Report Number: WUCS-86-13

1986-07-01

A Compiler for a Two-Dimensional Programming Language

Julie Wing Kam Choi

A visual programming language is presented. This language uses interactive graphics to convey notion such as subroutine, recursion, block structure, parallel and serial processing to school children. Currently the system is interpreter based. To overcome the inefficiency of the interpreter based system, a compiler is implemented for this language. This report gives an overview of the compiler and the details about the parser, semantic analyzer and the code generator. Finally, a performance comparison between the interpreter based system and the compiler based system is given.

A COMPILER FOR A TWO-DIMENSIONAL PROGRAMMING LANGUAGE

Julie Wing Kam Choi

WUCS-86-13

July 1986

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This research was supported in part by Computer Services Corporation (CSK). A thesis presented to the Sever Institute of Washington University in partial fulfillment of the requirements for the degree Master of Science.

Abstract

A visual programming language is presented. This language uses interactive graphics to convey notions such as subroutine, recursion, block structure, parallel and serial processing to school children. Currently the system is interpreter based. To overcome the inefficiency of the interpreter based system, a compiler is implemented for this language.

This report gives an overview of the compiler and the details about the parser, semantic analyzer and the code generator . Finally, a performance comparison between the interpreter based system and compiler based system is given.

A COMPILER FOR A TWO DIMENSIONAL PROGRAMMING LANGUAGE

1. INTRODUCTION

As personal computers become more accessible and powerful, people are finding them very useful in business, at school or at home due to the great variety of software available. However, this ready-made software is often addressed to a general audience. Therefore, many people find that some aspects of a given program serve their needs but other aspects are not satisfying. Those people who wish to use the computer to solve their specific needs often find themselves programming.

For most people, programming is a skill which can only be developed through training in college. Learning how to program without help is next to impossible. Learning a programming language itself, both syntax and semantics, and then transforming an algorithm into a program requires a great deal of effort. On top of that, in order to write a program, a user may have to learn how to enter, compile and assemble, and finally link and load the program. Many system manuals may have to be read before one can solve the problem.

New programming languages have been proposed through the years in order to solve this problem. But it is not until recent years that researchers realized that a radical departure from traditional programming languages may be useful. Hardware technology has made it possible to fabricate high resolution graphic chips and memory chips in the

order of megabytes at low cost. Various programming languages have been developed which make use of this graphics hardware to make programming easier. These kinds of languages are usually termed *visual programming languages*.

Since visual programming research is a relatively new field, there is no concensus as to what kind of language is more acceptable than another. Even how to represent popular programming constructs is a problem. In this thesis, a new visual programming language called the Show and Tell™* Language is presented. This language integrates three application areas: computation, database and communication. An interpreter-based implementation has been developed. However, to enhance the performance of the system, a compiler is proposed.

All of the known visual language system implementations are interpreter based. The compiler we propose represents the first step towards compiling these visual languages. There are three motivations for providing a compiler for the Show and Tell Language:

(1) For execution efficiency - During program development, an interpreter is convenient in spite of slow running time. However, when the debugging or refinment phase of program development is finished, the run time efficiency becomes important.

(2) Linkage between two-dimensional and one-dimensional programming environment - Since most of the existing software is written in one-dimensional programming languages, the object code generated by a compiler can be linked with the existing software. Thus, the proposed compiler will provide a tool to integrate one-dimensional and two-dimensional programming language programs.

(3) Lay ground for future research - The difficulty of generating a compiler will be better understood, and this will be useful for future research into this area.

*Show and Tell is a trademark of Computer Services Corporation (CSK).

The second chapter of this thesis will survey related work in the area of visual programming languages. Then the Show and Tell Language will be introduced through examples in the third chapter. Chapter four is an overview of the compiler specific for this language. Then detailed translation rules will be discussed. Finally, some measurements of compiler performance are presented in chapter five.

2. RELATED WORK

There are two approaches to integrate the graphic capabilities into a language system. The first approach is using pictures to help in representing various aspects of a program, in control flow, dataflow or data structure. The most popular example in representing control flow is to use flow charts or Nassi-Shneiderman diagrams. Dataflow diagrams are usually used in conjunction with dataflow languages. Displaying data structures are an effective use of graphics, particularly if the program can be run interpretively while changes to the data structures are displayed. A second approach in providing a visual language is that graphical elements such as rectangles, circles, arrows or other geometric objects are an integral part of the language, where the terminal symbols of the language consist of these kinds of objects. A number of systems have been developed using these two approaches. Some of them are described below.

2.1 PECAN

PECAN (1)* is a system developed at Brown University. It is a graphical extension to Pascal. This program development system allows multiple views of a user's program, both static and dynamic views. Static views include Nassi-Schneiderman diagram, structure flow graph of a program, data types, expression tree and symbol tables. Dynamic views show the stack content, error messages incurred during execution, program input and output and program in action by highlighting each statement as it is being executed.

A PECAN user indicates the actions he wishes to perform by pointing the mouse to items in system menus. Since the editor is syntax directed, it can give immediate feedback to user if there is any syntactic errors in the program. Incremental compilation allows a user to run a program after writing it or making changes in it without

* The numbers in parentheses in the text indicate references in the Bibliography.

recompiling the unchanged parts. A user can step through a program during execution or he can set break points and examines variables at those break points.

PECAN integrates different tools to support system development. Unlike earlier systems, such as (2) and (3), which only concentrate on the structure flow of a program, PECAN allows multiple views of a program at edit time and also at run time, a great help for debugging. However, it does not contribute any new paradigm that utilizes new technology to enhance program representation.

2.2 OMEGA

OMEGA (4) is a system which incorporates interactive graphics and pointing devices to allow a user to create and modify program structures stored in a database. Its approach is to combine one-dimensional and two-dimensional programming language into paradigms with a single language.

The fundamental concept is abstraction. An abstraction has three parts: the pictograph that the abstraction represents, the parameters, and the semantics of the operations. A pictograph is a visual object that the programmer sees and manipulates. It consists of letters and icons arranged in a two-dimensional area.

While the most fundamental abstractions (pictographs) are defined in textual form, high level abstractions can be defined in terms of the existing pictographs which are in two-dimensional form. This provides an integration between one-dimensional programming language and two-dimensional programming language. It gives a user a choice between expressing a program in terms of character-string tokens or graphical pictures.

A glossary is a list of pictographs defined by the user. Each pictograph can be used any time in order to define new abstractions. For example, to use a pictograph in a program, a user has to select it from the glossary and then place it at the desired point in

the statement list he is working on. All pictographs are stored in a general purpose database. The database system allows a user to view or modify pictographs.

Therefore, an OMEGA program is a mixture of text and pictures. The most fundamental pictographs are all defined by text. After a glossary of pictographs is defined, a program can be defined using these high level abstractions and eventually programs can be composed only of pictures.

2.3 PROGRAMMING BY REHEARSAL

Programming by Rehearsal (5), designed by Finzer and Gould, is a visual programming environment that non-programmers can use to create educational software. It is implemented in the Smalltalk-80 programming environment and runs on the Xerox Dorado.

Programming in this system is analogous to rehearsaling for a theater production. A program is a theater production. The basic component of a production is performer. A performer corresponds to a process in a program. Communications between processes are done by sending cues. A cue is a message sent by a performer.

The first step in composing a program in the system is to choose performers. Primitive performers are defined in the system and a user can select them using the mouse. The set of performers can be extended as users create new ones and teach them new cues. Each performer can be moved or resized to the desired location or size. The second step is rehearsing the production by showing each performer to perform what actions on what events. Each performer has a tiny icon representing an closed eye. When the eye of a performer is selected, the eye will open and all actions that the user perform will be recorded. The codes for the actions will also be displayed on another window.

A user does not need to understand the codes at all. Because he can debug the codes by sending cues to the performer which he has just programmed and see whether it

acts as it is told. A more advanced user can choose to examine the codes displayed on the window. There is a large on-line help facility to assist a rehearsal user.

Programming by Rehearsal is an experiment of how interactive, graphical programs can be built inside an interative, graphical programming environment. All objects and only objects that are visible can be manipulated.

2.4 PICT

PICT (6) is a system developed at the University of Washington. It is a pictorial representation of Pascal. All program names, parameter passing, data structures, variables and program operations are represented by icons. It is a complete system in that all the tools a user needs to compose, edit and run his program are in the same system framework.

There are five different mode of operations which a PICT user can select. They are the programming mode, the erasing mode, the icon editing mode, user library and program execution mode. It is in the programming mode that a user constructs a program. Erasing mode allows a user to delete control paths or icons. If an icon is deleted, all the paths related to that icon will also be deleted. Icon editing mode is the mode of operation which allows a user to design new names for user routines. User library mode allows a user to examine or select program names which are currently defined in the system. Finally, program execution mode runs users programs. Every program is identified by its icon. If an icon is selected, the associated program may either be edited or run.

As a programming language, prototype PICT/D's capabilities are limited to handle simple numeric calculations. However, it proposes a way of representing well-known programming language constructs such as subroutine, while-do loop, and if-then-else in a

two-dimensional manner. It is important to note the use of color in the PICT system, which gives more flexibility in designing a visual environment.

Besides the systems described above, some of the other projects under way are *Programming in Pictures* (7) at University of Southern California, *Program Visualization* (8) at Computer Corporation of America, and *PegaSys* (9) at SRI.

3. THE SHOW AND TELL LANGUAGE (STL)

3.1 OVERVIEW

In this section STL is introduced. A subset of STL, called the Simple Show and Tell Language (SSTL), will be the source language for the proposed compiler. SSTL will be introduced in the next section. STL is an icon-driven visual programming language. All programs and data are in pictorial form. It is designed for users who have little or no training in programming. Currently, it is being tested by school children. However, the language is powerful enough to express complex operations. In fact, it is computationally complete.

The language integrates three areas of applications: computation, database and communication. The integration is in a uniform conceptual framework of *dataflow* and *completion*. Dataflow is a well known concept in computer science (10). A dataflow program consists of nodes which represent operations that can be executed concurrently and arcs connecting nodes for communication between nodes. The only condition determining the execution of a node is that all the data from paths coming into a node have arrived. Unlike general dataflow programs, an STL program is an acyclic multigraph. There are no loops or cycles among the nodes. Besides the arrival of data, the condition determining the execution of a node is also affected by consistency. Consistency will be explained in section 3.4.

Completion on the other hand is a concept imported from psychology (11). Completion is the process of filling in missing portions of an incomplete pattern. It is illustrated in Figure 1 and Figure 2. The human brain performs the process of filling in incomplete patterns all the time. Given the pattern in Figure 1a, the brain completes the

missing portions and a person will abstract to the pattern in Figure 1b. Similarly, Figure 2a is another form of completion. However, the constraint is physical rather than psychological.

Figure 1a: Incomplete pattern

Figure 1b: Completed pattern

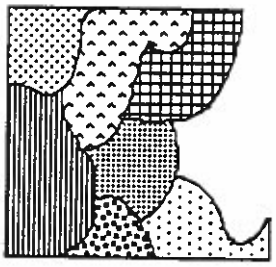


Figure 2a: Incomplete jig-saw puzzle

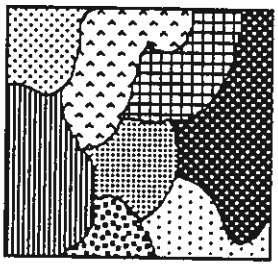


Figure 2b: Completed jig-saw puzzle

A Show and Tell program is called a *puzzle* because it is very similar to a jig saw puzzle except the constraints are logical rather than physical. The STL system is presented to users as a system which can perform completion. In the computation area, the system completes by execution of arithmetic operations. In the database area, completion is accomplished through pattern matching, and in the communication area, completion is accomplished through transmission of data through the network.

Section 3.2 describes the language by using a sequence of examples. It is not intended to be a reference manual. For a detail description of the language, refer to (12).

3.2 SYNTAX

3.2.1 Lexical Element

The following are the lexical elements and their names on the side.

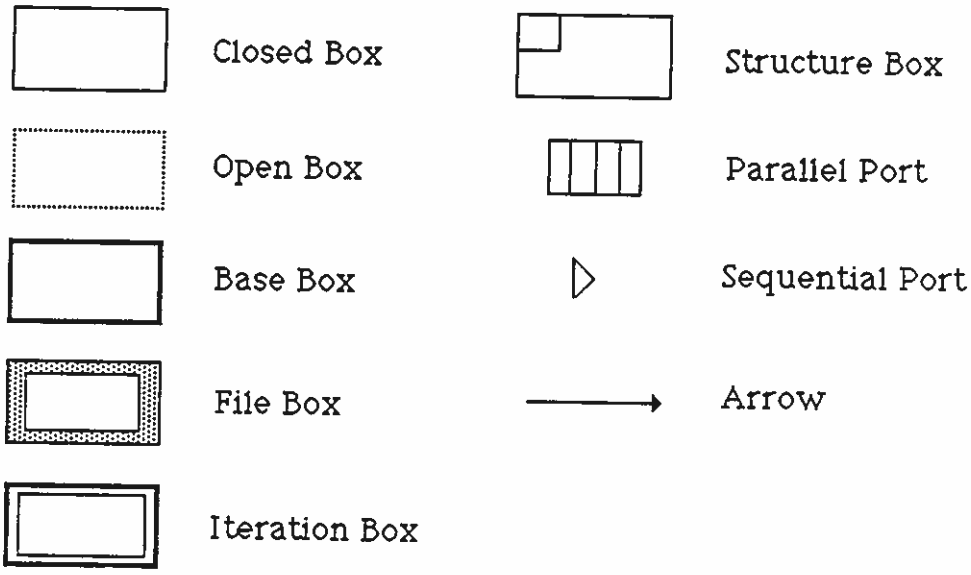


Figure 3: Lexical Elements of STL

3.2.2 Composition of ST Puzzle

A Show and Tell puzzle consists entirely of a nested set of *boxes* connected by *arrows*. There is no loop or cycle created by the combination of boxes and arrows. No box can overlap or touch another box. A box may be empty, or it may contain a data value or an icon which represents an operation or another STL puzzle. Each level of nesting is a partially ordered set of boxes. Each arrow serves as a path for a value to flow from one box to another. A box may not be executed until the value of each incoming arrow has arrived. Except for this dependency, there is no specific sequencing in the execution of a puzzle. Since there is no loop or cycle, once a box is executed, it will not be executed again. Thus once the value in a box is registered, it will never be changed again. Therefore, STL is a functional system; i.e., there are no side effects.

The composition rule of STL can be summarized in the following:

(1) No two boxes overlap with each other.

(2) No cycle nor loop exists.

(3) An arrow must start from a box and end at a box. An arrow can:

(i) starts from a simple box and ends at a simple box. A simple box can be a closed box, a base box, or an open box. The content of a simple box can be empty, a text, a number, an image, a user defined or a system defined icon.

(ii) starts from a structure and ends at a structure box.

(iii) starts from a simple box, goes through a sequential port and ends at a simple box.

(iv) starts from a simple or file box, goes through a parallel port and ends at a structure box.

(v) starts from a structure box, goes through a parallel port and ends at a simple or file box.

(4) No arrow traverses through any box other than an empty box or a box containing other boxes.

(5) An iteration box can have an arbitary number of sequential ports and parallel ports.

(6) A structure box may only contain base boxes.

(7) A file box can be connected to/from a structure box through a parallel port.

(8) There must be one and only one arrow through a sequential port.

(9) No arrow goes through more than one port.

(10) There must be at least one arrow through a parallel port.

3.3 SUBROUTINE

The most fundamental kind of box in STL is the *closed box*. This kind of box can hold values which can be numbers, text, images or definition of another puzzle (icon). Figure 4 shows how it is used (Figure 4 to Figure 20 are collected at the end of section 3). In this puzzle, there are five boxes connected by arrows. Box one has a value of 3; box two has a value of 5; box three and box four have values which are definitions of system puzzles named "*" and "+", and box five is currently empty. When the system solves this puzzle, the result will be in Figure 5.

In order to use the puzzle in Figure 4 as a subroutine, the system has to know what are the input and output variables. The *base box* is used for this purpose. A base box has the same meaning as a closed box except that it represents an input or output variable of a puzzle. The value of a base box can either be a number, text or an image. Figure 6 is a redefinition of Figure 4 with the input and output represented by base boxes.

An STL puzzle can be named by a user defined icon. Any Macpaint™ picture can be used as a puzzle name. In Figure 6, the name of the puzzle is the icon "star". It is used in Figure 7 as a subroutine. Recursive definition of a puzzle is also allowed. Figure 7.1 in Appendix 7.1 is an recursive definition of the factorial function. This example will be explained when the concept of consistency is introduced.

3.4 CONSISTENCY

Not all combinations of boxes and values represent correct puzzles. For example, in the puzzle in Figure 8, 2 is not greater than 3. If the system solves this puzzle, the result appears as in Figure 9. The box with a ">" sign is hatched to represent that there is something wrong with this box. This box is *inconsistent* in Show and Tell terms. From the system's point of view, the inconsistent box with all of its related arrows does not exist. There are two kinds of control mechanisms for propagation of the effects of inconsistency: the *closed box* and the *open box*. When a closed box is inconsistent, the

inconsistency is confined within the box. However, when an open box is inconsistent, the inconsistency can propagate out of it into the surrounding environment. Inconsistency propagates in the broadcasting mode within the boundary of the smallest closed box containing the inconsistent puzzle. There is no boolean value in STL; inconsistency is the main switching mechanism.

Figure 10 illustrates the difference between a closed box and an open box. When this puzzle is solved, the result will be as in Figure 11. The box A1 is inconsistent because there is a conflict of "2 flowing into 3". Since A1 is an open box, the inconsistency propagates to the surrounding box. Therefore B1 is also inconsistent. The inconsistent box B1 shuts off the communication between boxes C1 and D1. Therefore the data in C1 cannot reach its destination D1. However, when A2 become inconsistent, the inconsistency is confined within itself. Thus, the data in C2 can flow through the arrow and reaches its destination D2.

Consistency is used to switch between two sections of the puzzle in Figure 7.1. This puzzle is a recursive definition of the factorial function. There are two main parts in the puzzle, box 6 containing box 7 and 8 and box 0 containing boxes 1 to 5. Box 6 contains the actions to be performed when the incoming value from box 10 is zero. In this case, one is the result. Otherwise, box 6 will be inconsistency. On the other side, box 0 is the part determining whether the incoming value is greater than zero. If it is, another invocation of factorial with a new input argument will be made. Otherwise, box 0 will be inconsistency.

3.5 ITERATION

STL puzzles are strictly acyclic multigraphs. In order to represent iteration, a special kind of box called an *iteration box* is introduced. There are two kinds of iteration in the system, *sequential* and *parallel* iteration. Sequential iteration is indicated by the matching triangular sequential port attached on the side of an iteration box. Figure 12 is

an example of sequential iteration. This puzzle is a definition of the Fibonacci numbers. When this puzzle is executed, the puzzle inside the iteration box is expanded as long as it is consistent. Figure 13 is the conceptually expanded version of Figure 12.

Another form of iteration is parallel iteration. This kind of iteration is denoted by having a rectangular parallel port on the side of an iteration box. When both sequential and parallel iterations are present, the number of expansions is determined by the number of elements in a sequence entering the parallel port. Figure 14 is an example of parallel and sequential iteration. Figure 15 is the conceptual expansion of the iteration box in Figure 14 during execution.

To illustrate the power of parallel and sequential iteration, Figures 16 to 20 show the prime number generator. Figure 16 is the remainder function. It finds the remainder when the first value is divided by the second value. Figure 17 is a sequence generator. It will generate a sequence from 2,3,4 .... up to the number specified in the argument. Figure 18 is a filter. It accepts a sequence as its first argument and outputs as a sequence those numbers which are not divisible by the second argument. Figure 19, sieve, is the main driver. The sequence decomposition operation in the open box and the sequence composition operation in the closed box are system defined. The decomposition operation puzzle becomes inconsistent when the input sequence is null. This fact is used to terminate the recursion. It will call filter and itself recursively to filter out the non-prime numbers. Figure 20 is just calling the sequence generator and passing the sequence to sieve for processing.

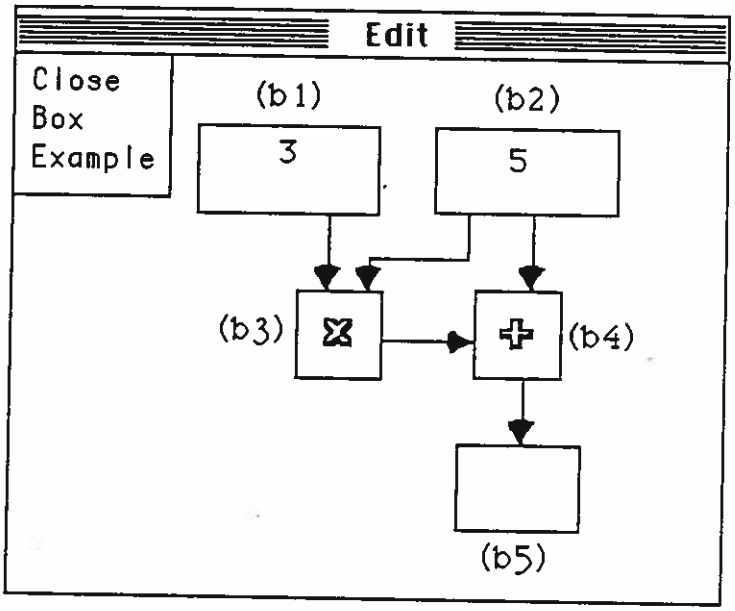


Figure 4: A Close Box Example

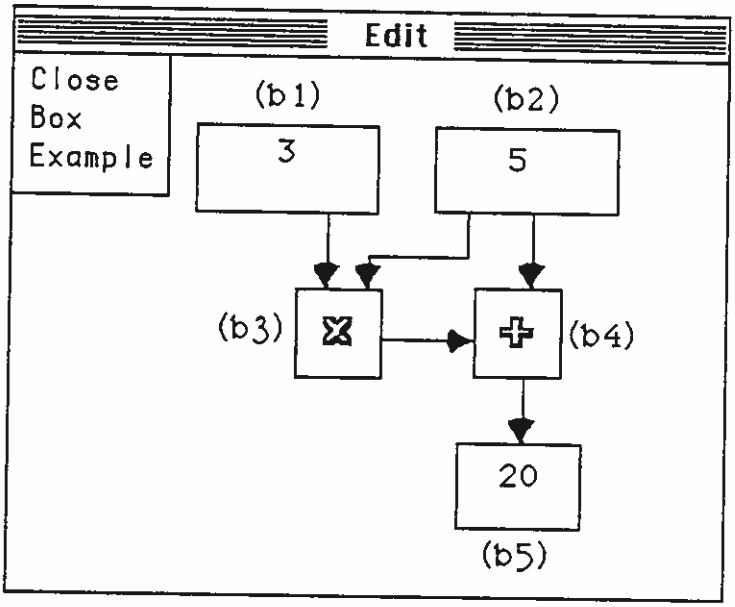


Figure 5: Display After Execution of Figure 4

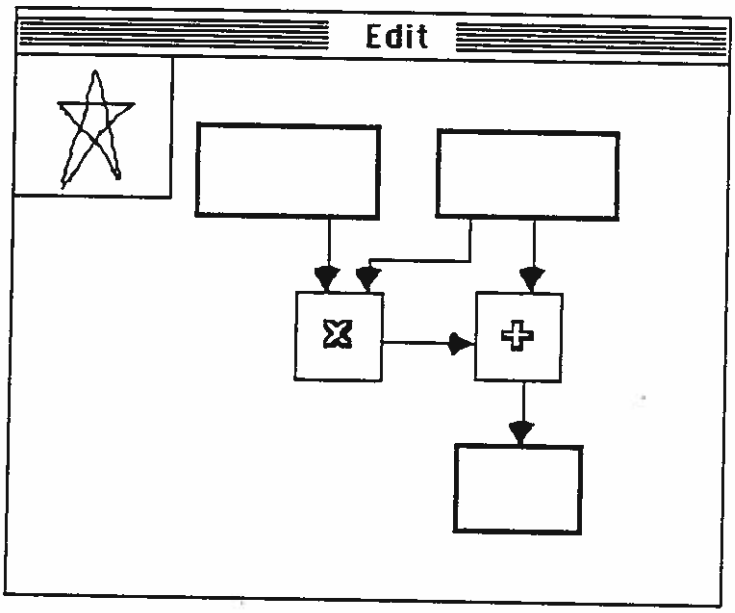


Figure 6: Redefinition of Figure 4

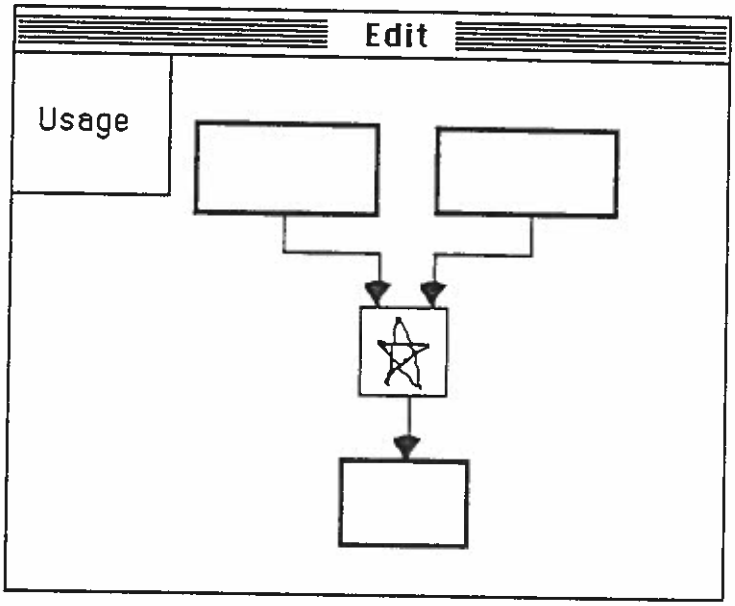


Figure 7: A Puzzle Using Figure 6 as Subroutine

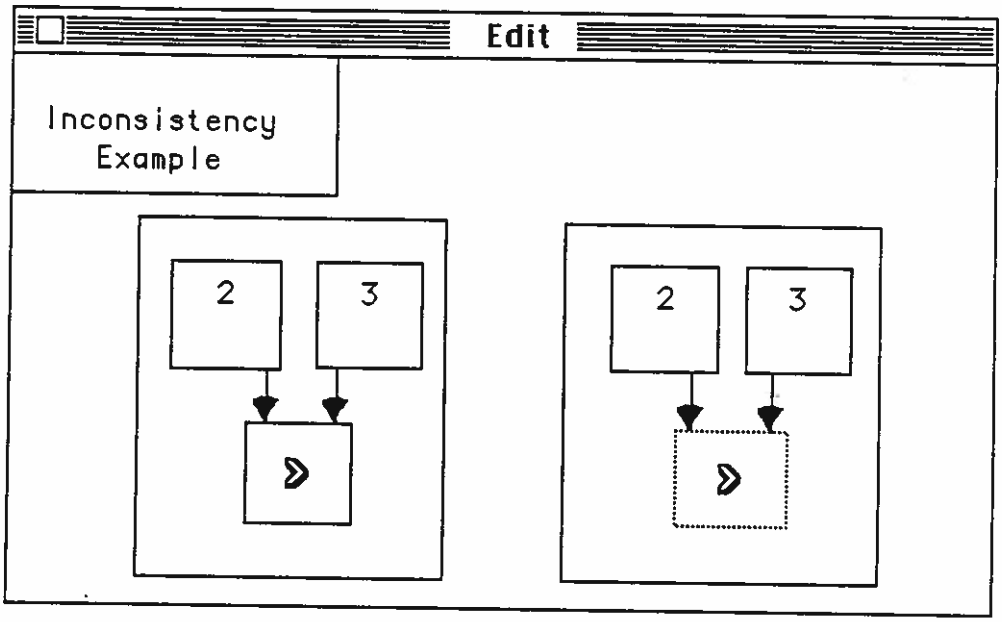


Figure 8: Inconsistency Example 1

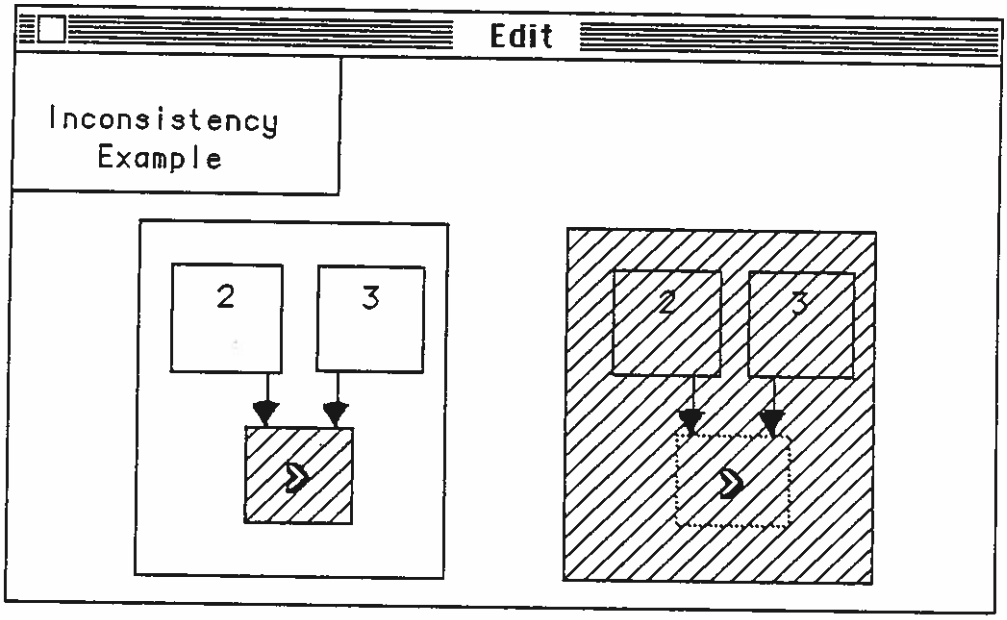


Figure 9: Display of Figure 8 after execution

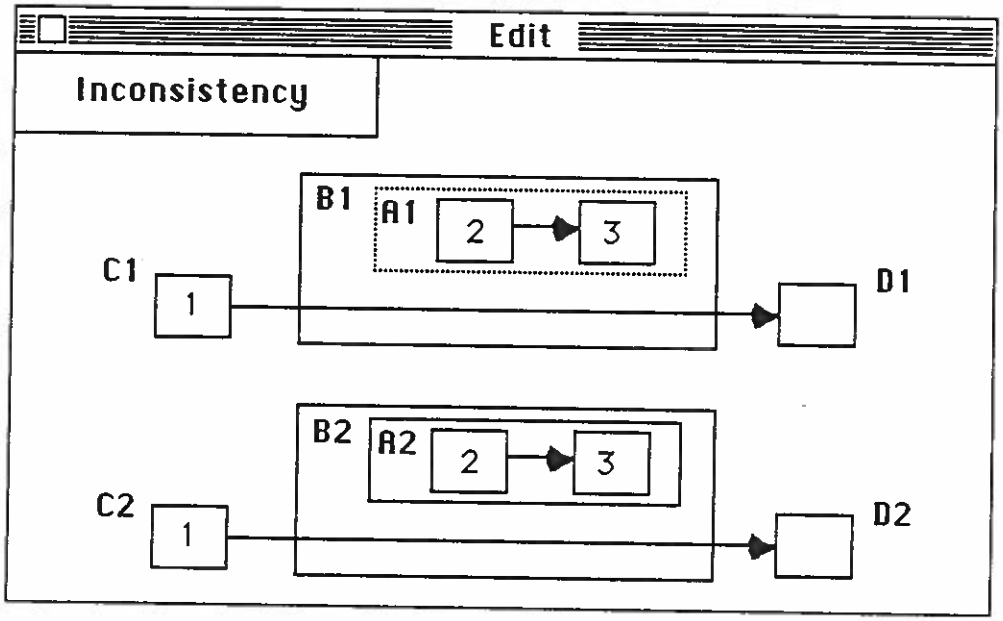


Figure 10: Inconsistency Example 2

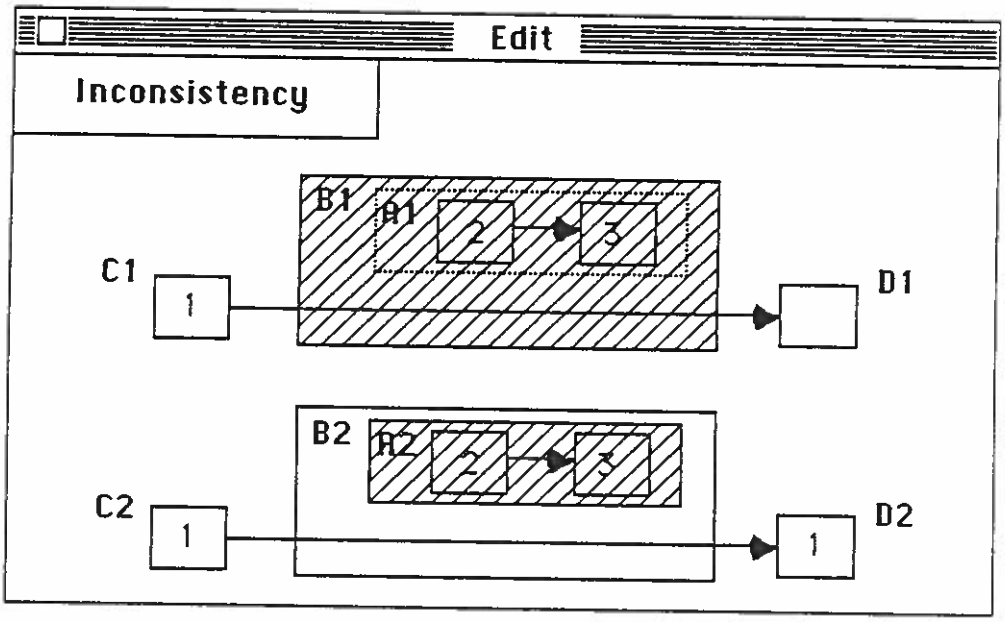


Figure 11: Display of Figure 10 after execution

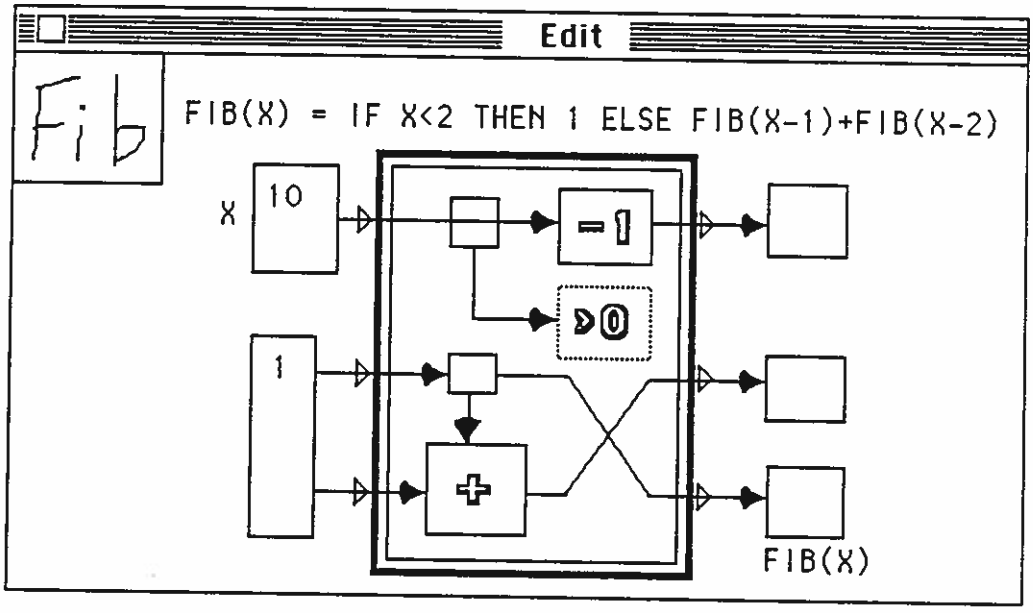


Figure 12: STL puzzle to find the 10th Fibonacci Number

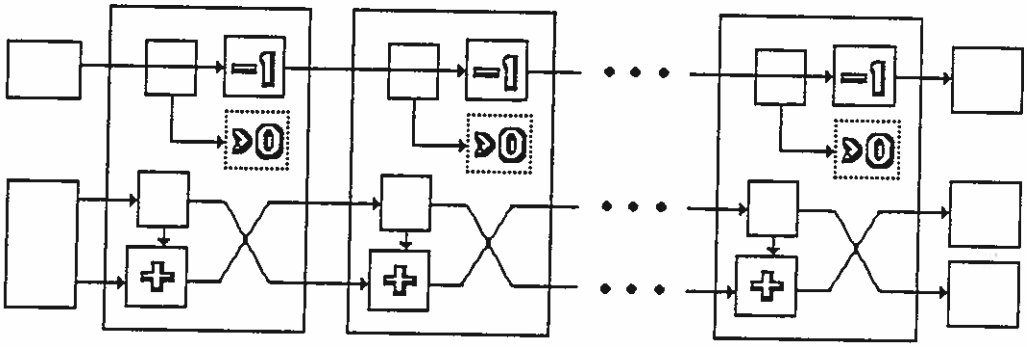


Figure 13: Conceptual Expansion of Figure 12

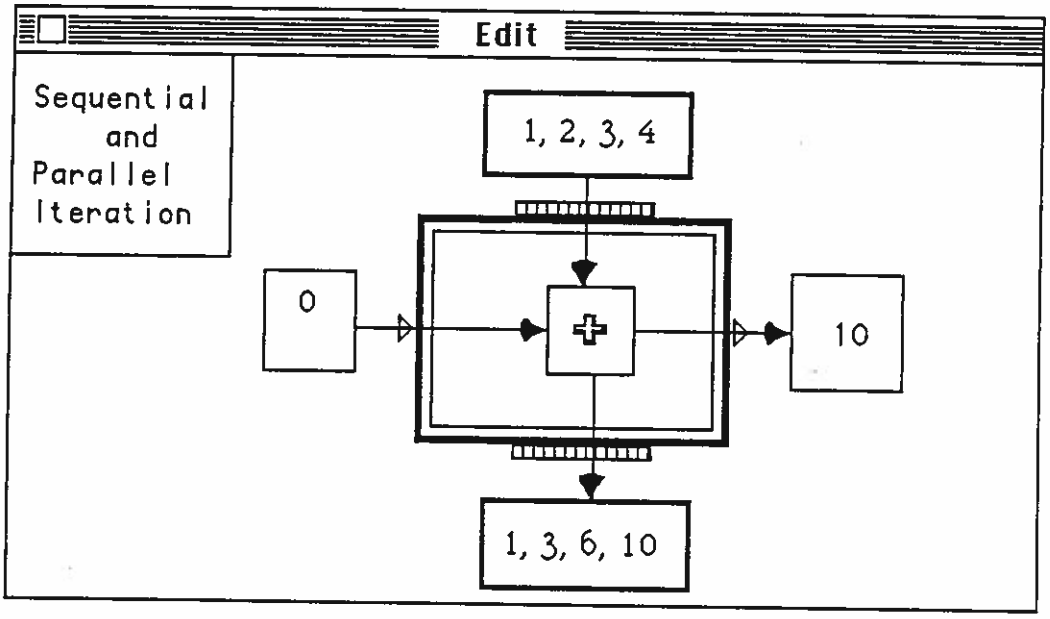


Figure 14: An example of sequential and parallel iteration

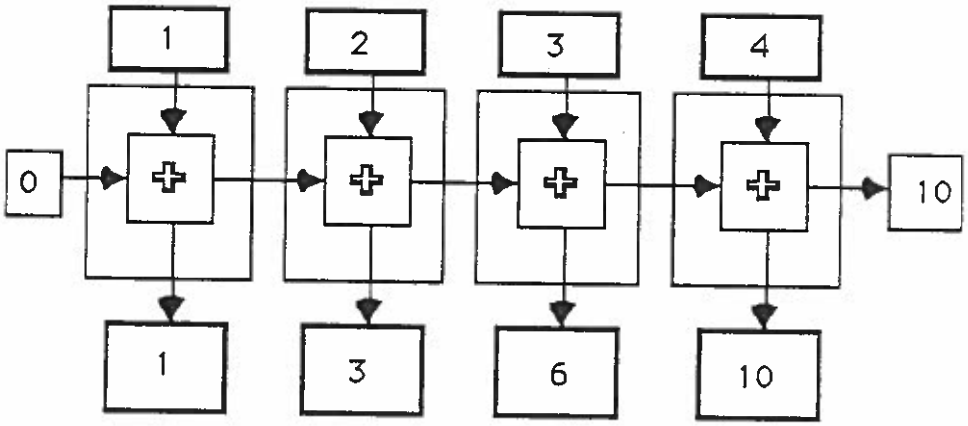


Figure 15: Conceptual Expansion of Figure 14

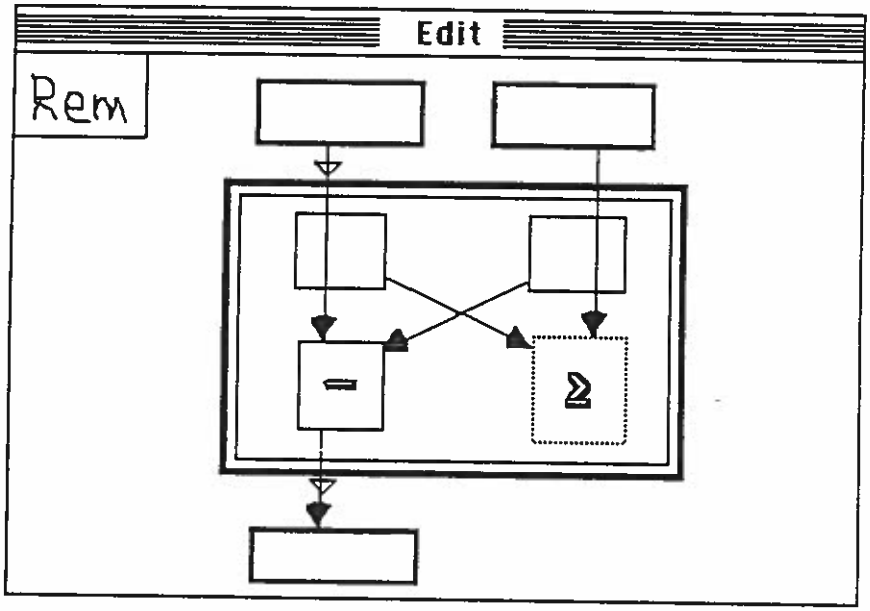


Figure 16: An STL puzzle defining the remainder function

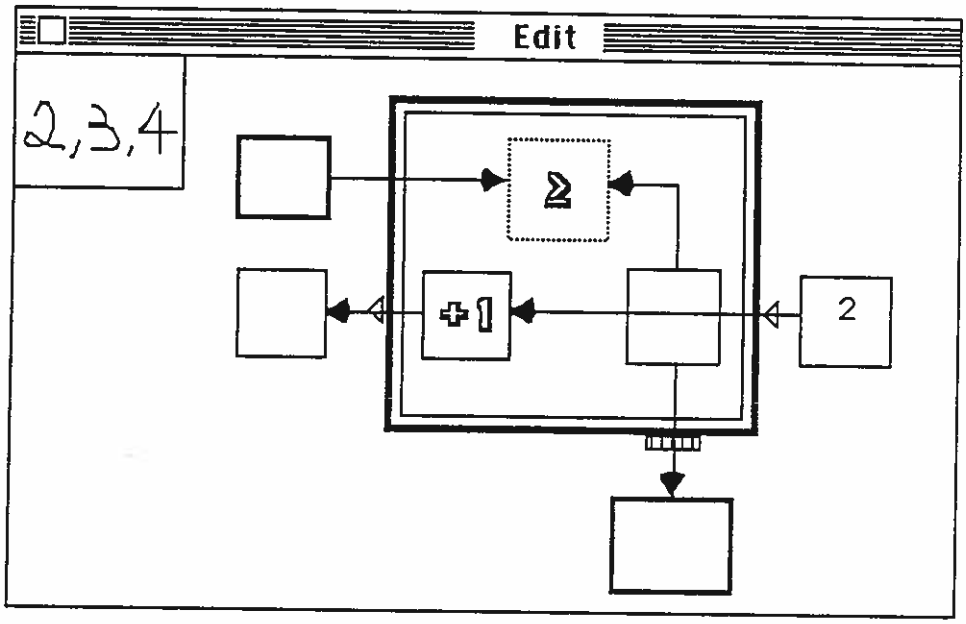


Figure 17: A sequence generator

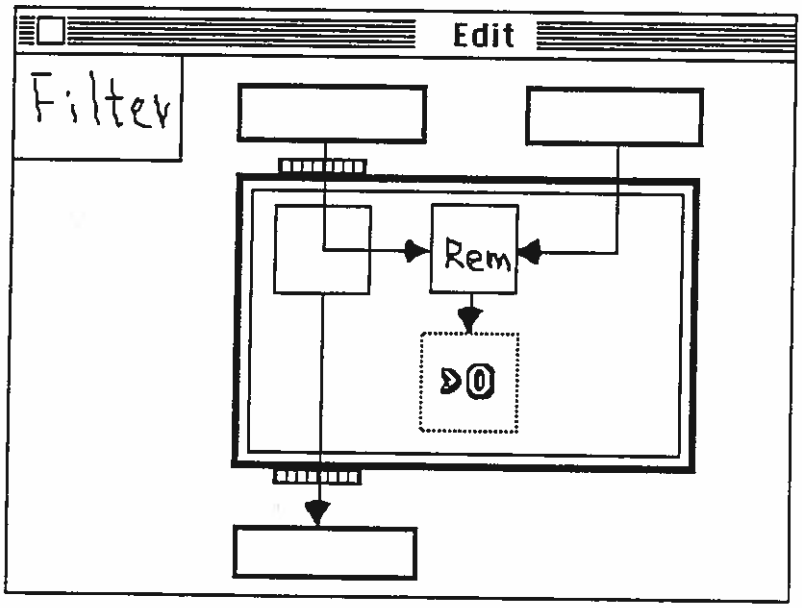


Figure 18: A filter definition

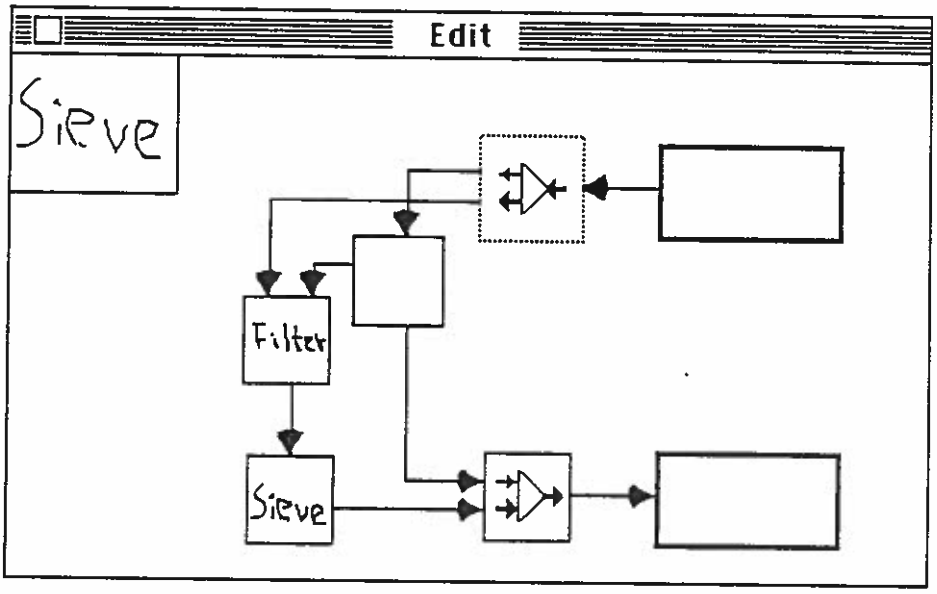


Figure 19: A definition of the sieve program

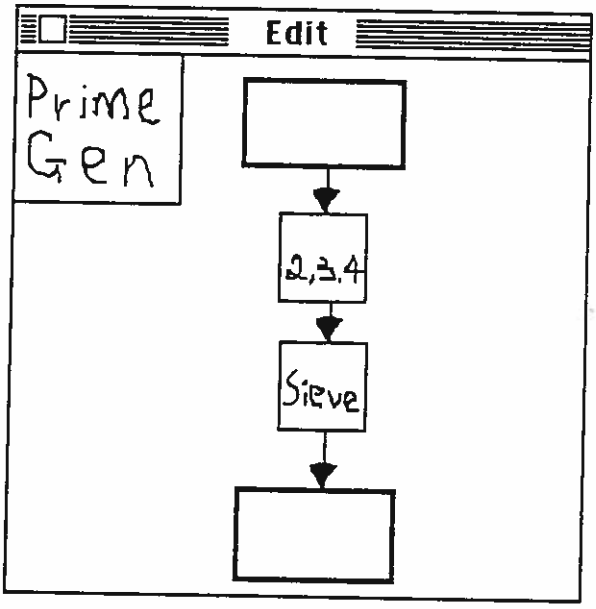


Figure 20: A prime number generator

4. THE SHOW AND TELL COMPILER

4.1 OVERVIEW OF COMPILER BASED SYSTEM

Currently, the Show and Tell Language system is interpreter based. The system can be divided into four main parts, the event recognizer, the screen manager, the editor and the interpreter. The event recognizer is the driver of the system which gathers all the events generated by the user or the system, analyzes them and then takes the appropriate actions. The screen manager manages the updating and refreshing of the various windows on the screen. It works very closely with the event recognizer and the editor. The editor manages the puzzles' construction. Besides calling the screen manager to update the puzzle drawn on the screen, the editor's main function is to update and maintain the internal representation of puzzles. The interpreter is divided into two parts, the scheduler and the execution manager. Since one or more puzzles can be solved concurrently, the scheduler determines which puzzle is to be processed next. The execution manager performs the actual processing of boxes and displays both the intermediate results at break points and the final answers.

The current configuration of the system allows a flexible environment for program development. Puzzles can be edited and solved for immediate feedback since the system is interpreter based. However, in order to achieve higher performance, a compiled version of the language is needed. Figure 21 is an overview of the system with an STL compiler integrated.

In this system, a user can use the interpreter during program development and the compiler after debugging is finished. The execution system provides a run time environment for the compiled programs. Its main functions are to link and load the executable module into memory as well as supplying library routines during execution.

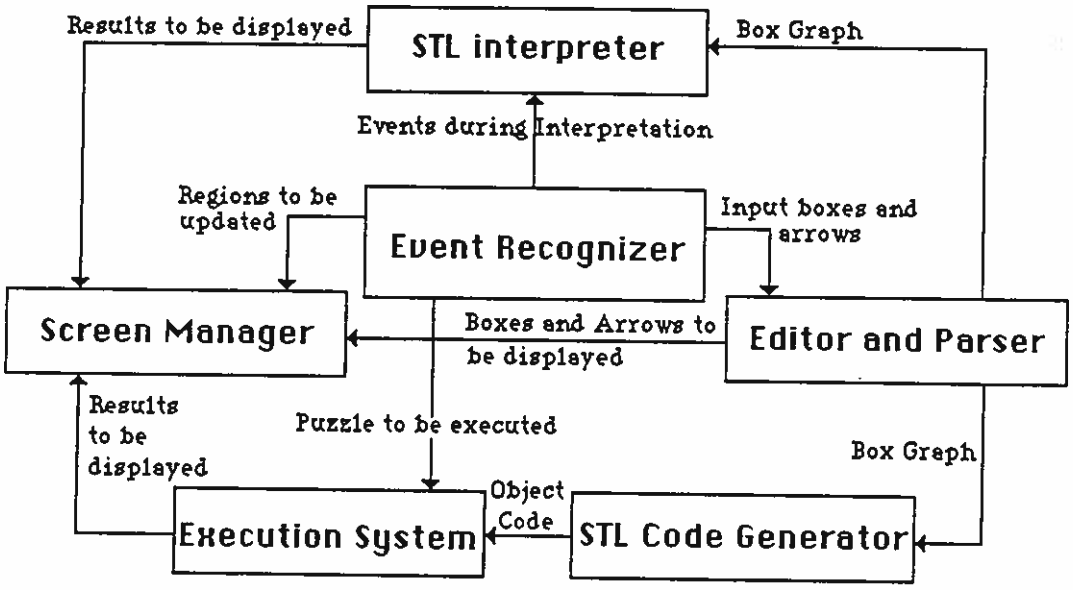


Figure 21: Overview of the STL system with a compiler integrated

Currently, the execution system is not implemented. Section 5.1 contains a description of a system to test the correctness and performance of the code generated by the compiler.

4.2 DEFINITION OF SIMPLE SHOW AND TELL LANGUAGE (SSTL)

Not all the constructs defined in section 3.2 are accepted by the current compiler. The database constructs which include the file box and the structure box, and all input/output primitives are excluded from the current implementation because they are machine dependent. This restricted STL is referred to as the Simple Show and Tell Language (SSTL). The syntax rules are the same as in STL. Figure 22 shows the lexical elements for the SSTL.

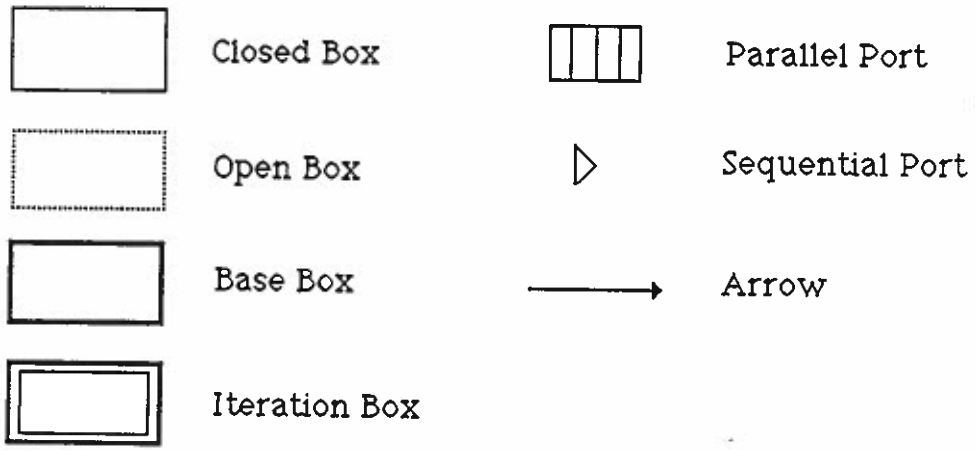


Figure 22: Lexical Elements of SSTL

4.3 OVERVIEW OF THE SSTL COMPILER

The SSTL Compiler has three main parts: the parser, the semantic checker and scheduler, and the code generator. Figure 23 is an overview of the compiler. The parser is an incremental parser which checks whether the combination of boxes and arrows entered so far represent a syntactically correct puzzle or not. It is the same parser used in the interpreter based system. Any new box or arrow entered by the editor will be rejected immediately if the resulting puzzle is not syntactically correct. The output of the parser is a data structure called a *box graph*. The box graph will pass through the semantic analyzer to determine whether the puzzle is semantically meaningful.

Because of data dependency, some boxes have to be executed before others. The scheduler determines which box should be compiled (and therefore executed at run-time) before which box. The output of the scheduler is a box sequence which is ordered according to their execution sequence. The final stage of compilation is the code generator. The code generator takes the box graph and the box sequence and generates C code as output. No code optimization is incorporated. Appendix 7.1 gives a complete example with the source puzzle, output of the parser, output of the scheduler and the output of the code generator.

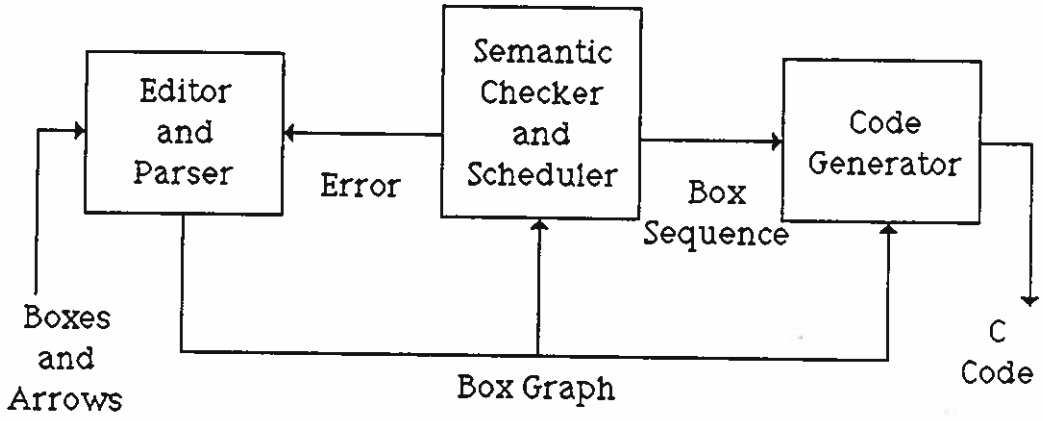


Figure 23: Overview of SSTL Compiler

4.3.1 Parser

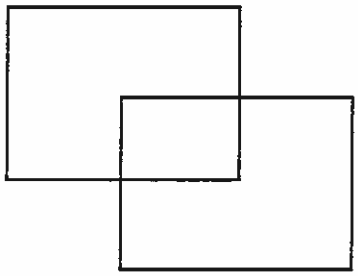
The Parser is the front end of the compiler. It works on a data structure called a *box graph*. A box graph contains four main structures: *box structure*, *arrow structure*, *segment structure* and the *dot structure*. A dot is an intersection of a box and an arrow, and an arrow is composed of a number of straight line segments. For a detailed example of a box graph, refer to Appendix 7.1. The following are some of the major structural components of a boxgraph:

```
type BOX      is pointer to Box_structure
type ARROW    is pointer to Arrow_structure
type DOT      is pointer to Dot_structure
type SEGMENT  is pointer to Segment_structure
type POINT    is pointer to Point_structure

Record Point_structure:
        int     x_coordinate;
        int     y_coordinate;
```

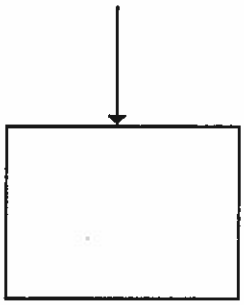
```
Record Box_structure:
        POINT       top_left;       -- coordinate of top_left corner
        POINT       bottom_right;   -- coordinate of bottom_right corner
        DOT         inport;         -- the first intersection of the box with
                                       an incoming arrow
        DOT         outport;        -- the first intersection of the box with
                                       an outgoing arrow
        BOX         son;            -- the biggest box enclosed by the box
        BOX         dad;            -- the immediate box enclosing it
        BOX         brother;        -- a neighbor with the same dad
        int         boxtype;        -- the type of the box; it can be
                                       simple box, base box, file box
                                       iteration box or structure box
        int         boxscope;       -- denotes whether the box is an
                                    -- open or closed box

Record Arrow_structure:
        DOT         first_dot;      -- the first intersection of the arrow
                                       with a box
        SEGMENT     first_segment;  -- the first segment of the arrow
        ARROW       next_arrow;     -- next arrow
        ARROW       pre_arrow;      -- previous arrow

Record Dot_structure:
        POINT       location;       -- coordinate of the point
        BOX         onbox;          -- the box where the dot is on
        ARROW       onarrow;        -- the arrow where the dot is on
        int         porttype;       -- indicate whether the dot is an
                                       inport or outport.
        DOT         next_dot;       -- next dot on the same arrow
        DOT         pre_dot;        -- previous dot on the same arrow
        DOT         next_port;      -- next inport or outport on the same
                                       box
        DOT         pre_port;       -- previous inport or outport on the
                                       same box

Record Segment_structure:
        POINT       start_point;    -- coordinate of the point where the
                                       segment starts
        POINT       end_point;      -- coordinate of the point where the
                                       segment ends
        SEGMENT     next_segment;   -- next segment on the same arrow
```

There are two main error checking modules in the parser, the check_box module and the cycle_detection module. The check_box module determines whether a newly entered box overlaps or touches any existing box. If it does, then it will be rejected. The cycle_detection module will be invoked each time a new box or new arrow is entered. It checks every path starting from the root nodes. A list of nodes for each path is generated.

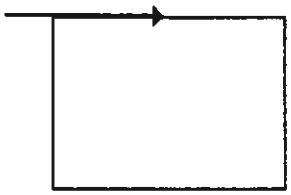
If a node on a path is visited twice, a cycle is found. Figure 24 shows some of the invalid puzzles.



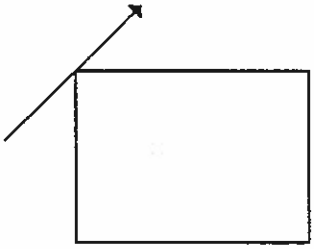
Overlapping boxes



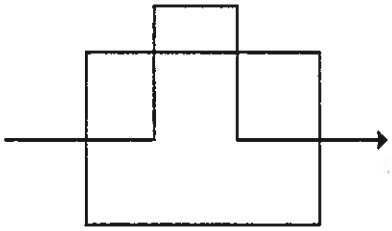
Arrow with no source box



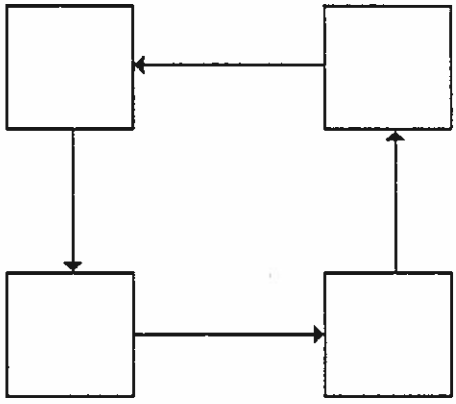
Line on the edge of a box



Line cutting the corner of a box



Self Loop



General Loop

Figure 24: Syntactically incorrect puzzles

4.3.2 Semantic Checker And Scheduler

The parser can only check for syntactic errors. But some combinations of boxes and arrows will not make a meaningful puzzle. The semantic analysis detects and reports meaningless puzzles. For example, a puzzle may not be meaningful because:

(1) More than one arrow goes through a serial port.

(2) No arrow goes through a serial port.

(3) No arrow goes through a parallel port.

The semantic analysis reports all these errors before execution.

Note that an STL puzzle is a partially ordered set of boxes. Because of the data dependency, some boxes have to be executed before others. The scheduler will generate a box sequence for the code generator so that the execution sequence is the same as the compilation sequence. It schedules the compilation of boxes in depth-first manner for boxes of different levels (i.e. nested) and topological sort for boxes of the same level. The meaning of level can be seen in Figure 25.

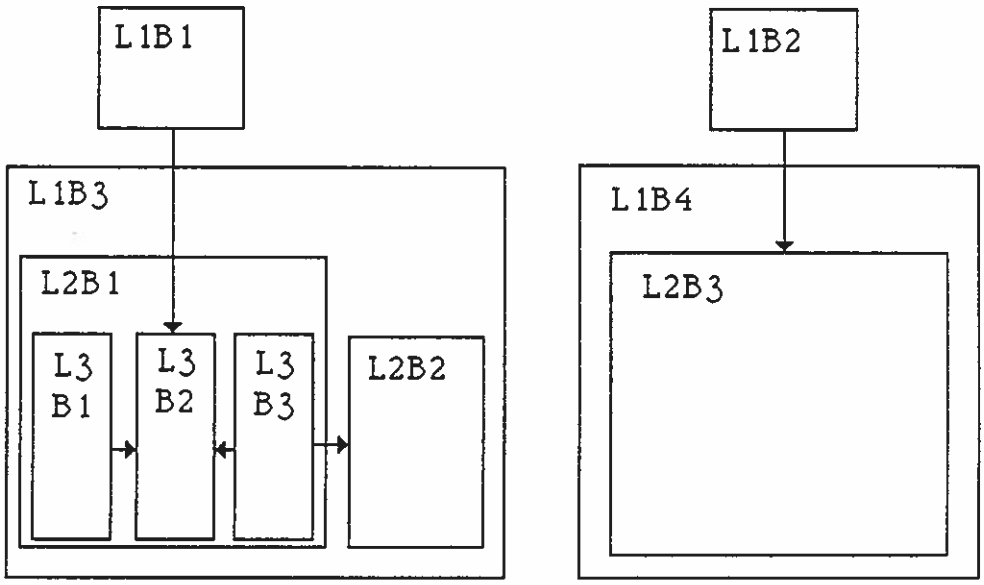


Figure 25: Levels in Show and Tell Puzzle

Each box enclosing other boxes represents a new level. For level 1, the data dependencies are L1B3 depends on L1B1, and L1B4 depends on L1B2. Therefore, the box sequences of level 1 can only be:

```
L1B1 L1B3 L1B2 L1B4   or   L1B1 L1B2 L1B3 L1B4
L1B1 L1B2 L1B4 L1B3   or   L1B2 L1B4 L1B1 L1B3
L1B2 L1B1 L1B4 L1B3   or   L1B2 L1B1 L1B3 L1B4
```

A box is not finished processing until all the boxes it encloses are processed. Therefore, boxes of different levels are processed in a depth-first manner. For example, the sequence L1B1 L1B3 L1B2 L1B4, between L1B3 and L1B2, all boxes enclosed by L1B3 have to be inserted. Combining the two conditions, some of the sequences generated for Figure 25 are:

```
L1B1 L1B3 L2B1 L3B3 L3B1 L3B2 L2B2 L1B2 L1B4 L2B3   or
L1B1 L1B2 L1B3 L2B1 L3B1 L3B3 L3B2 L2B2 L1B4 L2B3   or
L1B1 L1B2 L1B4 L2B3 L1B3 L2B1 L3B1 L3B3 L3B2 L2B2   or
L1B2 L1B4 L2B3 L1B1 L1B3 L2B1 L3B1 L3B3 L3B2 L2B2   or
L1B2 L1B1 L1B4 L2B3 L1B3 L2B1 L3B1 L3B3 L3B2 L2B2   or
L1B2 L1B1 L1B3 L2B1 L3B1 L3B3 L3B2 L2B2 L1B4 L2B3   or
    ...
```

4.3.3 Code Generator

The output of the parser is a valid box graph, and the output of the scheduler defines one sequence of the boxes from the box graph. The code generator takes this information and generates specific code. The code generator works closely with a set of templates. Each main template is a translation rule which is called for a particular box type with a particular box value. Other templates will be called by the main templates for further translations. The compiler can generate codes other than C code by redefining the set of templates.

The templates provide a way of changing the desired output without changing the compiler. The code generator analyzes the box type and the box value to determine which templates should be used and what parameters should be given to the templates.

Currently, the output of the compiler is not integrated with the system's screen display management; this is an experimental compiler for testing whether the translation rules are correct and sufficient. A work bench is devised to test the code generation part of the compiler. This is explained in section 5.1. The activations of the screen display manager can be embedded in the templates if future integration should be needed.

4.3.4 Inconsistency

Inconsistency is one of the most difficult and important concepts in the SSTL. A portion of the codes will be skipped for execution when inconsistency is found in a particular part. One of the main concerns of code generation is to make sure that inconsistency is properly represented in the object language.

In this compiler, an analogy between inconsistency and the concept of exception is made. The compiler translates inconsistency into an exception handling mechanism. In Ada®*, when an exception is raised, the control is automatically turned to the exception handling part. The C language does not have an exception handling capability, but with a package of macros and routines, it can be implemented. A detailed description of how it can be done is given in (13). The basic structure of a C program block containing exception handlers is shown as the Figure 26.

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

```
NEW_EXCEPTION(exception_name1);      /* declaration of exception    */
NEW_EXCEPTION(exception_name2);
BEGIN                                /* normal C code               */
        ...
        ...
EXCEPTION
        WHEN(<exception_name1>)      /* exception handler 1         */
                ...
        WHEN(<exception_name2>)      /* exception handler 2         */
                ...
END
```

Figure 26: A C Program Block Containing Exception Handlers

The NEW_EXCEPTION, BEGIN, EXCEPTION, WHEN and END are macros defined in the exception package. The exception_name should be declared in the program before it is used. Exception names are global throughout the system and there is only one exception name, "inconsistent". Exceptions can be propagated to the calling routine, as in Ada.

Each complex (a box containing other boxes) closed box is translated to a new exception block. Open complex boxes do not open up a new exception block because when an open box becomes inconsistent, the inconsistency propagates to the surrounding box. In the object code, the behavior of open box is retained by putting the code for the open box within the same block as its parent box. For the example in Figure 25, the skeleton of the output for the box sequence in Figure 27a is shown in Figure 27b, using exception blocks to denote inconsistency. (operations in italics, dead and eliminate, are to be further expanded.) If L2B1 is an open box, the skeleton of the output would be Figure 27c.

L1B1 L1B3 L2B1 L3B1 L3B3 L3B2 L2B2 L1B2 L1B4 L2B3

Figure 27a: A Box Sequence

```
do L1B1
BEGIN
        BEGIN
                do L3B1
                do L3B3
                do L3B2
        EXCEPTION
                WHEN(inconsistent)
                        eliminate  L2B1, L3B1, L3B3, L3B3
        END
        do L2B2
EXCEPTION
        WHEN(inconsistent)
                eliminate L1B3, L2B1, L2B2, L3B1, L3B2, L3B3
END
do L1B2
BEGIN
        do L2B3
EXCEPTION
        WHEN(inconsistent)
                eliminate L1B4, L2B3
END
```

Figure 27b: Skeleton of Output For Figure 25

```
do L1B1
BEGIN
        do L3B1
        do L3B3
        do L3B2
        do L2B2
EXCEPTION
        WHEN(inconsistent)
                eliminate L1B3, L2B1, L2B2, L3B1, L3B2, L3B3
END
do L1B2
BEGIN
        do L2B3
EXCEPTION
        WHEN(inconsistent)
                eliminate L1B4, L2B3
END
```

Figure 27c: Skeleton of Output when L2B1 is an Open Box

The analogy between inconsistency and the concept of exception contributes to the simplicity of the code generation. We conjecture that if this analogy were not made, the code generation would be more difficult to implement.

4.3.5 Translation Schema

Translation in this compiler is the process of generating C object codes from a box graph. The compilation model is arrow based. Values of computations are stored in arrows, and data for the computation can either be in the boxes or in the arrows. Each arrow represents a communication path between boxes, both for data and consistency information. Therefore, an arrow has two main pieces of information associated with it, its *value*, the data information and its *status*, the consistency information. The value of an arrow is a tagged data item which can be text, number, image or user defined icon. The status of an arrow denotes whether the arrow starts from or passes through any box which is inconsistent. If an arrow does lie on an inconsistent box, the status is *dead*. Otherwise, it is *running*. The value and status of an arrow are represented by *avalue[arrownumber]* and *astatus[arrownumber]*, respectively, in the object code. An arrow which goes through a port (sequential or parallel), there is an additional value attached to it, represented by *aportval[arrownumber]* . It is used to keep track of values in the previous stages of iteration.

Each SSTL puzzle is a function in the object code. In order to handle user defined and system defined functions in a uniform way, all functions have one input and one output parameter, which is a list of the actual input and output parameters. Argument passing is accomplished by two variables in the object code, *tempin* and *tempout*. Tempin is a list of all the incoming values of the box being translated. It will be used as the list of input arguments for both system defined and user defined functions. Tempout stores the list of output values returned by any system defined or user defined function.

In SSTL, all user defined functions are named by an icon. However, in order to represent a subroutine in C, an identifier is needed that corresponds to the icon. When a puzzle is constructed, the editor gives the icon a time stamp as its icon_id. The compiler uses this time stamp as the identifier for that icon.

To complete translation, two more pieces of codes, besides the translated code for each box, have to be attached onto the object code. The first one is the function and variables declarations, and the initialization of variables. The second one is the preparation of return values and the deletion of local variables. Function declarations include definition of user defined functions called by the puzzle being translated. Variable declarations include both global and local declarations. Global variables are *boxvalue* (values inside each box), *TTLbox* (total number of boxes in the box graph), *TTLin* (total number of input boxes), *TTLout* (total number of output boxes), *TTLarw* (total number of arrows in the box graph), and *boxinit* (a boolean flag signal whether all the box values are initialized). Local variables are *STinbox* (the values of the input boxes), *SToutbox* (the values of the output boxes), *avalue*, *astatus*, *aportval*, *tempin*, *tempout* and *STreturn* (store the function return values). The initialization part will give initial values to all the variables declared.

STreturn is a list storing all the output values. If a box is an output base box, the value of the box after execution will be stored in SToutbox. Therefore, all the output values can be collected from SToutbox at the end and put them in STreturn. Deletion of local variables include deleting the values in avalue, aportval, STinbox, SToutbox, tempin and tempout.

4.3.6 Value Manager

A package is used throughout the translated code. It is the value manager(vm) package which is used to manipulate all data objects created during execution of the compiled program. The following are the structures declared in the package:

```
type VALUE   is pointer Value_structure
type LHEAD   is pointer List_structure
type VLIST   is pointer Vlist_structure
type TEXT    is pointer Text_structure
type IMAGE   is pointer Image_structure
type PUZZLE  is pointer Puzzle_structure
                              -- Text, Image, Puzzle structure are  defined in
                                 other packages.
Record VList_structure
        VALUE       val;
        VLIST       next;

Record List_structure
        VLIST       first;
        VLIST       last;
        VLIST       lastvisit;
        int         count;

Record Value_structure
        int         ref_count;
        int         value_type;
        case  value_type is
                TEXTDATA:       TEXT        dataitem;
                IMAGEDATA:      IMAGE       dataitem;
                NUMBERDATA:     float       dataitem;
                LISTDATA:       VHEAD       dataitem;
                NULLDATA:       int         dataitem;
                PUZZLEDATA:     PUZZLE      dataitem;
        end case
                              -- Value_Structure is a variant record. The dataitem
                                 can either be of the type TEXT, IMAGE, float,
                                 VHEAD, int or PUZZLE depending on the tag
                                 value_type.
```

Only the type VALUE can be used by a user. All the other types are private to the vm package. The operations on VALUE are the following:

```
vm_Delete(VALUE)          -- delete a value
vm_Copy(VALUE)            -- make a copy of the value
vm_ListVal()              -- create an empty list value
vm_Add(VALUE, VALUE)      -- add the value in the second argument onto the list
                             value in the first argument
```

```
vm_GetNext(VALUE)        -- get the next value in the list. The lastvisit field
                            stores which element was used previously.
vm_Reset(VALUE)          -- reset the lastvisit so that the next call to GetNext
                            will get the first element in the list.
vm_NumberVal(float)      -- create a new number value
vm_TextVal(TEXT)         -- create a new text value
vm_PuzzleVal(PUZZLE)     -- create a new puzzle value
vm_ImageVal(IMAGE)       -- create a new image value
vm_NullVal()             -- create a new empty value
```

A complete set of templates is given in Appendix 7.2. Figure 28 is a simple example to explain how the templates are used in translation.

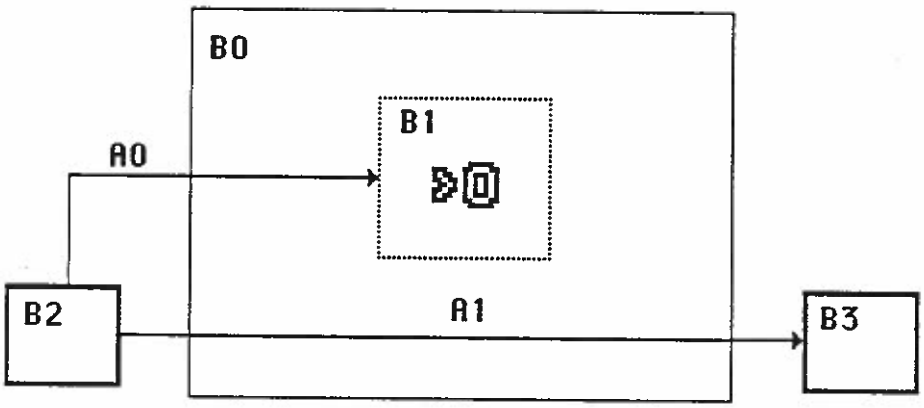


Figure 28: An example STL puzzle

Box sequence generated by the scheduler : B2 B0 B1 B3

For B2, since it is a base box, rules R1, R11 and R12 are used (All rules are defined in Appendix 7.2). After applying the rules, the annotated output is:

```
vm_Delete(tempin);
tempin = vm_ListVal();
vm_Delete(tempout);
tempout = NULL;
tempout = doprimitive(STinbox[0], tempin);      -- Since B2 is an input box,
                                                   the value in STinbox is
                                                   used instead of using the
                                                   value in boxvalue[2]

vm_Reset(tempout);
vm_Delete(avalue[0]);
avalue[0] = vm_Copy(vm_GetNext(tempout));
```

```
astatus[0] = running;
vm_Delete(avalue[1]);
avalue[1] = vm_Copy(vm_GetNext(tempout));
astatus[1] = running;
```

For B0, rules R6, R13 and R14 are used; the output is (characters in bold are to be expanded by another rule) :

```
BEGIN
        <ProcessSon(SonSize,  SizeNumberList)>
EXCEPTION
        WHEN(inconsistent)
                astatus[1] = dead;
END
```

For B1, rules R4, R11, and R12 are used; the output combining with the output of B0 is:

```
BEGIN
vm_Delete(tempin);
tempin = vm_ListVal();
vm_Delete(tempout);
tempout = NULL;
if(astatus[0] != dead)
        vm_Add(tempin,vm_Copy(avalue[0]));
tempout = doprimitive(boxvalue[1], tempin);
vm_Reset(tempout);
EXCEPTION
        WHEN(inconsistent)
                astatus[1] = dead;
END
```

Finally, for B3, rules R1, R11 R12 are used; the annotated output is:

```
vm_Delete(tempin);
tempin = vm_ListVal();
vm_Delete(tempout);
tempout = NULL;
if(astatus[1] != dead)
        vm_Add(tempin,vm_Copy(avalue[1]));
tempout = doprimitive(boxvalue[3], tempin);
vm_Reset(tempout);
STboutbox[0] = vm_Copy(vm_GetNext(tempout));    -- Since B3 is an output
                                                   box, the value of B3 is
                                                   also stored in STboutbox.
```

Combining the output of each box and the pieces of code for declarations and return, the final annotated output for the puzzle in Figure 28 is :

```
#include "compile.h"                            -- header file for vm package
        static VALUE  boxvalue[4];              -- total # of box = 4
        static int              TTLbox = 4;
        static int              TTLin = 1;      -- total # of in box = 1
```

```
static int              TTLout = 1;          -- total # of out box = 1
static int              TTLarw = 2;          -- total # of arrows = 2
static int              boxinit = 0;
VALUE ST2532(STarg)
VALUE STarg;
{
        VALUE       STinbox[1];
        VALUE       STоutbox[1];
        VALUE       avalue[2];
        VALUE       aportval[2];
        int         astatus[2];
        VALUE       tempin;
        VALUE       tempout;
        VALUE       STreturn;
        int     STindex;
                                                -- initializing arrow values
        for(STindex = 0; STindex < 2; STindex++) {
                avalue[STindex] = NULL;
                aportval[STindex] = NULL;
                astatus[STindex] = running;
        }
                                                -- initializing box values
        if(!boxinit) {
                STboxinit(boxvalue, "ST2532_box", TTLbox);
                boxinit = 1;
        }
        vm_Reset(STarg);
                                        -- getting arguments into in boxes
        for(STindex = 0; STindex < TTLin; STindex++)
                STinbox[STindex] = vm_Copy(vm_GetNext(STarg));
        for(STindex = 0; STindex < TTLout; STindex++)
                STоutbox[STindex] = NULL;
        tempin = NULL;
        tempout = NULL;
        STreturn = NULL;
/* Box 2 */
        vm_Delete(tempin);
        tempin = vm_ListVal();
        vm_Delete(tempout);
        tempout = NULL;
        tempout = doprimitive(STinbox[0], tempin);
        vm_Reset(tempout);
        vm_Delete(avalue[0]);
        avalue[0] = vm_Copy(vm_GetNext(tempout));
        astatus[0] = running;
        vm_Delete(avalue[1]);
        avalue[1] = vm_Copy(vm_GetNext(tempout));
        astatus[1] = running;
/* Box 0 */
        BEGIN
/* Box 1 */
        vm_Delete(tempin);
```

```
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[0] != dead)
                        vm_Add(tempin,vm_Copy(avalue[0]));
                tempout = doprimitive(boxvalue[1], tempin);
                vm_Reset(tempout);
                EXCEPTION
                        WHEN(inconsistent)
                                astatus[1] = dead;
                END
/* Box 3 */
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[1] != dead)
                        vm_Add(tempin,vm_Copy(avalue[1]));
                tempout = doprimitive(boxvalue[3], tempin);
                vm_Reset(tempout);
                SToutbox[0] = vm_Copy(vm_GetNext(tempout));
                STreturn = vm_ListVal();
                                                -- preparing return values
                for(STindex = 0; STindex < TTLout; STindex++)
                        vm_Add(STreturn, vm_Copy(SToutbox[STindex]));
                                                -- deleting local variables
                STarwdelete(avalue, TTLarw);
                STarwdelete(aportval, TTLarw);
                STboxdelete(STinbox, TTLin);
                STboxdelete(SToutbox, TTLout);
                vm_Delete(tempin);
                vm_Delete(tempout);
                return(STreturn);
        }
```

Appendix 7.1 contains a full example of the box graph, the box sequence generated by the scheduler and the output of the code generator.

5. EVALUATION AND CONCLUSION

5.1 TESTING SET-UP

Figure 29 is the system currently used to give the performance measurements of the compiled version of an STL puzzle.

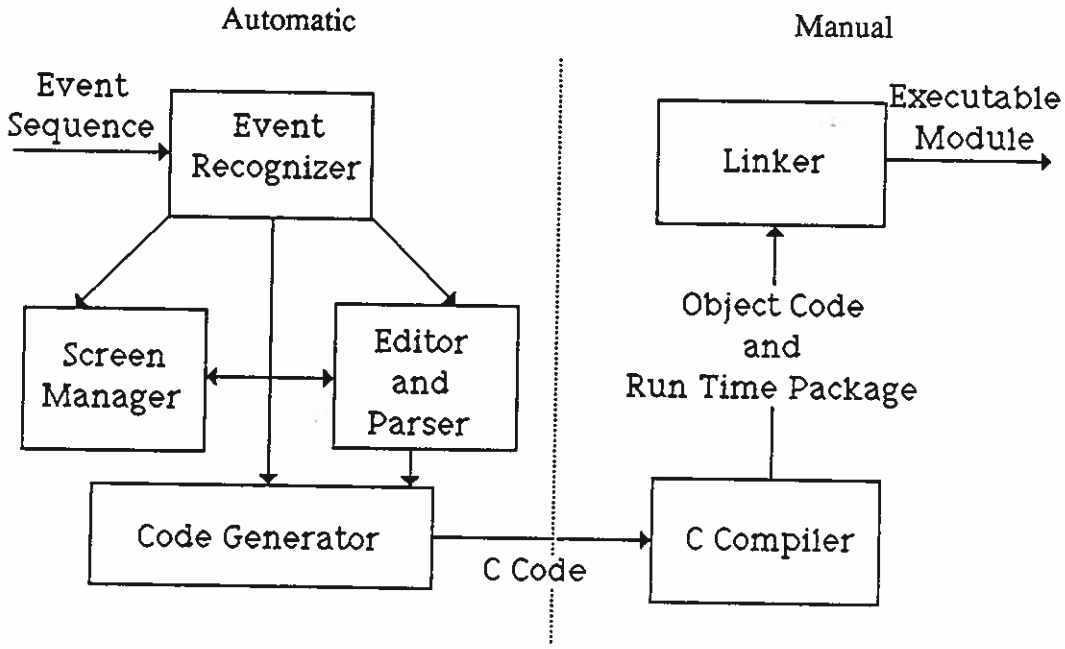


Figure 29: Configuration of the STL system used in Performance measurement

The event recognizer, the screen manager and the editor are the same as in the interpreter based system. The code generator outputs object code in the C language. The C language was chosen for experimentation purpose only. Other languages could be used instead. The C code generated can be compiled by any regular C compiler and the object code can be linked with a run time package to generate an executable module. The run time package provides support for system defined operations. Eventually, the manual portion of the system will be eliminated and will be replaced by the execution system described in section 4.1. At that time, the compilation and the linkage of object code is automatic and load modules can be directly executed in the STL system.

5.2 PERFORMANCE MEASUREMENT

Three puzzles are used to compare the performance between the interpreter based version and the compiler based version. The first puzzle is the definition Fibonacci numbers defined in Figure 12. The second puzzle is the factorial program defined in Figure 7.1 and the third puzzle is the prime number generator defined in Figure 20. This puzzle uses puzzles defined in Figures 16 to 19. These three programs are chosen because they use the iteration construct alone, recursion alone and the combined usage of iteration and recursion.

For analysis purpose, the screen managements of the interpreter are turned off because the compiled versions of the puzzles omit all invocations to the screen manager. Figure 30 through 32 summarize the comparison of the interpreter vs compiler version of the system for various input parameters.

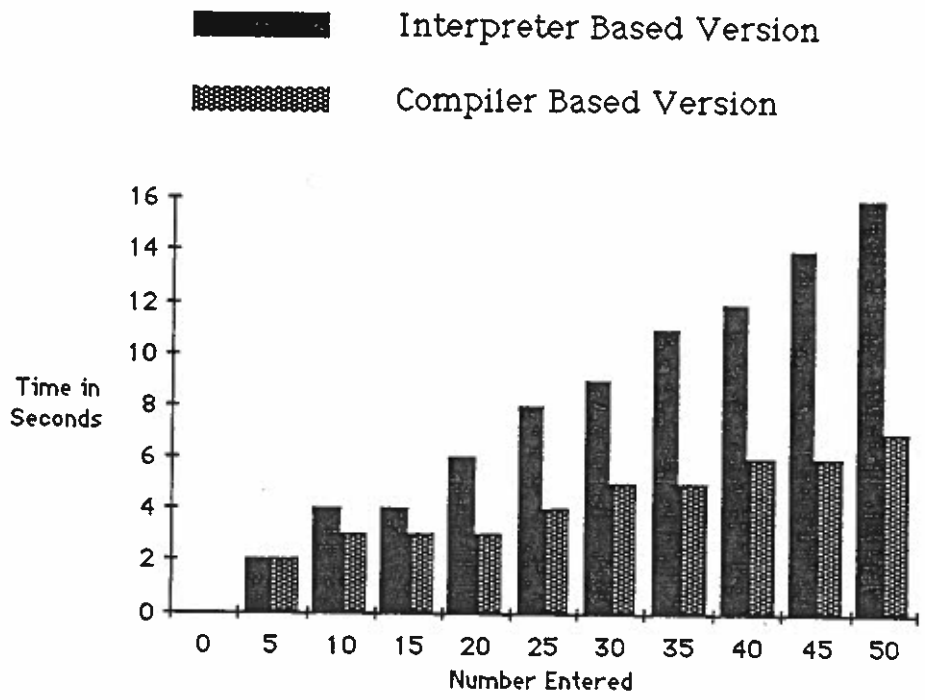


Figure 30: Performance Comparison of the Fibonacci Program

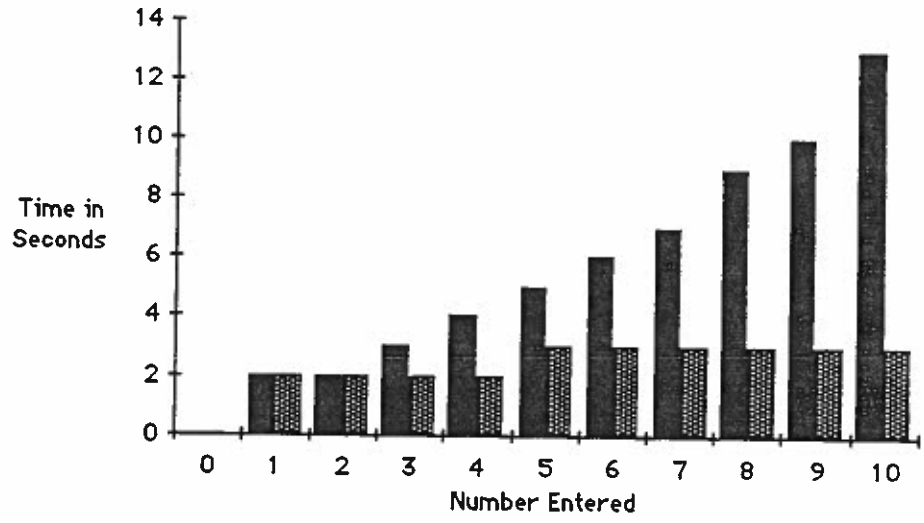


Figure 31: Performance Comparison of the Factorial Program

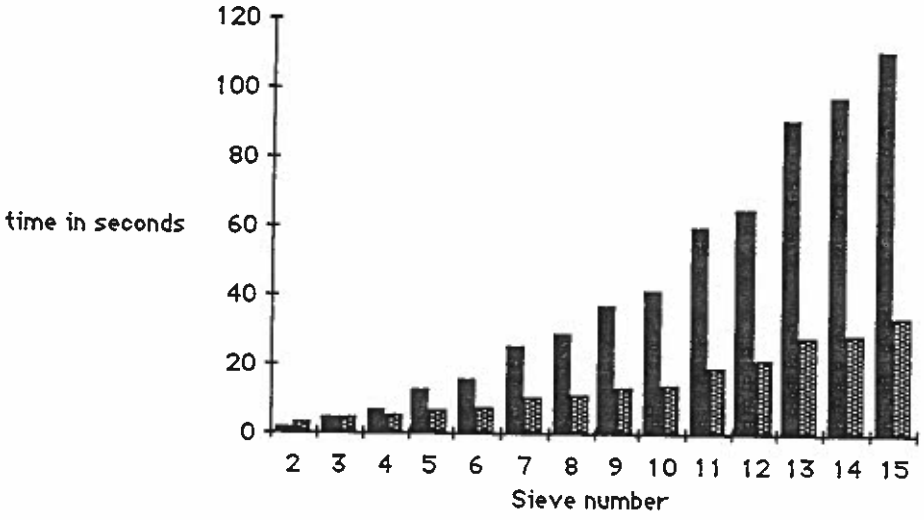


Figure 32: Performance Comparison of The Prime Number Generator

The compiler can produce a speedup factor of about 2 or 3 depending on the program being run and the input. In the compiled version of an STL program, there is an overhead during start up for initializing global variables. However, the true nature of the

differences between the compiled version and the interpreted version become apparent as larger numbers are used in the comparison.

In common programming languages, the speedup of a compiled version over an interpreted version is usually more than the STL compiler currently achieves. There are a number of reasons why the speedup is not as much as expected:

(1) The object code of the compiler is in the C language. There is overhead involved in C subroutine calls. A compiler generating assembler code reduces this problem.

(2) The code generated by the compiler is not optimized. For example, in all the rules, the call to <GenerateInputList> can be eliminated when the box being translated has no input arrow.

(3) The box graph is not analyzed to determine the real intention of the graph before code generation. This leads to the structural resemblance of the output (the C code) with the input (the boxgraph). In many situations, especially in scientific programming, a STL program can be expressed with one or two expressions in a one-dimensional program. For example, the factorial function can be expressed as :

fact(x) := 1 for x = 0

fact(x) := x * fact(x - 1) for x > 0

Therefore, a box graph structure can usually be broken up to determine the linguistic meaning which can in turn be expressed in other programming language more efficiently.

5.3 CONCLUSION

In this thesis, a new picture language is presented, in which programs are expressed in two-dimensional structures and data can either be text, numbers or images. Except for entering textual and numeric data, a program can be created solely by using a

pointing device such as a mouse. The language can express high-level constructs such as recursion, iteration, parallel processing and exception handling.

The problem and solution of compiling from a picture language into a one-dimensional language is shown. The first version of the compiler was implemented on a Macintosh™* and its performance was measured. The object code size of the different parts of the compiler are:

Parser	22K bytes
Semantic Analysis	14Kbytes
Scheduler	4K bytes
Code Generator	10K bytes
Macro Expansion Package	17K bytes

In summary, this thesis has:

(1) Introduced the major concepts of the Show and Tell Language through examples.

(2) Demonstrated how this language can be compiled into a common one-dimensional programming language. The main idea is to translate inconsistency, which is most fundamental in STL, into exception handling capability similar to that in Ada.

(3) Compared the performance between a compiled version and an interpreted version. The limitation and possible enhancements for future compiler construction has been addressed.

Future research will involve defining a formal syntax for such a two-dimensional language and a parsing algorithm based on that formalism. The parser and the semantic analysis in the STL compiler are ad hoc because of the lack of grammar for this language. There are at least two approaches in defining a syntax for a two-dimensional

*Macintosh is a trademark of Apple Computer, Inc.

programming language. The first approach is to define a two-dimensional meta-language for specifying the syntax of the object language. Lexical elements are also two-dimensional objects. The second approach is to use a known one-dimensional meta-language to specify the set of one-dimensional representations of two-dimensional program constructs. Attempts to define a formal grammar for the STL using the second approach can be found in (14) and (15).

The success of two-dimensional programming languages will depend on their availability and performance. Building a compiler for such a language will help in the performance. In the near future it is expected that a large database will become available to end users through laser disk technology. A visual programming languages such as STL may make it easier for users to extract useful information from large databases.

6. Acknowledgments

I would like to acknowledge my deepest appreciation to Dr. T.D. Kimura, my academic and thesis advisor. Without his encouragement and continuing guidance, this work could never have taken place. I also gratefully acknowledge Dr. Will. D. Gillett and Dr. G.C. Roman for being on my thesis committee and providing so many constructive comments. Dr. Gillett gave me many ideas about the code generation part of the compiler.

The compiler is a product of many people's contributions. Ms. Jane Mack implemented the screen manager, the event recognizer and part of the editor. Ms. Yukari Esman implemented the parser. The macro expansion module was implemented by Dr. Gillett.

I wish to extend my appreciation to Mr. Darren Cruse and Ms. Jane Mack for proof reading the thesis for me. Last, but not least, I wish to thank all the people, particularly Ms. Jane Mack, Ms. Yukari Esman, Mr. Darren Cruse, Mr. Kam Yuen Chan and Dr. Ken Wong, who have constantly given me support and encouragement to finish my thesis. This research was partially funded by the Computer Services Corporation (CSK), Japan.

7. APPENDICES

APPENDIX 7.1

A Complete Example

7.1.1 Input To The Editor

The event recognizer gathers all the events generated by the user or the system, analyzes them and then distributes them to different parts of the system. The event types and event sequence for the entering the puzzle of Figure 7.1 is as follows:

Types of event: box select, mousedown, mouseup, line select, single click, double click.

Events for drawing Figure 7.1:

box select, mousedown(149, 70), mouseup(305, 183)
box select, mousedown(156, 137), mouseup(196, 171)
box select, mousedown(262, 88), mouseup(296, 124)
box select, mousedown(215, 137), mouseup(251, 171)
box select, mousedown(157, 89), mouseup(192, 121)
box select, mousedown(225, 99), mouseup(240, 109)
box select, mousedown(70, 81), mouseup(136, 182)
box select, mousedown(86, 98), mouseup(121, 126)
box select, mousedown(87, 142), mouseup(120, 171)
box select, mousedown(127, 203), mouseup(200, 237)
box select, mousedown(126, 27), mouseup(198, 60)
line select, single click(198, 42), single click(233, 42), double click(233, 137)
line select, single click(232, 171), single click(232, 219), double click(219, 200)
line select, single click(102, 171), single click(102, 219), double click(127, 219)
line select, single click(126, 43), single click(103, 43), double click(103, 98)
line select, single click(175, 121), double click(175, 137)
line select, single click(196, 155), double click(215, 155)
line select, single click(225, 104), double click(104, 192)
line select, single click(104, 240), double click(104,262)

In Figure 7.1, the number on each box and arrow corresponds to the subscript on the box and arrow array in the box graph. These numbers are internal to the system and are not shown on a normal STL screen.

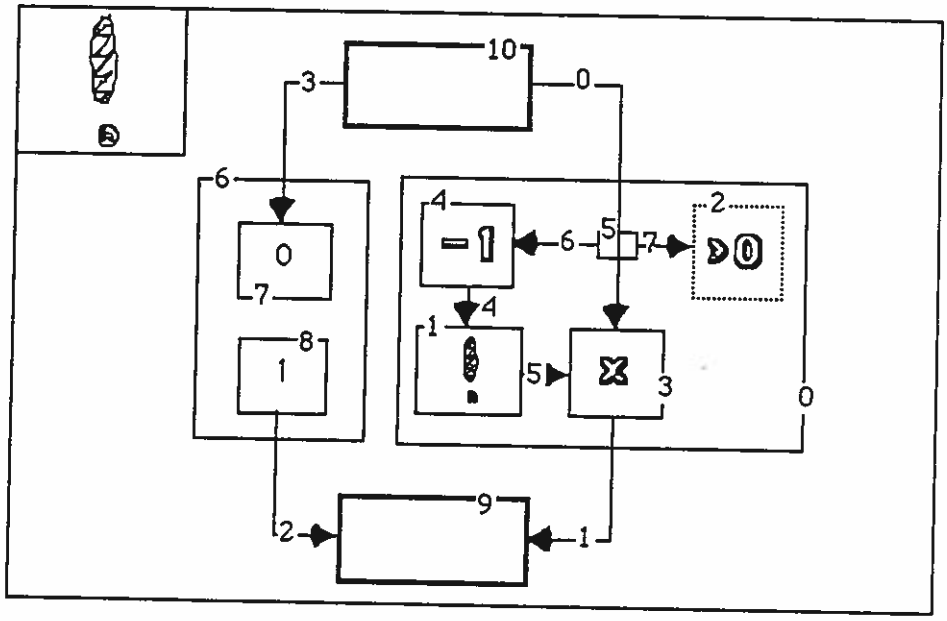


Figure 7.1: STL puzzle representing the factorial function

7.1.2 Output Of The Editor

There are three main structures in a box graph: the box array, arrow array and dot array. Each box has a list of *inports* and a list of *outports*. An inport is a dot on the intersection of a box with an incoming arrow. Similarly, an outport is a dot on the intersection of a box with an outgoing arrow. The inport and outport field of a box points to the first inport or the first outport. The next port on a box can be traced by following the next_port field of the dot. The son field of a box points to the largest box inside the box. The list of boxes inside a box can be traced by following first the son field and then the brother field of the rest. The boxtype field denotes the type of a box. In STL, the type of a box can either be simple, base, iteration, file or structure. The scope field denotes whether a box is opened or closed.

For arrow and dot, the key fields are self-explanatory, except for the porttype, next_dot and pre_dot fields. The porttype field denotes whether the dot is an inport or outport. The next_dot and pre_dot fields point to the next or previous dot on the same arrow. The following is the output of the editor.

Box Graph:

boxsize 11, arrowsize 8, dotsize 22

Box[0] : (149 79, 305 183)
inport 1, outport 6
son 1, brother 6, dad -1
Boxtype 0, scope 1

Box[1] : (156 137, 196 171)
inport 15, outport 16
son -1, brother 2, dad 0
Boxtype 0, scope 1

Box[2] : (262 88, 296 124)
inport 21, outport -1
son -1, brother 3, dad 0
Boxtype 0, scope 0

Box[3] : (215 137, 251 171)
inport 4, outport 5
son -1, brother 4, dad 0
Boxtype 0, scope 1

Box[4] : (157 89, 192 121)
inport 19, outport 14
son -1, brother 5, dad 0
Boxtype 0, scope 1

Box[5] : (225 99, 240 109)
inport 2, outport 20
son -1, brother -1, dad 0
Boxtype 0, scope 1

Box[6] : (70 81, 136 182)
inport 12, outport 9
son 7, brother 9, dad -1
Boxtype 0, scope 1

Box[7] : (86 98, 121 126)
inport 13, outport -1
son -1, brother 8, dad 6
Boxtype 0, scope 1

Box[8] : (87 143, 120 171)
inport -1, outport 8
son -1, brother -1, dad 6
Boxtype 0, scope 1

Box[9] : (127 203, 200 237)
inport 7, outport -1
son -1, brother 10, dad -1
Boxtype 7, scope 1

Box[10] : (126 27, 198 60)
inport -1, outport 0
son -1, brother -1, dad -1
Boxtype 7, scope 1

Arrow[0] : dot 0, next arrow 1, pre arrow -1
Arrow[1] : dot 5, next arrow 2, pre arrow 0
Arrow[2] : dot 8, next arrow 3, pre arrow 1
Arrow[3] : dot 11, next arrow 4, pre arrow 2
Arrow[4] : dot 14, next arrow 5, pre arrow 3
Arrow[5] : dot 16, next arrow 6, pre arrow 4
Arrow[6] : dot 18, next arrow 7, pre arrow 5
Arrow[7] : dot 20, next arrow -1, pre arrow 6

Dot[0] (198,42)
onbox 10, onarrow 0, porttype 1
next_dot 1, pre_dot -1
next_port 11, pre_port -1

Dot[1] (233,79)
onbox 0, onarrow 0, porttype 0
next_dot 2, pre_dot 0
next_port -1, pre_port -1

Dot[2] (233,99)
onbox 5, onarrow 0, porttype 0
next_dot 3, pre_dot 1
next_port -1, pre_port -1

Dot[3] (233,109)
onbox 5, onarrow 0, porttype 1
next_dot 4, pre_dot 2
next_port 18, pre_port 20

Dot[4] (233,137)
onbox 3, onarrow 0, porttype 0
next_dot -1, pre_dot 3
next_port 17, pre_port -1

Dot[5] (232,171)
onbox 3, onarrow 1, porttype 1
next_dot 6, pre_dot -1
next_port -1, pre_port -1

Dot[6] (232,183)
onbox 0, onarrow 1, porttype 1
next_dot 7, pre_dot 5
next_port -1, pre_port -1

Dot[7] (200,219)
onbox 9, onarrow 1, porttype 0
next_dot -1, pre_dot 6
next_port 10, pre_port -1

Dot[8] (102,171)
onbox 8, onarrow 2, porttype 1
next_dot 9, pre_dot -1
next_port -1, pre_port -1

Dot[9] (102,182)
onbox 6, onarrow 2, porttype 1
next_dot 10, pre_dot 8
next_port -1, pre_port -1

Dot[10] (127,219)
onbox 9, onarrow 2, porttype 0
next_dot -1, pre_dot 9
next_port -1, pre_port 7

Dot[11] (126,43)
onbox 10, onarrow 3, porttype 1
next_dot 12, pre_dot -1
next_port -1, pre_port 0

Dot[12] (103,81)
onbox 6, onarrow 3, porttype 0
next_dot 13, pre_dot 11
next_port -1, pre_port -1

Dot[13] (103,98)
onbox 7, onarrow 3, porttype 0
next_dot -1, pre_dot 12
next_port -1, pre_port -1

Dot[14] (175,121)
onbox 4, onarrow 4, porttype 1
next_dot 15, pre_dot -1
next_port -1, pre_port -1

Dot[15] (175,137)
onbox 1, onarrow 4, porttype 0
next_dot -1, pre_dot 14
next_port -1, pre_port -1

Dot[16] (196,155)
onbox 1, onarrow 5, porttype 1
next_dot 17, pre_dot -1
next_port -1, pre_port -1

Dot[17] (215,155)
onbox 3, onarrow 5, porttype 0
next_dot -1, pre_dot 16
next_port -1, pre_port 4

Dot[18] (225,104)
onbox 5, onarrow 6, porttype 1
next_dot 19, pre_dot -1
next_port -1, pre_port 3

Dot[19] (192,104)
onbox 4, onarrow 6, porttype 0
next_dot -1, pre_dot 18
next_port -1, pre_port -1

```
Dot[20] (240,104)
        onbox 5, onarrow 7, porttype 1
        next_dot 21, pre_dot -1
        next_port 3, pre_port -1

Dot[21] (262,104)
        onbox 2, onarrow 7, porttype 0
        next_dot -1, pre_dot 20
        next_port -1, pre_port -1
```

7.1.3 Output Of The Scheduler

```
B10
B0
        B5
        B2
        B4
        B1
        B3
B6
        B7
        B8
B9
```

7.1.4 Output Of The Code Generator

```
#include "compile.h"
VALUE       ST27180();
static VALUE  boxvalue[11];
static int              TTLbox = 11;
static int              TTLin = 1;
static int              TTLout = 1;
static int              TTLarw = 8;
static int              boxinit = 0;
VALUE ST27180(STarg)
VALUE STarg;
{
        VALUE       STinbox[1];
        VALUE       SToutbox[1];
        VALUE       avalue[8];
        VALUE       aportval[8];
        int         astatus[8];
        VALUE       tempin;
        VALUE       tempout;
        VALUE       STreturn;
        int     STindex;

        for(STindex = 0; STindex < 8; STindex++) {
                avalue[STindex] = NULL;
                aportval[STindex] = NULL;
                astatus[STindex] = running;
        }
        if(!boxinit) {
                STboxinit(boxvalue, "ST27180_box", TTLbox);
                boxinit = 1;
```

```
                }
                vm_Reset(STarg);
                for(STindex = 0; STindex < TTLin; STindex++)
                        STinbox[STindex] = vm_Copy(vm_GetNext(STarg));
                for(STindex = 0; STindex < TTLout; STindex++)
                        SToutbox[STindex] = NULL;
                tempin = NULL;
                tempout = NULL;
                STreturn = NULL;
/* Box 10 */
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                tempout = doprimitive(STinbox[0], tempin);
                vm_Reset(tempout);
                vm_Delete(avalue[3]);
                avalue[3] = vm_Copy(vm_GetNext(tempout));
                astatus[3] = running;
                vm_Delete(avalue[0]);
                avalue[0] = vm_Copy(vm_GetNext(tempout));
                astatus[0] = running;
/* Box 0 */
                BEGIN
/* Box 5 */
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[0] != dead)
                        vm_Add(tempin,vm_Copy(avalue[0]));
                tempout = doprimitive(boxvalue[5], tempin);
                vm_Reset(tempout);
                vm_Delete(avalue[6]);
                avalue[6] = vm_Copy(vm_GetNext(tempout));
                astatus[6] = running;
                vm_Delete(avalue[0]);
                avalue[0] = vm_Copy(vm_GetNext(tempout));
                astatus[0] = running;
                vm_Delete(avalue[7]);
                avalue[7] = vm_Copy(vm_GetNext(tempout));
                astatus[7] = running;
/* Box 2 */
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[7] != dead)
                        vm_Add(tempin,vm_Copy(avalue[7]));
                tempout = doprimitive(boxvalue[2], tempin);
                vm_Reset(tempout);
/* Box 4 */
```

```
                BEGIN
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[6] != dead)
                        vm_Add(tempin,vm_Copy(avalue[6]));
                tempout = doprimitive(boxvalue[4], tempin);
                vm_Reset(tempout);
                vm_Delete(avalue[4]);
                avalue[4] = vm_Copy(vm_GetNext(tempout));
                astatus[4] = running;
                EXCEPTION
                        WHEN(inconsistent)
                                astatus[4] = dead;
                END
/* Box 1 */
                BEGIN
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[4] != dead)
                        vm_Add(tempin,vm_Copy(avalue[4]));
                tempout = ST27180(tempin);
                vm_Reset(tempout);
                vm_Delete(avalue[5]);
                avalue[5] = vm_Copy(vm_GetNext(tempout));
                astatus[5] = running;
                EXCEPTION
                        WHEN(inconsistent)
                                astatus[5] = dead;
                END
/* Box 3 */
                BEGIN
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[5] != dead)
                        vm_Add(tempin,vm_Copy(avalue[5]));
                if(astatus[0] != dead)
                        vm_Add(tempin,vm_Copy(avalue[0]));
                tempout = doprimitive(boxvalue[3], tempin);
                vm_Reset(tempout);
                vm_Delete(avalue[1]);
                avalue[1] = vm_Copy(vm_GetNext(tempout));
                astatus[1] = running;
                EXCEPTION
                        WHEN(inconsistent)
                                astatus[1] = dead;
                END
```

```
                EXCEPTION
                        WHEN(inconsistent)
                                astatus[1] = dead;
                END
/* Box 6 */
                BEGIN
/* Box 7 */
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[3] != dead)
                        vm_Add(tempin,vm_Copy(avalue[3]));
                tempout = doprimitive(boxvalue[7], tempin);
                vm_Reset(tempout);
/* Box 8 */
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                tempout = doprimitive(boxvalue[8], tempin);
                vm_Reset(tempout);
                vm_Delete(avalue[2]);
                avalue[2] = vm_Copy(vm_GetNext(tempout));
                astatus[2] = running;
                EXCEPTION
                        WHEN(inconsistent)
                                astatus[2] = dead;
                END
/* Box 9 */
                vm_Delete(tempin);
                tempin = vm_ListVal();
                vm_Delete(tempout);
                tempout = NULL;
                if(astatus[2] != dead)
                        vm_Add(tempin,vm_Copy(avalue[2]));
                if(astatus[1] != dead)
                        vm_Add(tempin,vm_Copy(avalue[1]));
                tempout = doprimitive(boxvalue[9], tempin);
                vm_Reset(tempout);
                SToutbox[0] = vm_Copy(vm_GetNext(tempout));
                STreturn = vm_ListVal();
                for(STindex = 0; STindex < TTLout; STindex++)
                        vm_Add(STreturn, vm_Copy(SToutbox[STindex]));
                STarwdelete(avalue, TTLarw);
                STarwdelete(aportval, TTLarw);
                STboxdelete(STinbox, TTLin);
                STboxdelete(SToutbox, TTLout);
                vm_Delete(tempin);
                vm_Delete(tempout);
                return(STreturn);
        }
```

APPENDIX 7.2

A Complete Rule Set for Code Generation

The following are the translation rules. Each rule accepts a different numbers of arguments. The characters in bold are the actual output. Names in angle brackets are calls to another translation rule.

R1:
Rule Name: SimpleBox
Arguments: BoxNumber, InArrowSize, OutArrowSize, InArrowList, OutArrowList
BoxType: closed box, open box, base box
BoxContent: empty, number, text, picture
Output: < GenerateInputList(InArrowSize, InArrowList) >
if the box is an input box then output the following:
tempout = doprimitive(STinbox[InBoxNumber**],**
tempin);
else output the following:
tempout = doprimitive(boxvalue[BoxNumber**],**
tempin);

if the box is an output box then output the following:
vm_Reset(tempout);
SToutbox[OutBoxNumber**] =**
vm_Copy(vm_GetNext(tempout));
else output the following:
< AssignOutput(OutArrowSize, OutArrowList) >

R2:
Rule Name: SystemIconWithClosedBox
Arguments: BoxNumber, InArrowSize, OutArrowSize, InArrowList, OutArrowList
BoxType: closed box
BoxContent: system icon
Output:
BEGIN
< GenerateInputList(InArrowSize, InArrowList) >
tempout = doprimitive(boxvalue[boxnumber**],**
tempin)
< AssignOutput(OutArrowSize, OutArrowList) >
EXCEPTION
WHEN(inconsistent)
< MarkArrowDead(OutArrowSize, OutArrowList) >
END

R3: Rule Name: UserIconWithClosedBox
Arguments: BoxNumber, InArrowSize, OutArrowSize, InArrowList, OutArrowList, UserIconName
BoxType: closed box
BoxContent: user icon
Output:
```
BEGIN
        < GenerateInputList(InArrowSize, InArrowList) >
        tempout = UserIconName(boxvalue[boxnumber],
                tempin)
        < AssignOutput(OutArrowSize, OutArrowList) >
EXCEPTION
        WHEN(inconsistent)
            < MarkArrowDead(OutArrowSize, OutArrowList) >
END
```

R4: Rule Name: SystemIconWithOpenBox
Arguments: BoxNumber, InArrowSize, OutArrowSize, InArrowList, OutArrowList
BoxType: open box
BoxContent: system icon
Output:
```
< GenerateInputList(InArrowSize, InArrowList) >
tempout = doprimitive(boxvalue[boxnumber], tempin)
< AssignOutput(OutArrowSize, OutArrowList) >
```

R5: Rule Name: UserIconWithOpenBox
Arguments: BoxNumber, InArrowSize, OutArrowSize, InArrowList, OutArrowList, UserIconName
BoxType: open box
BoxContent: user icon
Output:
```
< GenerateInputList(InArrowSize, InArrowList) >
tempout = UserIconName(boxvalue[boxnumber], tempin)
< AssignOutput(OutArrowSize, OutArrowList) >
```

R6: Rule Name: ComplexClosedBox
Arguments: OutArrowSize, OutArrowList, SonSize, SonNumberList
BoxType: closed box
BoxContent: son boxes
Output:
```
BEGIN
        < ProcessSon(SonSize, SizeNumberList) >
EXCEPTION
        WHEN(inconsistent)
            < MarkArrowDead(OutArrowSize, OutArrowList) >
END
```

R7: Rule Name: ComplexOpenBox
Arguments: SonSize, SonNumberList
BoxType: open box
BoxContent: son boxes
Output:
```
< ProcessSon(SonSize, SonNumberList) >
```

R8: Rule Name: SeqIterationOnly
Arguments: SeqPortSize, SeqPortList, SonSize, SonNumberList
BoxType: iteration box
BoxContent: son boxes
Output:

```
BEGIN
        < FirstSeqPortInit(SeqPortSize, SeqPortList) >
        while(1) {
                < SeqPortReInit(SeqPortSize, SeqPortList) >
                < ProcessSon(SonSize, SonNumberList) >
        }
EXCEPTION
        WHEN(inconsistent)
                < EndSeqIteration(SeqPoarSize, SeqPortList) >
END
```

R9: Rule Name: SeqIterationWithParallelOut
Arguments: SeqPortSize, SeqPortList, SonSize, SonNumberList, ParOutSize, ParOutList
BoxType: iteration box
BoxContent: son boxes
Output:

```
BEGIN
        < FirstSeqPortInit(SeqPortSize, SeqPortList) >
        < FirstParOutPortInit(ParOutSize, ParOutList) >
        while(1) {
                < SeqPortReInit(SeqPortSize, SeqPortList) >
                < ProcessSon(SonSize, SonNumberList) >
                < SaveParOutValue(ParOutSize, ParOutList) >
        }
EXCEPTION
        WHEN(inconsistent)
                < EndSeqIteration(SeqPortSize, SeqPortList) >
END
< EndParIteration(ParOutSize, ParOutList) >
```

R10: Rule Name: SeqAndParIteration
Arguments: SeqPortSize, SeqPortList, SonSize, SonNumberList, ParOutSize, ParOutList, ParInSize, ParInList
BoxType: iteration box
BoxContent: son boxes
Output:

```
int  i1, i2, i3, ..... ,in;
/* depends on how many parallel port */
int  Lcount1, Lcount2, Lcount3, ... , Lcountn;
Lcount1 = vm_Count(aportval[1]);
...
Lcountn = vm_Count(aportval[n]);
< FirstSeqPortInit(SeqPortSize, SeqPortList) >
< FirstParOutInit(ParOutSize, ParOutList) >
< FirstParInInit(ParInSize, ParInList) >
for(i1 = 0; i1 < Lcount1; i1++) {
        < ParInReInit(1) >
        for(i2 = 0; i2 < Lcount2; i2++) {
```

```
                    < ParInReInit(2) >
                          .
                          .
                          .
                    for(i_n = 0; i_n < Lcount_n; i_n++) {
                        < ParInReInit(n) >
                         < SeqPortReInit(SeqPortSize, SeqPortList) >
                          BEGIN
                         <ProcessSon(SonSize, SonNumberList)>
                          <SaveParOutValue(ParOutSize,ParOutList)>
                          EXCEPTION
                                WHEN(inconsistent)
                                      ;
                          END
                    }
                .
                .
                .
                }
            }
            < EndParIteration(ParOutSize, ParOutList) >
```

R11: Rule Name: GenerateInputList
Arguments: InArrowSize, InArrowList
Output:

```
vm_Delete(tempin);
tempin = vm_ListVal();
vm_Delete(tempout);
tempout = NULL;
for each arrow  i in the InArrowList, output the following:
     if(astatus[InArrowList[i]] != dead)
          vm_Add(tempin, avalue[InArrowList[i]]);
/* generate vm_add(...) for each arrow in the list */
```

R12: Rule Name: AssignOutput
Arguments: OutArrowSize, OutArrowList
Output:

```
vm_Reset(tempout);
for each arrow i in the OutArrowList, output the following:
        vm_Delete(avalue[OutArrowList[i]]);
        avalue[OutArrowList[i]] =
                vm_Copy(vm_GetNext(tempout));
        astatus[OutArrowList[i]] = running;
```

R13: Rule Name: MarkArrowDead
Arguments: OutArrowSize, OutArrowList
Output:

```
for each arrow i in the OutArrowList, output the following:
        astatus[OutArrowList[i]] = dead;
```

R14: Rule Name: ProcessSon
Arguments: SonSize, SonNumberList
Output:

```
for each son i in the SonNumber List, output the following:
        dobox(SonNumberList[i]);
```

R15: Rule Name: FirstSeqPortInit
Arguments: SeqPortSize, SeqPortList(Each port in the list comes in a pair, the from_port and the to_port)
Output: for each sequential port i in the SeqPortList, output the following:

```
if(astatus[] == running)
        aportval[SeqPortList.to[i]] =
                vm_Copy(avalue[SeqPortList.from[i]]);
else
        aportval[SeqPortList.to[i]] = vm_NullVal();
astatus[SeqPortList.to[i]] = running;
```

R16: Rule Name: SeqPortReInit
Arguments: SeqPortSize, SeqPortList
Output: for each sequential port i in the SeqPortList, output the following:

```
if(astatus[SeqPortList.to[i]] != dead) {
        vm_Delete(aportval[SeqPortList.to[i]]);
        aportval[SeqPortList.to[i]] =
                vm_Copy(avalue[SeqPortList.to[i]]);
}
vm_Delete(avalue[SeqPortList.from[i]]);
avalue[SeqPortList.from[i]] =
        vm_Copy(aportval[SeqPortList.from[i]]);
astatus[SeqPortList.from[i]] = running;
```

R17: Rule Name: EndSeqIteration
Arguments: SeqPoarSize, SeqPortList
Output: for each sequential port i in the SeqPortList, output the following:

```
vm_Delete(avalue[SeqPortList.to[i]]);
avalue[SeqPortList.to[i]] =
        vm_Copy(aportval[SeqPortList.to[i]]);
```

R18: Rule Name: FirstParOutPortInit
Arguments: ParOutSize, ParOutList
Output: for each parallel out port i in the ParOutList, output the following:

```
vm_Delete(aportval[ParOutList[i]]);
aportval[ParOutList[i]] = vm_ListVal();
```

R19: Rule Name: SaveParOutValue
Arguments: ParOutSize, ParOutList
Output: for each parallel out port i in the ParOutList, output the following:

```
vm_Add(aportval[ParOutList[i]],
        vm_Copy(avalue[ParOutList[i]]));
```

R20: Rule Name: FirstParInInit
Arguments: ParInSize, ParInList
Output: for each parallel in port i in the ParInList, output the following:

```
if(astatus[ParInList[i]] == dead)
        exc_raise(inconsistent);
vm_Delete(aportval[ParInList[i]]);
aportval[ParInList[i]] =
        vm_Copy(avalueParInList[i]]);
```

R21: Rule Name: ParInReInit
Arguments: PortNumber
Output:
```
vm_Delete(avalue[PortNumber]);
avalue[PortNumber] =
        vm_Copy(vm_GetNext(aportval[PortNumber]));
astatus[PortNumber] = running;
```

R22: Rule Name: EndParIteration
Arguments: ParOutSize, ParOutList
Output: for each parallel out port i in the ParOutList, output the following:
```
vm_Delete(avalue[ParOutList[i]]);
avalue[ParOutList[i]] =
        vm_Copy(aportval[ParOutList[i]]);
```

8. BIBLIOGRAPHY

(1) Steven P. Reiss, "PECAN: Program Development Systems that Support Multiple Views", *Proceedings of Seventh International Conference on Software Engineering*, March 1984, pp 30 - 41.

(2) Miren Begona Albizuri-Romero, "Grase - A Graphical Syntax-Directed Editor for Structured Programming", *Sigplan Notices, V19*, #2, Feb 1984, pp 28 - 36.

(3) M.C. Pong, N. Ng, "PIGS - A System for Programming with Interactive Graphical Support", *Software - Practice and Experience*, Vol 13, 1983, pp 847 - 855.

(4) Michael L. Powell, Mark A. Linton, "Visual Abstraction in an Interactive Programming Environment", *ACM Tenth Annual Symposium on Principles of Programming Language*, 1983, pp 14 - 21.

(5) William Finzer, Laura Gould, "Programming by Rehearsal", *Byte 9(6)*, June 1984, pp 187 - 210.

(6) Ephriam P. Glinert, Steven L. Tanimoto, "PICT: An Interactive Graphical Programming Environment", *IEEE Computer, 17:11*, Nov. 1984, pp 7 - 25.

(7) G. Raeder, Programming in Pictures, PhD dissertation, University of Southern California, Los Angeles, Calif., Nov 1984; Technical Report TR-84-318, USC; Technical Report 8-85, Norwegian Institute of Technology, Trondheim-NTH, Norway.

(8) Gretchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich, Paul Souza, "Program Visualization: Graphical Support for Software Development", *IEEE Computer, 18:8*, August 1985, pp 27 - 35.

(9) Mark Moriconi, Dwight F. Hare, "Visualizing Program Designs Through PegaSys", *IEEE Computer, 18:8*, August 1985, pp 72 - 85.

(10) A.L. David, and R.M. Keller, "Data Flow Program Graphs", *IEEE Computer, 15:2*, February 1982, pp 26 - 41.

(11) T.D. Kimura, *Completion Problem and Its Solution for Context-Free Languages (Algebraic Approach)*, Moore School Report 72-09, University of Pennsylvania, Philadelphia, PA., May 1971.

(12) P. McLain, and T.D. Kimura, *Show and Tell User's Manual*, Technical Report WUCS-86-4, Department of Computer Science, Washington University, St. Louis, March 1986.

(13) P.A. Lee, "Exception Handling in C Programs", *Software - Practice and Experience*, Vol. 13, 1983, pp 389 - 405.

(14) W.D. Gillett and T.D. Kimura, "Parsing Two-Dimensional Languages", manuscript submitted to COMPSAC86, Chicago, October 1986.

(15) A.W. Bojanczyk and T.D. Kimura, *A Systolic Parsing Algorithm for a Visual Programming Language*, Technical Report WUCS-86-7, Department of Computer Science, Washington University, St. Louis, March 1986.