + 1; (* add 1 to predecessor count *)
      if graph_rec[pred].succlist = nil then (* is this the first succ. node*)
        begin
          new(box); (* get a new node *)
          box^.value := succ; (* store the succ. value in node *)
          box^.next := nil; (* this node goes at the end *)
          graph_rec[pred].succlist := box; (* link node into succ. list *)
        end
      else (* this isn't the first succ. node*)
        begin
          new(box); (* get a new node *)
        end
    end
  end;
begin
for in_p := 1 to total_rows do
begin
in_ptr := in_list[in_p];
while in_ptr <> nil do
begin
match_found := false;
in_value := in_ptr^.value;
for out_p := 1 to total_rows do
begin
out_ptr := out_list[out_p];
while (out_ptr <> nil) and (not match_found) do
begin
out_value := out_ptr^.value;
if in_value = out_value then
begin
insert_node(in_p, out_p);
match_found := true;
end;
end;
out_ptr := out_ptr^.next;
end;
end;
if not match_found then
begin
pre_ptr := pre_list;
while (pre_ptr <> nil) and (not match_found) do
begin
if in_value = pre_ptr^.value then
begin
match_found := true;
end;
else
pre_ptr := pre_ptr^.next;
end;
if not match_found then
writeln('*** Completeness Error on input line ',in_p:l);
end;
in_ptr := in_ptr^.next;
end;
end;
end;
module cycles(input,output);

**********************************************************************
(* Module : CYCLES *)
(* This module has two main procedures. TOP_SORT will detect if a *)
(* cycle exists in the graph. If a cycle does exist, a message is sent*)
(* to the user indicating such. REPORT_CYCLES will inform the user *)
(* which nodes are contained in the cycle. The algorithm used by this *)
(* module to detect and report cycles will only report the first cycle *)
(* it encounters. If more than one cycle exist in the graph, the first*)
(* one will be reported and this module will stop execution. *)
(* )
***********************************************************************


type
descriptor = varying[50] of char; (* Descrip. of a row of GRAPH_REC *)
desc_pointer = ^descriptor;
(* Pointer to DESCRIPTOR *)
ode_pointer = ^node_record;
(* Points to nodes of the graph *)
node_record = record
  value: integer;
  (* Designates which node in graph *)
  next: node_pointer;
  (* Pointer to next node *)
end;

graph_record = record
  desptr: desc_pointer;
  (* Points to descrip. of graph *)
  precount: integer;
  (* No. of predecessors of node *)
  tag: boolean;
  (* Used in cycle detection *)
  succlist: node_pointer;
  (* List of successors of node *)
end;

record_132 = packed array[1..132] of char;(* 80 character record *)
top_sort_str = varying[50] of char; (* save the topological sort *)

var
description: [external] desc_pointer; (* Desc. of a row of GRAPH_REC *)

graph_rec: [external] array[1..100] of graph_record; (* Main graph *)
pre, source, out: [external] text; (* Input and output file names *)
prefile, (* pre-existing parameters *)
infline, outfile: [external] varying[25] of char; (* holds file names *)
lex_error: [external] boolean; (* True if syntax errors exit *)
in_list, out_list: [external] array[1..100] of node_pointer;
(* Input, output params *)
test, pre_list: [external] node_pointer; (* Pre-existing parameters *)
p: [external] integer;
record_hold: [external] array[1..100] of record_132;(* Hold for input rec *)
total_rows: [external] integer; (* Total number of program rows*
cycles_exist: [external] boolean; (* True if a cycle is found *)
top_sort_save: [external] array[1..100] of top_sort_str;
(* save the topological sort *)

***********************************************************************
[global] procedure top_sort;

(* *********************************************************************** *)
(* Procedure: TOP_SORT *)
(* Sorts the graph into topological order. The algorithm is obtained *)
(* from "An Introduction to Data Structures with Applications" by *)
(* Tremblay and Sorenson, published by McGraw-Hill Book Company. *)
(* Pages 438-439. *)
(* *********************************************************************** *)

var q, trash, front, rear: node_pointer;
hold, k, i, n: integer;
box, work_ptr: node_pointer;
save_succs: array[l..100] of node_pointer;
program_sort_number: integer;

(* *********************************************************************** *)
procedure lqinsert(n: integer);

(* *********************************************************************** *)

begin
box: node_pointer;

begin
new(box);
box^.value := n;
box^.next := nil;
if rear = nil then
begin
front := box;
rear := box;
end

(* get a new box record *)
(* n - current node visited *)
(* to be put at end of list *)
(* is this first node in list *)
(* link to front pointer *)
(* link to rear pointer *)
else
begin
  rear^.next := box;
  rear := box;
end;
end;

(* The following segment of code is used to work around a bug found *)
(* in the Pascal compiler. The problem being caused by the bug is *)
(* that the graph data structure (GRAPH_REC) is being altered in the*)
(* immediately preceding procedure LQINSERT. As can be seen by *)
(* examining LQINSERT, no reference is ever made to GRAPH_REC. *)
(* However, the graph data structure will be in one state when *)
(* LQINSERT is entered, but will be in another state when *)
(* LQINSERT finishes executing. To solve this problem, a duplicate *)
(* copy of GRAPH_REC is made at this point of the proc. TOP_SORT. *)
(* TOP_SORT calls the procedure LQINSERT several times and it is at *)
(* these times that the error occurs. At the end of TOP_SORT, the *)
(* original graph is recreated from the copy that is being made at *)
(* this time. The entire graph is not copied, but only the edge *)
(* nodes are copied as this is where the problem was occurring *)

begin (* TOP_SORT *)
  front := nil;
  rear := nil;
  for i := 1 to total_rows do
    begin
      work_ptr := graph_rec[i].succlist;
      while work_ptr <> nil do
        begin
          new(box);
          box^.value := work_ptr^.value;
          box^.next := nil;
          if rear = nil then
            begin
              front := box;
              rear := box;
            end
          else
            begin
              rear^.next := box;
              rear := box;
            end;
          work_ptr := work_ptr^.next;
        end;
      save_succs[i] := front;
    end;
end;
(* End of saving the graph--continue with main part of TOP_SORT now. *)

front := nil;
rear := nil;
for i := 1 to total_rows do
    if graph_rec[i].predcount = 0 then
        lqinsert(i);
program_sort_number := 0;
while front <> nil do
    begin
        k := front^.value;
        n := n - 1;
        trash := front;
        front := front^.next;
        dispose(trash);
        program_sort_number := program_sort_number + 1;
        top_sort_save[program_sort_number] := graph_rec[k].desptr;
    end;
n := total_rows;
for i := 1 to total_rows do
    if n <> 0 then
        begin
            writeln(' ');
            writeln('*** System not feasible--One or more cycles exist in system.');
            writeln('The following cycle exists:');
            cycles_exist := true;
            * global variable
        end;
[global] procedure report_cycles;

(* Procedure: REPORT_CYCLES *)
(* This procedure reports all nodes that are contained in a cycle. *)
(* This procedure is only called if the procedure TOP_SORT detects *)
(* that cycles do exist in the system. The algorithm for this *)
(* procedure was obtained from "An Introduction to Data Structures *)
(* with Applications" by Tremblay and Sorenson, published by *)
(* *)

var
  k, i, j, succ: integer;
  p: node_pointer;
  quit: boolean;
begin
  for i := 1 to total_rows do
    with graph_rec[i] do
      begin
        predcount := 0;
        tag := false;
      end;
  for i := 1 to total_rows do
    if graph_rec[i].desptr <> nil then
      begin
        p := graph_rec[i].succlist;
        while p <> nil do
          begin
            succ := p^.value;
            if graph_rec[succ].predcount = 0 then
              graph_rec[succ].predcount := i;
            (* predcount points to next *)
            (* node in the cycle *)
            p := p^.next;
            end;
          end;
        i := 1;
        quit := false;
        while (i <= total_rows) and (not quit) do
          begin
            (* start at the first node *)
            (* true when begin of cycle found *)
            (* marks each node in cycle *)
            (* finishes when all nodes have *)
            (* been marked *)
  end;
if graph_rec[i].predcount <> 0 then
  while not graph_rec[i].tag do
    begin
      graph_rec[i].tag := true;
      i := graph_rec[i].predcount;
      quit := true;
    end
    else
      i := i + 1;
  end
  j := 0;
  while graph_rec[i].predcount <> 0 do
    begin
      k := j;
      j := i;
      i := graph_rec[j].predcount;
      graph_rec[j].predcount := k;
    end;
  graph_rec[i].predcount := j;
  while graph_rec[i].tag do
    begin
      writeln(graph_rec[i].desptr);
      graph_rec[i].tag := false;
      i := graph_rec[i].predcount;
    end;
  writeln(graph_rec[i].desptr);
end;
end.

(* not zero if in cycle *)
(* check all nodes in cycle *)
(* true for a nodes in cycle *)
(* points to next node *)
(* cycle has been found *)
(* cycle not found-try next node*)
(* reverse the order of the cycle*)
(* go through the cycle and *)
(* print the nodes in the cycle*)
(* node has been printed *)
(* get next node to be printed *)
(* the node that completes the *)
(* cycle *)
module semas(input, output);

(* **********************************************************************)
(* * Module: SEMAS *
(* * This module takes the graph and uses it to create a new program *
(* source listing with the proper semaphores inserted in the proper *
(* places. This is performed by looking at each row of the graph *
(* record. Each of the successors are then examined. Since each of *
(* the successors is dependent on its predecessor a semaphore will be *
(* inserted between these two programs. This will be done by inserting*
(* a "P" operator infront of the successor program and inserting a "V" *
(* operation behind a predecessor program. This module also checks to *
(* make sure that there is enough room remaining on each row of the *
(* program to insert the semaphore. If there is insufficient room *
(* the semaphore isn't inserted and an error message is reported. The *
(* output file isn't produced when this error occurs. *
(* *
(* **********************************************************************)

**type**
descr = varying[50] of char; (* Descrip. of a row of GRAPH_REC *)
desc_pointer = ^descr; (* Pointer to DESCRIPTOR *)
desc_pointer = node_record; (* Points to nodes of the graph *)
node_record = record
  value: integer; (* Designates which node in graph *)
  next: node_pointer; (* Pointer to next node *)
end;

graph_record = record
  descr: desc_pointer; (* Points to descr. of graph *)
  predcount: integer; (* No. of predecessors of node *)
  tag: boolean; (* Used in cycle detection *)
  succlist: node_pointer; (* List of successors of node *)
end;
record_132 = packed array[1..132] of char; (* 80 character record *)
top_sort_str = varying[50] of char; (* save the topological sort *)

**var**
description: [external] descr_pointer; (* Desc. of a row of GRAPH_REC *)
graph_rec: [external] array[1..100] of graph_record; (* Main graph *)
pre, source, out: [external] text; (* Input and output file names *)
prefile, infile, outfile: [external] varying[25] of char; (* pre-existing parameters *)
lex_error: [external] boolean; (* True if syntax errors exit *)
in_list, out_list: [external] array[1..100] of node_pointer; (* Input, output params *)
test, pre_list: [external] node_pointer; (* Pre-existing parameters *)
p: [external] integer;
record_hold: [external] array[1..100] of record_132; (* Hold for input rec *)
total_rows: [external] integer; (* Total number of program rows*)
cycles_exist: [external] boolean; (* True if a cycle is found *)
top_sort_save: [external] array[1..100] of top_sort_str;
(* save the topological sort *)

[global] procedure insert_semaphores;

type
init_string - packed array[1..13] of char; (* Sernas that are initialized *)

var
serna_inits: array[1..200] of init_string; (* array on init'ed semas *)
semaphore_number: integer;
(* index into serna_inits *)
i, (* a loop counter *)
j, (* a loop counter *)
k, (* a loop counter *)
l: integer; (* a loop counter *)
value: integer; (* a temporary variable *)
succs: node_pointer; (* ptr to graph nodes *)
sernal, (* a semaphore number *)
sema2: packed array[1..3] of char; (* a semaphore number *)
do_print, (* output file created ?? *)
end_found: boolean; (* end of a row found *)

function convert(n: integer): char;

(* CONVERT *)
(* Converts an integer number to a character *)
(* digit. *)
(* *)

begin
  case n of
    0: convert := '0';
    1: convert := '1';
    2: convert := '2';
    3: convert := '3';
    4: convert := '4';
    5: convert := '5';
    6: convert := '6';
    7: convert := '7';
    8: convert := '8';
    9: convert := '9';
  end;

begin
for i := 1 to 200 do
  sema_inits[i] := ' '; (* init semas array *
  semaphore_number := 0; (* with blanks *
  do_print := true; (* no semas have been created *
for i := 1 to total_rows do
  begin
    value := i; (* look at first row of graph *
    semal[3] := convert(value mod 10); (* init semas array *
    value := value div 10; (* be optimistic *
    semal[2] := convert(value mod 10); (* look at first row of graph *
    value := value div 10;
    semal[1] := convert(value mod 10); (* convert integer to char *
    succs := graph_rec[i].succlist;
    while succs <> nil do
      begin
        semaphore_number := semaphore_number + 1; (* this is a new one *
        j := succs\^\ .value;
        value := j; (* get the successor value *
        sema2[3] := convert(value mod 10); (* convert the third digit *
        value := value div 10; (* get rid of third digit *
        sema2[2] := convert(value mod 10); (* convert the second digit *
        value := value div 10; (* get rid of second digit *
        sema2[1] := convert(value mod 10); (* convert the first digit *
        sema_inits[semaphore_number,1] := 'S'; (* create the sema *
        sema_inits[semaphore_number,2] := semal[1]; (* insert number *
        sema_inits[semaphore_number,3] := semal[2];
        sema_inits[semaphore_number,4] := semal[3];
        sema_inits[semaphore_number,5] := sema2[1]; (* insert number *
        sema_inits[semaphore_number,6] := sema2[2];
        sema_inits[semaphore_number,7] := sema2[3];
        sema_inits[semaphore_number,8] := ' ';
        sema_inits[semaphore_number,9] := ' ';
        sema_inits[semaphore_number,10] := ' ';
        sema_inits[semaphore_number,11] := ' ';
        sema_inits[semaphore_number,12] := '0'; (* init to this value *
        sema_inits[semaphore_number,13] := ' '; (* insert "V" sema at rear of program line *
        k := 132; (* No of pos. in record_hold *
        end_found := false; (* end of input record_found *
        while (k <> 0) and (not end_found) do (* do until end of record *
          if record_hold[i,k] = ' ' then (* check next position *
            k := k - 1 (* is position blank *
            else (* if not blank *
              end_found := true; (* end has been found *
              if k <= 121 then (* enough room in insert sema? *
                begin
                  k := k + 2; (* yes *
                  (* back up to positions *)
record_hold[i,k] := 'V'; (* insert the "V" operation *)
record_hold[i,k + 1] := '(';
record_hold[i,k + 2] := 'S'; (* insert the proper semaphore *)
record_hold[i,k + 3] := semal[1];
record_hold[i,k + 4] := semal[2];
record_hold[i,k + 5] := semal[3];
record_hold[i,k + 6] := sema2[1];
record_hold[i,k + 7] := sema2[2];
record_hold[i,k + 8] := sema2[3];
record_hold[i,k + 9] := ')';
end
else (* no *)
begin (* indicate that sema can't *)
writeln(' '); (* be inserted *)
writeln('Program Record Length Too Long For Semaphore Addition');
writeln('Program Line Number: ',i:1);
do_print := false; (* don't print output file *)
end;
(* insert "P" sema at front of program line *)
k := 132; (* no. of pos. in record_hold *)
end_found := false; (* end hasn't been found yet *)
while (k <> 0) and (not end_found) do (* do until line end found *)
if record_hold[j,k] = ' ' then (* is position blank *)
k := k - 1 (* check next position *)
else (* end has been found *)
end_found := true;
if k <= 121 then (* enough room to insert sema *)
begin (* yes *)
for l := k + 11 downto 12 do (* move chars right by 12 *)
  record_hold[j,l] := record_hold[j,1 - 11];
for l := 1 to 11 do (* fill emptied pos. with blank*)
  record_hold[j,1] := 'P'; (* insert "P" operation *)
  record_hold[j,2] := '(';
  record_hold[j,3] := 'S'; (* insert the semaphore *)
  record_hold[j,4] := semal[1];
  record_hold[j,5] := semal[2];
  record_hold[j,6] := semal[3];
  record_hold[j,7] := sema2[1];
  record_hold[j,8] := sema2[2];
  record_hold[j,9] := sema2[3];
  record_hold[j,10] := ')';
end
else (* no *)
begin (* indicate the sema can't *)
writeln(' '); (* be inserted *)
writeln('Program Record Length Too Long For Semaphore Addition');
writeln('Program Line Number: ',j:1);
do_print := false;
end;
succs := succs^.next; (* get the next successor and *)
end;
end;
if do_print then (* is it ok to print *)
begin (* yes *)
  open(out,outfile,history := new);
  rewrite(out);
  i := 1;
  while sema_inits[i,1] <> '' do (* index into sema_init table *)
  begin (* do until no more entries *)
    begin (* write out the line *)
      writeln(out,sema_inits[i]); (* go to the next line *)
      i := i + 1;
    end;
    writeln(out);
  end;
  for i := 1 to total_rows do (* write out the program lines *)
    writeln(out,record_hold[i]);
  writeln(out); (* skip some lines *)
end;
end else (* no *)
begin
  writeln(' ');
  writeln('Program Row Length Errors Encountered In Source Program');
  writeln('No Output File Was Produced');
end;
end.
end.