

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering
Research

Computer Science and Engineering

Report Number: WUCS-86-06

1986-06-01

A Visual Language for Keyboardless Programming

Takayuki Dan Kimura, Julie W. Choi, and Jane M. Mack

A visual language Show and Tell is introduced as a programming language for home information systems, integrating the computer capabilities of managing computation, communications, and database. It is shown that keyboardless programming is possible with Show and Tell. The language is implemented on the Apple Macintosh personal computer. The semantic model of the language is based on the concepts of dataflow and completion. A Show and Tell program is a partially ordered set of nested boxes and arrows. Traditional programming constructs such as subroutine, iteration, record structure recursion, exception, concurrency and so forth, are represented by two-dimensional graphical structures... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation

Kimura, Takayuki Dan; Choi, Julie W.; and Mack, Jane M., "A Visual Language for Keyboardless Programming" Report Number: WUCS-86-06 (1986). *All Computer Science and Engineering Research*. https://openscholarship.wustl.edu/cse_research/841

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

A Visual Language for Keyboardless Programming

Takayuki Dan Kimura, Julie W. Choi, and Jane M. Mack

Complete Abstract:

A visual language Show and Tell is introduced as a programming language for home information systems, integrating the computer capabilities of managing computation, communications, and database. It is shown that keyboardless programming is possible with Show and Tell. The language is implemented on the Apple Macintosh personal computer. The semantic model of the language is based on the concepts of dataflow and completion. A Show and Tell program is a partially ordered set of nested boxes and arrows. Traditional programming constructs such as subroutine, iteration, record structure recursion, exception, concurrency and so forth, are represented by two-dimensional graphical structures of boxes and arrows. The design philosophy, conceptual model, syntax, and semantics of major language constructs are described. Various research problems in the visual programming area are discussed.

**A VISUAL LANGUAGE FOR KEYBOARDLESS
PROGRAMMING**

**Takayuki Dan Kimura, Julie W. Choi
and Jane M. Mack**

WUCS-86-6

June 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).

Abstract

A visual language Show and Tell is introduced as a programming language for home information systems, integrating the computer capabilities of managing computation, communication, and database. It is shown that keyboardless programming is possible with Show and Tell. The language is implemented on the Apple Macintosh personal computer. The semantic model of the language is based on the concepts of dataflow and completion. A Show and Tell program is a partially ordered set of nested boxes and arrows. Traditional programming constructs such as subroutine, iteration, record structure, recursion, exception, concurrency and so forth, are represented by two-dimensional graphical structure of boxes and arrows. The design philosophy, conceptual model, syntax, and semantics of major language constructs are described. Various research problems in the visual programming area are discussed.

Acknowledgement

Many people contributed to the Show and Tell project . First of all, the authors appreciate contributions made by other project members, in particular, Darren Cruse, Yukari Esman, Peter McLain and Jennifer Pats. Professor Jerry Cox, Jr. has provided constant support and encouragement for the project in various forms and various times. President Isao Ohkawa of CSK had funded this ambitious project for two years with the endorsement of Mr. Shigemi Shiraishi and Dr. Koji Yada of CSK. We very much appreciate their courage and foresight.

The project has received many suggestions, criticisms, and encouragements during its life time. All contributed to the enhancement of the project's results. We are grateful to Professors G.C. Roman, W.D. Gillett, and A. Bojanczyk of Washington University CS Department; Drs. Alan Kay, Andy Hertzfeld, and Dan Ingalls of Apple, Inc.; Professor M. Minsky of MIT; Professor M. Rabin of Harvard; Professor M. Rem of Eindhoven; and Professor Gyula Mago of University of North Carolina.

1. Introduction

This report presents the design philosophy and the basic features of an icon-driven visual programming language called Show and Tell^{TM+}. By a *visual language*, we mean a programming language in which two-dimensional graphic patterns are used to represent various programming concepts, such as subroutines, iteration, recursion, concurrency, exception, data values and data types. For more general usage of the term *visual programming*, see Reference [1].

The Show and Tell Language (STL) is designed for novice computer users who are not familiar with keyboarding. In STL, program construction requires no keyboarding, except for textual data entry. A pointing device, a mouse, is the primary mechanism for user interface. Currently the language is implemented on the Apple[®] Macintosh[™] personal computer. *Programming in MacPaint[™]* or *keyboardless programming* is the main goal of the Macintosh version of STL.

The Show and Tell language system was developed as the first phase of the research activity that started two years ago. The research problem was to investigate the possibilities and limitations of computers at home in future information oriented society. The research goal is divided into three phases; the long term, the short term, and the immediate goal. The outline of research problems at each phase are described below.

1.1 Long Term Goal

The long term goal is to construct a computer system which accomplishes the integration in the following three areas:

- (1) Integration of Environments: home, work, and school.

According to Alvin Toffler [2], powerful micro-mainframe capabilities at home combined with teletext/videotex services will transform a home into an *electronic cottage*, where adults work and children learn in the environment of family life. Information oriented societies are home-centered societies. Office automation and home automation will coincide in such a society.

- (2) Integration of Applications: communication, computation, and database.

In an information oriented society, computation will be only a small part of computer application at home. Communication will be more dominant. A computer will be used to manage communication activities regarding what information to be shared with whom, when, and how. Information sharing can be done either by an exchange of messages, e.g., sending a letter to a school teacher, or through a database, e.g., announcing a garage sale in the community bulletin board. A database query of a CD-ROM based encyclopedia can be considered as another communication activity (sharing information) that requires both database and computation management capabilities. Home computer users would see one application of computer, i.e., information management, instead of three different applications of a computer. The computer capabilities for managing computation, communication, and database must be integrated into a single conceptual framework.

⁺ Show and Tell is a trademark of Computer Services Corporation.
Macintosh is a trademark of Apple Computer, Inc.
MacPaint is a trademark of Apple Computer, Inc.

(3) Integration of Communication Media: image, audio, and text.

Information can be represented in different forms on different media. Communication management needs the capabilities of dealing with different communication media in a unified manner. Hardware technologies for video, audio and textual information have been integrated under the digital technology. However, a similar integration has not been attempted in the current software technology. For example data types for non-textual objects such as video images and audio messages are not available in general purpose programming languages.

There are two fundamental research problems associated with the above long term goal; user interface design and a conceptual model of computer capabilities at home. Home video and audio equipment have a different user interface design than that of a home computer. They provide more direct feedback to the user's operations. User interfaces for different system components with different communication media, must be integrated into a uniform paradigm of user interface for a home information system.

An equally important problem is to construct a unified semantic model of computer capabilities covering the three different application areas of computation, communication and data query. A traditional von Neumann type machine model is not appropriate for modeling communication activities because it can not represent concurrent activities naturally. It is not appropriate for modeling database applications, because it cannot represent pattern matching and pattern search activities without simulating them. Present computer users are forced to learn, implicitly or explicitly, different models of 'computer' in different application areas in order to use application-specific computer languages.

1.2 Short Term Goal

There are different approaches for achieving the long term goal described in §1.1. One approach is to start with design of a hardware equipment for a future home information system. Another is to design an operating system environment in which many software tools will be made available for different home applications. The third approach, which we have taken, centers around design of a new computer language.

Our working hypothesis is that *programming*, a process of assembling or modifying simple computational resources into a complex one, will be required by every user of future information system, in spite of many would-be available application software packages. A general purpose application program or database must be refined to fit the specific needs of a particular user. In this broader definition of programming for end users, algorithm design is not an essential part of programming, contrary to the traditional definition.

The short term goal is to construct a computer language which has the following three aspects:

(1) Visual programming language

High resolution graphics capabilities can facilitate efficient man-machine communication because of its high bandwidth. Direct object manipulation is possible with the high bandwidth, and is essential in any programming environment for home computer users. A visual programming language offers a mechanism for specifying an assembling process in two-dimensional graphic layout, which is more intuitive to the end user than a

textual layout. In addition, a visual program can display the effects of the program execution directly, giving the user the sense of direct object manipulation.

(2) Keyboardless programming language

Programming, as an assembly process of a complex object, involves the selection of components and establishing connections among the components. Programming is a sequence of decision making which consists of choosing objects and the establishment of connections. It requires knowledge about what components are available, how each component can be connected with others, and how to make newly composed objects available for future composition.

In a traditional text-based programming language, identification of components and connections are made through textual names (identifiers, labels and keywords). Even though a construction of a textual name is not usually considered a part of programming activity (some may agree that it is a part of coding), it still requires selections of a primitive object, a character, to construct a complex object, a word. For small children the task of composing a word on a keyboard is a difficult task, primarily because the range of selection is too large to grasp in one span of attention and not all of the choices are displayed explicitly; for example, the effect of Shift key is not visible. Thus, text-based programming requires end users to perform two levels of selection process, one to select a computational resource and the other to select a name component.

In order to simplify user programming task we propose to eliminate the second level of selection process by (i) identifying computational resources with two-dimensional patterns (icons), (ii) displaying all available components in a two-dimensional layout, and (iii) providing a user with a pointing device for selection. We call this *keyboardless programming*.

(3) Job Control Language

There exists no fundamental difference between a programming language and a job control language, or a shell language, both as an assembly language for computational resources. Only difference is that a job control language is used to assemble a large unit of resources such as files and job steps, while a programming language is used to assemble a smaller units such as memory cells and CPU instruction cycles. Users should not be required to learn two different languages with different syntax and semantics for the same purpose of assembling computational resources.

The main research problem for the short term goal is to construct two-dimensional abstract syntax for known programming concepts such as assignment, iteration, recursion, concurrency, exception, synchronization, data type, module, file, quoting, and unquoting. Pictorial representation of computational resources and their relationships are the foundations of visual programming and keyboardless programming.

Designing a two-dimensional syntax requires a different set of design rules than those for linear syntax. For example, due to limited size of screen display, the *economy of space* has to be an important design consideration. Color graphics will enhance the economy. In general a concrete syntax will depend upon the characteristics of the display device.

1.3 Immediate Goal

We had chosen to design a computer language for school children on the Macintosh personal computer system as a case study for visual programming language research. The immediate goal was to design and implement a visual programming language with a concrete syntax suitable for the specific hardware and specific group of end users, in such a way that the research problems of the short term and long term goals could be addressed.

The research problem associated with this phase was integration of visual programming with the Macintosh user interface mechanism, i.e., the desktop metaphors with windows and menus. Our solution to the problem can be characterized by the term *Programming in MacPaint*, one of the most popular application software for Macintosh. Anyone who can use MacPaint should be able to construct a program in the new language.

STL is the result of our efforts to achieve this immediate goal. In this report we will describe the syntax and semantics of STL constructs. The box-arrow syntax of STL is expected to play a key role in development of abstract syntax for visual programming language in the next phase of research. The semantic foundation of STL is a new computation model called Hierarchical Dataflow Model (HDM) which is reported in [3]. We view HDM as a tentative answer to the research problem in the long term phase, of constructing a conceptual model of computation for home computer users.

This report is organized as follows:

1. Introduction
2. System Overview
 - Macintosh Environment
 - System Architecture
 - Basic Capabilities
 - Implementation Note
3. Language Overview
 - Conceptual Model
 - Syntax
 - Semantics
4. Programming in STL
 - Programming Constructs
 - Programming Aspects
5. Conclusions and Future Directions
6. References.

2. System Overview

2.1 Macintosh Environment

STL is currently implemented on a 512K Macintosh with an external disk drive and an Appletalk local area network. A mouse and keyboard are available for user input, and a bit mapped screen and internal speaker are available for output. The operating system is provided by the Finder program and the ToolBox ROM programs. The voice output capability is made available through a commercial product, SmoothTalker^{TM++} from First Byte[®]. For more detailed information about Macintosh system, see Reference [4].

The STL users are assumed to be familiar with the Macintosh environment to the extent that the MacPaint application program can be used reasonably well with window operations and menu selections. We assume the same for the reader.

2.2 System Architecture

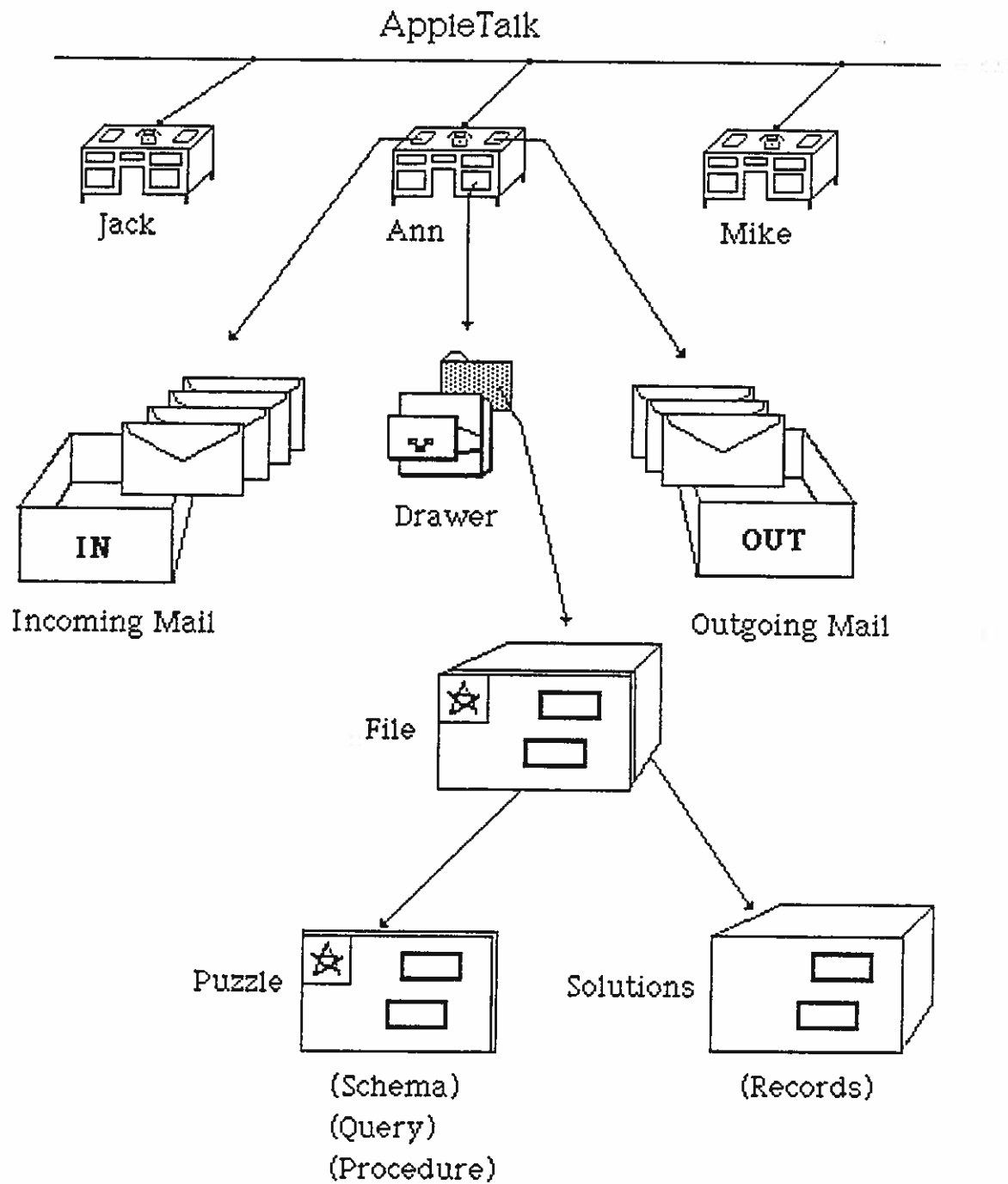
A Macintosh computer, connected to an AppleTalk network, running a copy of the STL system disk, represents a *desk* for STL users. (See Figure 1.) Each desk has a *name* (Macintosh volume name), an *incoming mail box*, *outgoing mail box*, *picture telephone*, and a set of *drawers*. The desk name is unique in the network. The incoming mail box retains a queue of received mail until the mail is opened by the user. The user can send mail to any other desk by putting the mail in the outgoing mail box, as long as the destination desk's name is known. The STL system tries to deliver outgoing mail each time a new desk appears on the network.

A user at a desk can communicate with another user through a picture telephone. The user may initiate a call to any other desk on the network at any time. When the call is made, the computer running the called desk will beep, indicating the incoming call, and the user has an option of answering or neglecting the call. When the connection is established, an empty *phone window* will be created at the both ends. The window is partitioned into three areas; caller's protected area, the responder's protected area, and the unprotected common area. A user can enter or erase any pencil drawing and text in the common area or in his own protected area. Any action on the unauthorized area will be neglected. All drawings and text will be displayed at the both ends simultaneously. (See Figure 2 for an example of phone window.) What you see on your screen is what your partner sees on his screen. Either user can hang up on the picture phone at any time.

A STL drawer is implemented as a Macintosh file and identified by the file name. It contains a collection of (Show and Tell) *files*, where a file is a combination of a *puzzle* and a sequence of known *solutions*. A Show and Tell file represents a relation, in the sense of relational database, where the puzzle specifies the schema and each solution in the file is a record representing a tuple in the relation. The puzzle is a specification of database schema, data query, and computational procedure (operation), altogether in a single form. A drawer is a collection of database and related operations in a particular application area. The concept of drawer was used to introduce a packaging capability into STL for modular programming.

⁺⁺ SmoothTalker is a trademark of First Byte.

Figure 1: Show and Tell Architecture



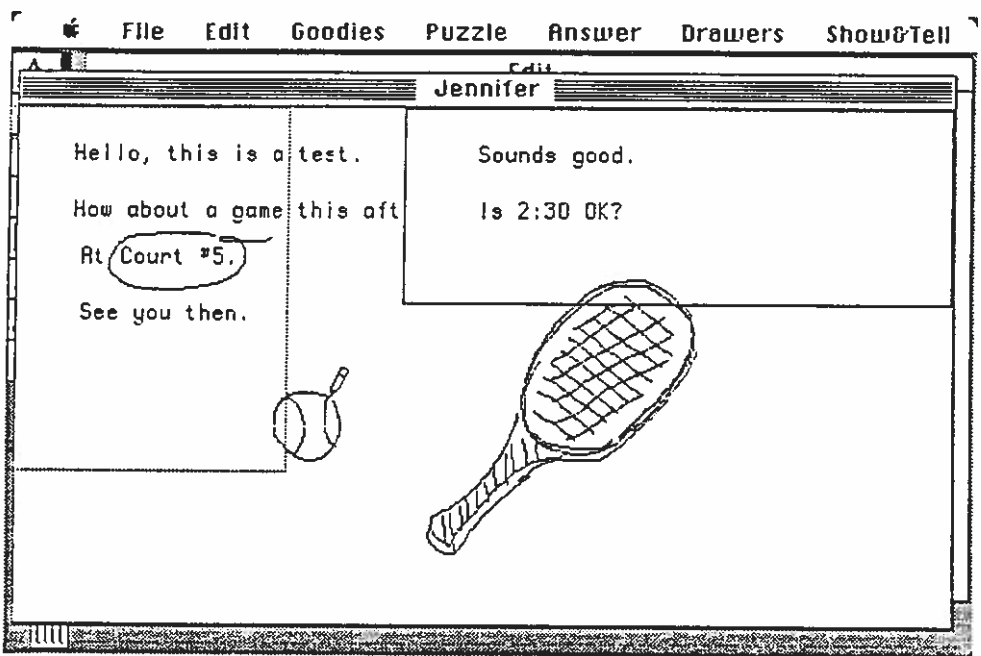


Figure 2: Phone Window

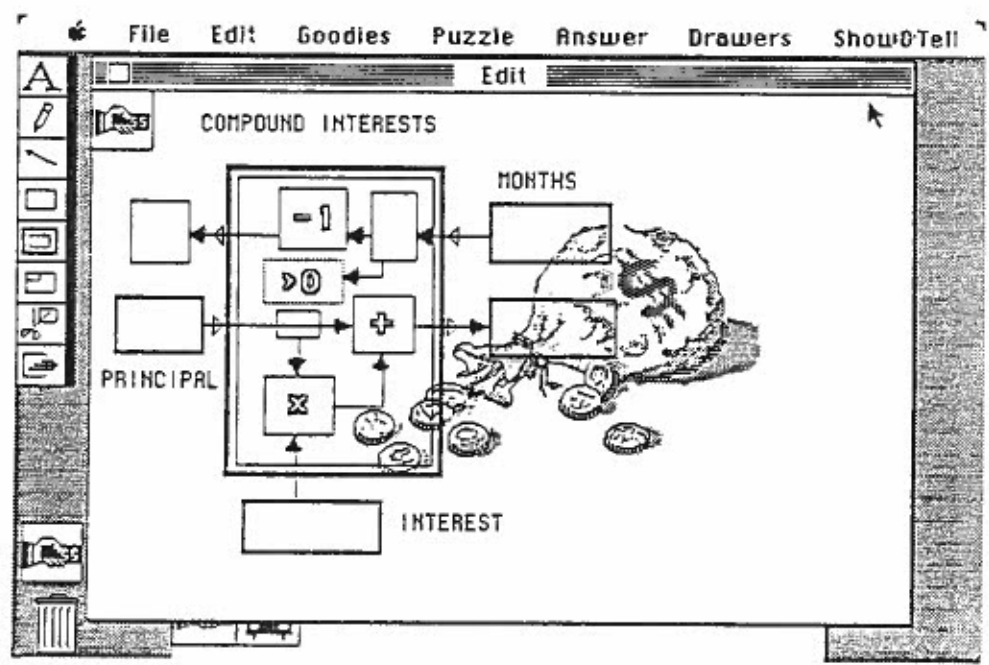


Figure 3: Edit Window

A puzzle is identified by a standard-sized icon which is automatically constructed from the image entered by the user into the *name area* of the puzzle. A puzzle name is also used to identify the file for which the puzzle is the schema definition. Each drawer contains a *directory* which is a sequence of icons representing the all files (puzzles) contained in the drawer. When a drawer is *opened*, the directory of the drawer is displayed on the screen (*directory window*) showing its content. The general syntax and semantics of STL puzzles will be given in the sections that follow.

A user constructs a new puzzle on *Edit Window* using the editing tools similar to those available in MacPaint. The vertical menus provide a set of commands to control saving and solving the newly constructed puzzle. (See Figure 3.) A puzzle consists of boxes connected by arrows. A box may be empty, contain a data value, or may contain another puzzle which is represented either by its icon name or by a nested set of boxes and arrows. Names of component puzzles can be selected from the drawers currently opened. Programming in STL is assembling existing puzzles into a new complex one.

The STL system provides three predefined system drawers; Input/Output, Arithmetic, and Miscellaneous. The directories of the system drawers are given in Figure 4. The standard arithmetic operations are available from the Arithmetic drawer. The semantics of each system-defined puzzle is given in the STL User's Manual ([5]).

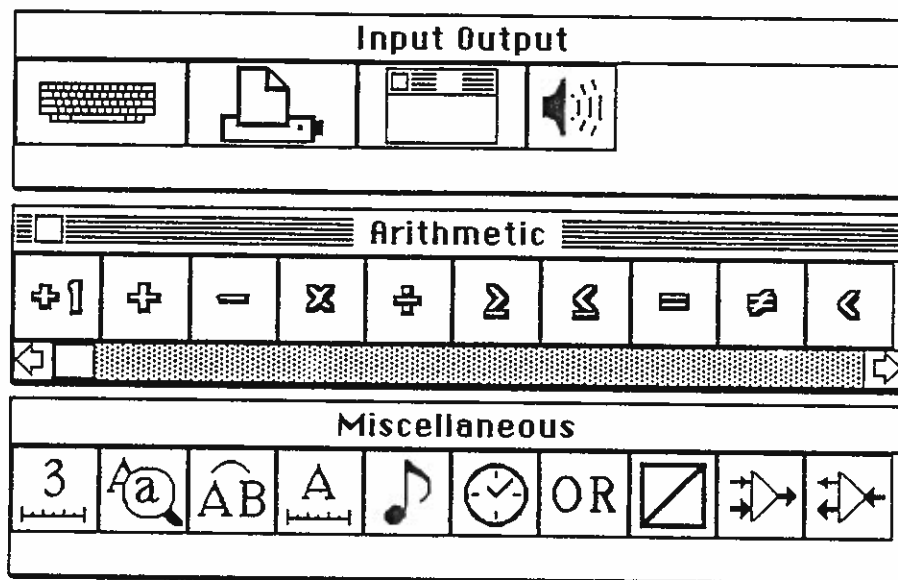


Figure 4: System Defined Drawer Directories

A user can start the system to solve the puzzle in the current Edit window by clicking the "Solve" item in the Puzzle menu and the solution will be displayed in the same Edit window. The system can solve more than one puzzle concurrently, i.e., multitasking is possible in MSTL. The user can start another puzzle by making another Edit window current and by clicking the "Solve" item again, while the first puzzle is being solved.

2.3 Basic Capabilities

We will list the basic capabilities of STL here as a summary of the system overview. Management of computation and data is the main application area of STL. Local area network communication capabilities are incorporated into the system but not integrated into the STL language at the present time, i.e., the user can not control communication activities through a STL program (puzzle). As an algorithmic programming language, STL can be characterized as a functional language based on the dataflow concept.

In the following sections, we will discuss the STL capabilities listed under (2) Database and (3) Computation.

(1) Communication⁺⁺⁺ : AppleTalk local area network

Mailing:	Drawer (file) transfer.
Picture Phone:	Text and pencil drawing.

(2) Database: Relational approach.

Data Objects:	Image, text, and number.
Schema:	Record structure with constraints.
Query:	Join, projection, selection operations.

(3) Computation: Functional, data-driven, and asynchronous execution.

Data Structure:	File (sequence of records). Image, text, number (no logical values).
Control:	Block Structure. Recursion. Sequential iteration (no loops). Parallel iteration. Multitasking.

2.4 Implementation Note

The current version of MSTL is implemented in C using the AZTEC C compiler from Manx Software Systems. The compiler was augmented by us with a preprocessor to simulate the Ada package construct including the exception handling capability. Our method for modular programming in C is reported in [6].

The system consists of approximately 30 packages. The size of source codes is about 700 KB in total. The size of object codes is about 240 KB excluding resource files. The size of resource file is about 130 KB which includes the SmoothTalker speech driver.

⁺⁺⁺ The communication modules are not integrated into the current version of the MSTL system. They are available as stand alone systems.

3. Language Overview

In this section we will introduce the conceptual model, the syntax, and the semantics of STL. Our presentation here is informal. The current implementation of STL on Macintosh, MSTL, is not a full version of STL.

3.1 Conceptual Model

By the conceptual model of STL we mean a definition of the STL system from the user's point of view. There are two fundamental concepts incorporated in STL, *dataflow* and *completion*. Motivations for the choice are given in [3]. We assume that the reader is familiar with the notion of dataflow. (See [7] for reference.) The notion of completion is well-known in psychology ([8]) but little known in computer science ([9]). We will illustrate the notion shortly.

To an end user the STL system is a tool for defining and solving a certain kind of puzzle, called a Show and Tell (ST) puzzle. A ST puzzle consists of boxes connected by arrows. The arrows define the neighborhood relationship among the boxes. Some boxes may be empty. The puzzle is solved when the empty boxes are filled with data objects satisfying the constraints imposed by the neighboring boxes. A ST puzzle defines a *completion problem* of filling the missing portions of a graphic pattern in the same way a jig-saw puzzle defines a completion problem. A puzzle is solved when the puzzle is completed.

The STL system solves a puzzle either by computation or by database search of existing solutions. A puzzle is solved by computation when empty boxes are filled by transfer of data objects from the neighboring boxes. A computation model for STL is similar to the dataflow model of computation. A database search in STL consists of selecting a solution in the database that matches with the partial information given in the remaining part of the puzzle. It is similar to the associative memory model of database. It is possible to combine computation and database search in solving a single ST puzzle.

The notion of completion is used in STL as a mechanism for integrating computation and database into a single conceptual framework. The STL system is a system with completion capability. It can solve puzzles and provide tools for saving, modifying, and sharing puzzles and solutions with other users. The Show and Tell language is a specification language for the Show and Tell puzzles.

Figure 5 illustrates the completion problem by examples. In Figure 5, (a) is a completion problem in perception, (b) defines a jig-saw puzzle as a completion problem in which the constraint is physical shape, (c) is a ST puzzle defining a computation as a completion problem in which the constraint is logical rather than physical, and (d) is a ST puzzle defining a data query as a completion problem in which the constraint is the content of a database.

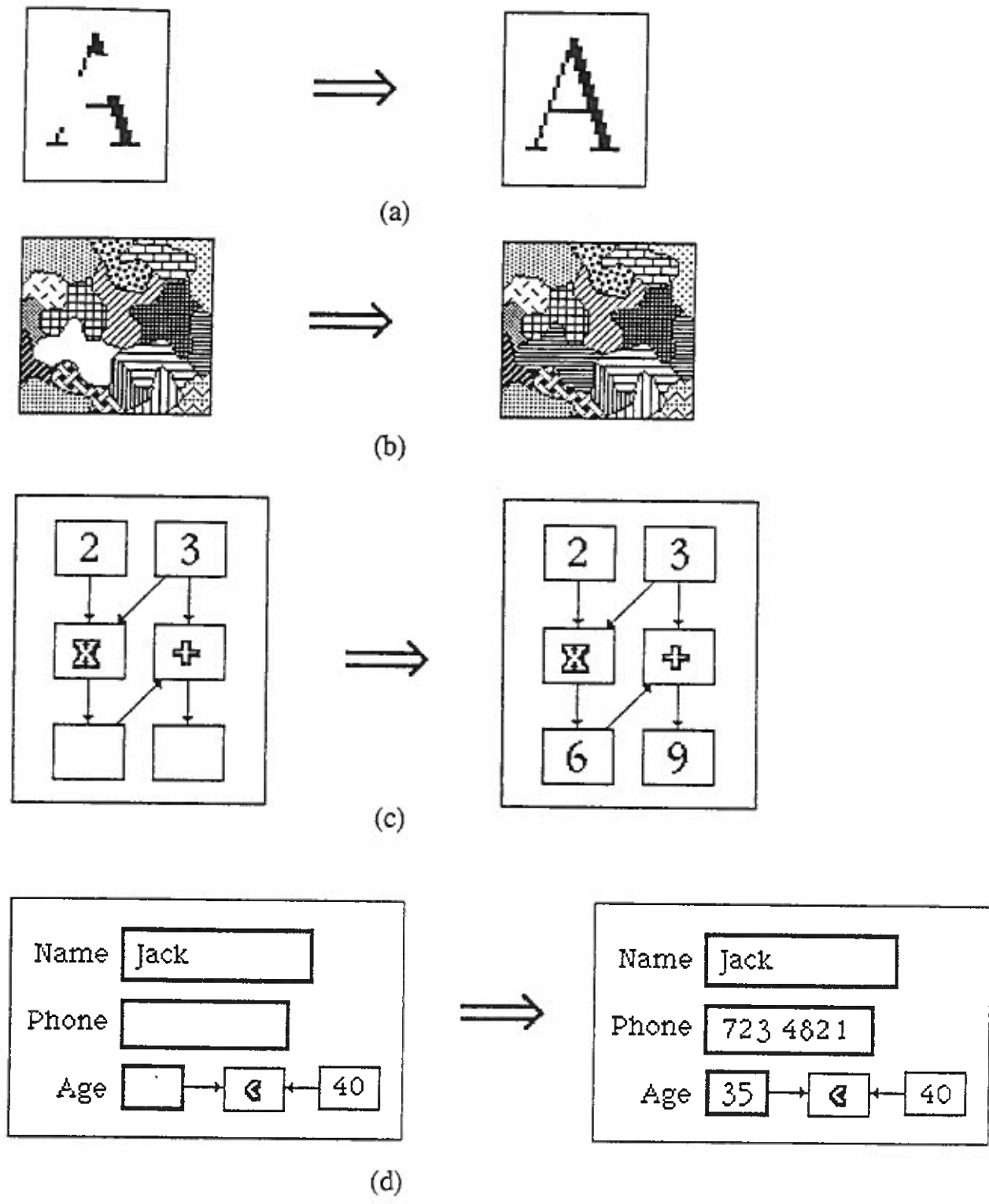


Figure 5: Completion Problems

3.2 Syntax

There is no known formal grammar for STL puzzles. Some of our efforts to construct formal syntax for two-dimensional languages are described in [10] and [11]. We will use a pseudo BNF grammar assisted by English to specify the syntax of STL. In this informal two-dimensional grammar only the shape and type of a figure are significant and the size and geographic location are not. We introduce two composition operations, *concatenation* and *superposition*.

The concatenation of two figures is defined as a juxtaposition of the figures without overlapping. The superposition is a juxtaposition with overlaps sharing some elements of the figures. We denote the concatenation operation by '+' and the superposition operation by '•'. The results of these operations are not necessarily unique, because there may exist many different ways of juxtaposing two-dimensional figures. For example, consider the two figures A and B in Figure 6 (a). Figure 6 (b) enumerates some members of $A+B$, and (c) enumerates some members of $A \cdot B$. Note that since the juxtaposition operation is both commutative and associative, so are the concatenation and superposition operations.

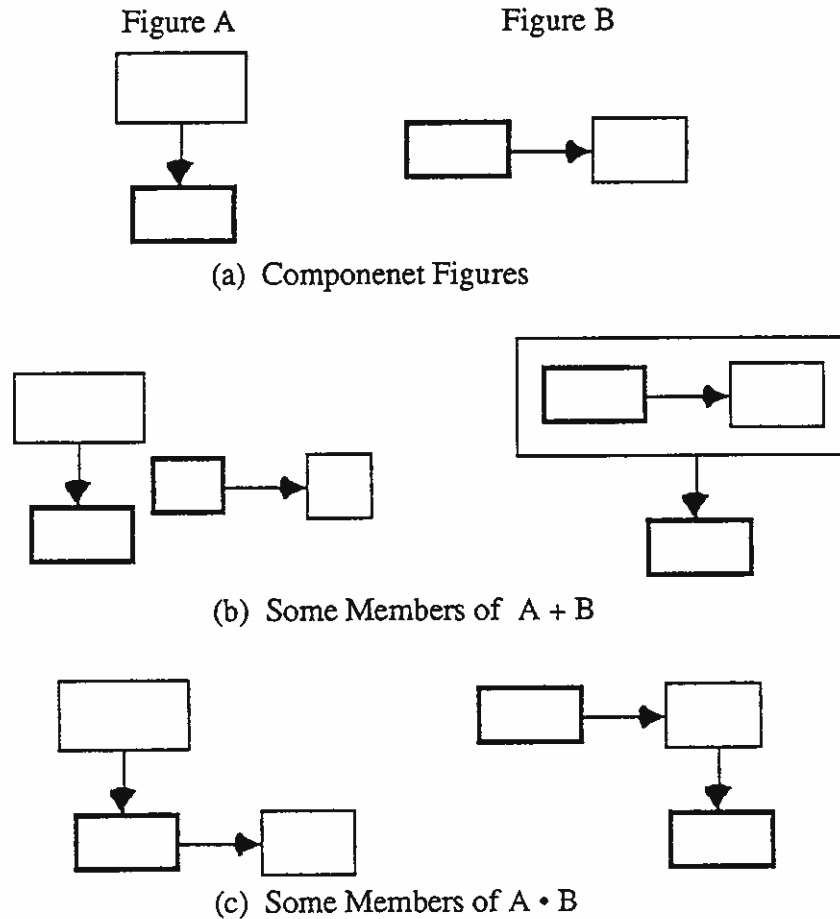


Figure 6: Concatenation and Superposition

3.2.1

puzzle ::= (name + background) • boxgraph
name ::= picture
background ::= picture
picture ::= bit image of any size (MacPaint picture in MSTL)

A puzzle consists of three components; name, background, and boxgraph. A name is a rectangular bit image used to identify the puzzle. A background can be any picture commenting about the puzzle. The name area and the background do not overlap with each other. The boxgraph is the main component of a puzzle that the STL system interprets. It can overlap with the name and/or background. An example of an MSTL puzzle with empty boxgraph is given in Figure 7.

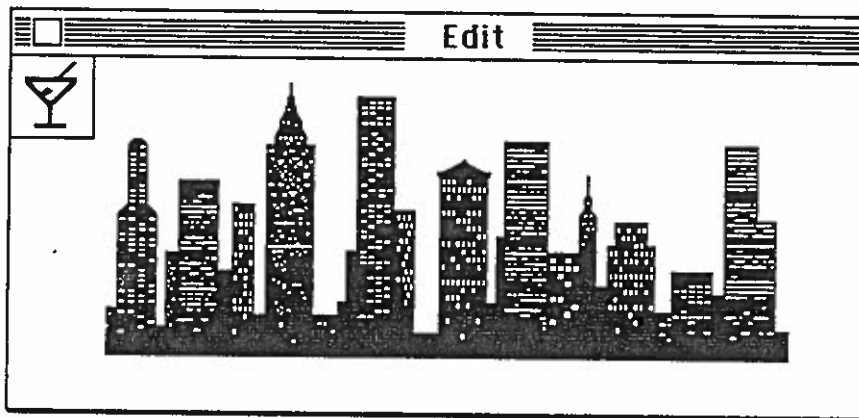


Figure 7: Name Area and Background in MSTL

3.2.2

boxgraph ::= box { box } • [flow]

The notation { } denotes the concatenation closure and [] denotes the superposition closure. The above rule is equivalent to:

boxgraph ::= (box + . . . + box) • (flow • . . . • flow)

A boxgraph consists of one or more boxes connected by a set of arrows. An arrow (flow) connects one box to another defining a flow of data from the originating source box to the destination box. Two boxes may be connected by more than one arrow. An arrow may intersect with other arrows and boxes. While no two boxes may overlap with each other, nesting of boxes is allowed, i.e., one box may contain other boxes. The boxes and arrows may not form any cycle or a loop; a boxgraph is a partially ordered set of nested boxes. It is characterized formally as a directed acyclic multi-graph ([3]).

There are eleven different types of box frames (Figure 8), and there is only one kind of arrow, a solid line with an arrow-head. A line may consist of arbitrarily many line segments each of which is a straight line.

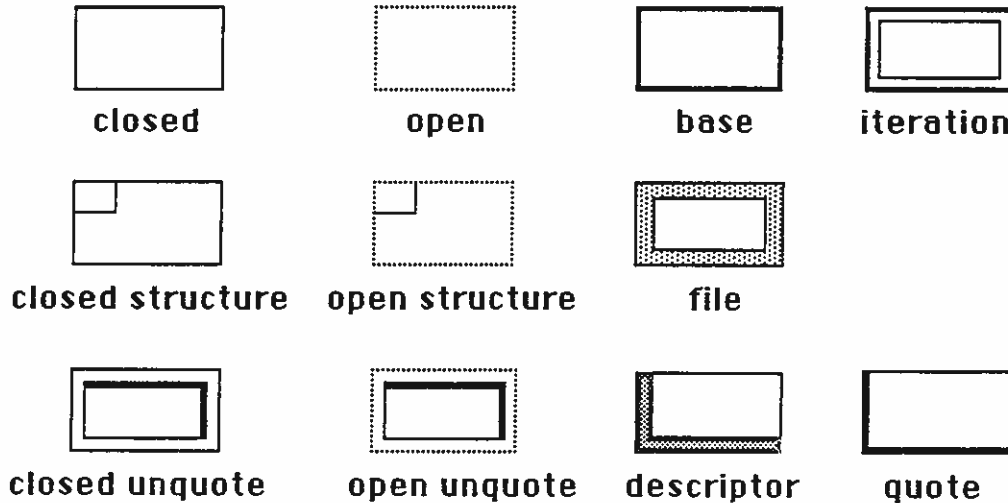


Figure 8: Different Box Frames


3.2.3

**box ::= simple | complex | iteration | file |
structure | unquote**

simple ::= constant | variable | operation

constant ::=  | 

variable ::=  | 


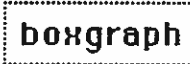
operation ::=  | 

data ::= number | text | picture

icon ::= square bit image of standard size

A simple box may be a constant (a box containing a data object), a variable (an empty box), or an operation (a box containing an icon). A data object can be a number, a text, or a picture. An icon is a fixed sized picture, 32 X 32 in the case of MSTL, constructed from the name area of some existing puzzle. The icon represents the named puzzle.

3.2.4


complex ::=  | 

A complex box is a box containing another boxgraph. An arrow may start from a box inside of a complex box, crossing its frame, and end at a box outside of the complex box. Similarly an arrow may cross the box frame from the outside. However, no arrow may start from or end at the complex box itself. An arrow may start from a box outside of the

complex box, passing through it by crossing its frame twice, and end at another outside box.

3.2.5

iteration ::=  + { port }

port ::= ▷ | 

An iteration box contains a boxgraph. An arbitrary number of ports, either *sequential* or *parallel*, can be attached to the outside frame of an iteration box. A sequential port is represented by a pair of triangles positioned at the opposite sides of the frame. A parallel port is represented by a meshed rectangle on the outside frame. One and only one arrow may cross a sequential port while more than one arrow may cross a parallel port as long as they are in the same direction. No arrow may pass through an iteration box. In MSTL, no arrow may cross more than one port. As in the case for a complex box, an arrow may cross the frame of an iteration box in either direction. Figure 9 illustrates possible ways of intersecting an iteration box with arrows.

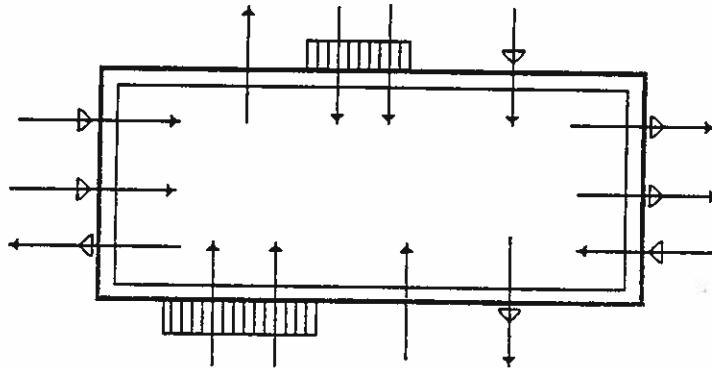

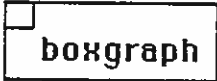
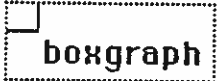
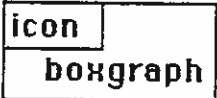



Figure 9: Iteration Box and Ports

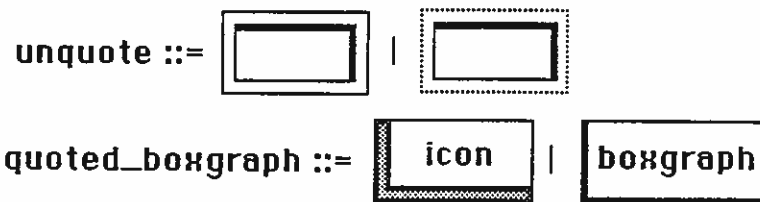
3.2.6

file ::= 

structure ::=  |  |
 | 

A file box must contain an icon which represents the solution sequence of the named puzzle. A file box may have incoming arrows only or outgoing arrows only. A structure box defines the record structure (schema) of a file. It may contain an icon in its name area to share the record structure of the named puzzle. Structure boxes are similar to complex boxes except that an arrow may start from or may end at a structure box itself.

3.2.7



An arrow may start from or end at the outside frame of the unquote box. No arrow may pass through the frame of an unquote box. An unquote box may have incoming arrows ending at the inside frame.

3.2.8

flow ::= direct_flow | indirect_flow
direct_flow ::= constant \longrightarrow **constant** |
structure \longrightarrow **structure** |
quoted_boxgraph \longrightarrow **unquote** |
file \longrightarrow **structure** | **structure** \longrightarrow **file**

A flow is an arrow connecting two boxes. Every flow has a unique source and destination box. There may be more than one flow between two boxes. An arrow consists of a chain of line-segments with the last segment having the arrow head. An arrow may intersect with the frames of variable (empty), complex, iteration, or structure boxes. When an arrow intersects with the frame of an iteration box, it may or may not pass through a sequential port or a parallel port. An arrow may pass through a box, intersecting twice with the box frame, only if it is an empty box or a complex box.

A direct flow connects directly two boxes having or expecting a particular type of value. A flow between a file box and structure box must pass through exactly one parallel port. The arrow head of a flow from a quoted boxgraph to an unquote box must reach the inside frame of the unquote box.

3.2.9

indirect_flow ::= station \longrightarrow **station** |
(constant | structure | file | quoted_boxgraph) \longrightarrow **station** |
station \longrightarrow **(constant | structure | file | unquote)**

station ::= variable | operation | unquote

An indirect flow contributes to a dataflow through a general flow control box such as an empty box and operation box. We call them *station* boxes. A station box can be used to control dataflow of any value type. An arrow connecting a station to an unquote must reach the inside frame of the destination. Figure 10 gives several examples of direct and indirect flow.

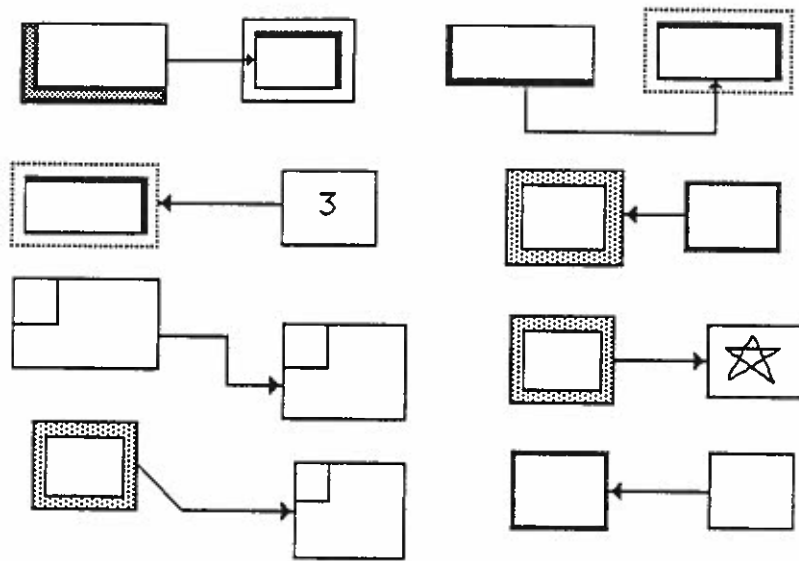


Figure 10: Examples of Flow

3.3 Semantics

We will define the semantics of STL by describing how a solution for a STL puzzle can be constructed. Even though there exists a declarative definition of a STL solution, we will use imperative descriptions in this informal exposition for easier understanding. A formal declarative definition is given in [3].

A solution is an assignment of data values to the empty (variable) boxes of a puzzle such that the resulting puzzle may be *consistent*, i.e., free from contradiction. The notion of consistency is the most fundamental semantic concept for STL. Since the STL consistency is based on the identity of data values, we will first discuss the data types of STL.

3.3.1 Data Type

We define a data type as a set of functions and predicates defined on the value set. The following data types are available in STL:

<i>Number</i>	a floating point number.
<i>Text</i>	an arbitrarily long sequence of alpha-numeric characters.
<i>Picture</i>	a rectangular bit-image of arbitrary size.
<i>Boxgraph</i>	a data structure representing a grammatical boxgraph.
<i>List</i>	a list of data values of any type including the list type.

Note that there is no Boolean data type in STL. The set of operations for each data type is provided in the system defined drawers. In the current implementation of MSTL, however, the following limitations exist:

- (1) The boxgraph data type is not supported.
- (2) No operation on the picture data type is available, nor the identification operation; the system cannot decide whether two pictures are identical or not.
- (3) Standard arithmetic operations are available. (See Figure 3.)
- (4) Three operations on the text data type are available:



search



concatenation



length

The system can identify two text values.

- (5) Two operations on the list data type are available:



construct



decompose

The system cannot identify two lists. A list object cannot be made visible on the screen.

3.3.2 Dataflow

The STL system solves a puzzle by transferring a data object from one box to another following the direction of an arrow. The data transfer is asynchronous and the duration is unknown. If the arrow passes through an empty box on the way to the destination box, the system fills the empty box with the data object, changing the variable box into a constant box. If the arrow passes through a complex box which contains an inconsistent boxgraph, the data transfer will be terminated at the complex box. If the complex box is consistent, the data transfer continues toward the destination box. Thus, a dataflow can be switched on or off by the consistency status of a complex box intersecting with the flow.

When a data value is transferred to a variable (empty) box, the value will be registered inside the box and the box becomes a constant box. If the destination is a constant box containing the same value as the one transferred, no change occurs and the destination box stays the same constant box. If the destination box contains a different value from the transferred data object, the smallest boxgraph that contains the arrow becomes *inconsistent*. If the destination is an operation box, the transferred value will be consumed by the operation box as one of its arguments. Such an operation box will produce new values on the outgoing arrows.

Using the system defined arithmetic operations, we will illustrate the notion of dataflow in Figure 11(a).

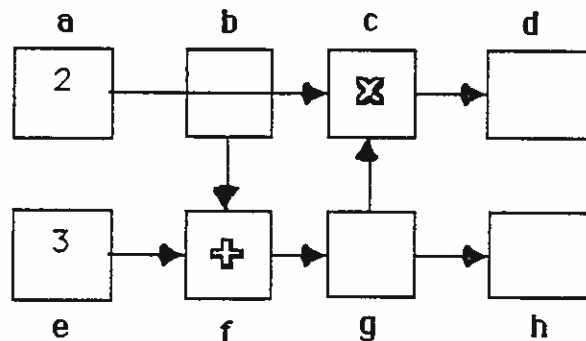


Figure 11 (a): Dataflow

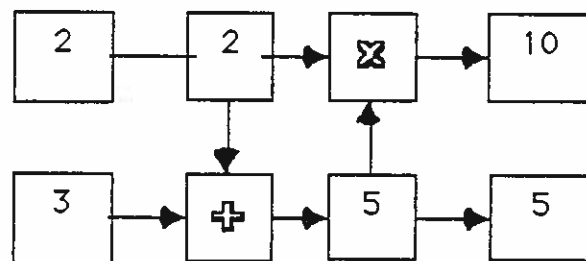


Figure 11 (b): Solution for (a)

The number value 2 in the constant box a is transferred to the multiplication operation box c through a variable box b, which will be filled by the value 2. Meanwhile, the number value 3 in e is transferred to the addition operation box f and waits there for consumption. As soon as the necessary two arguments become available to the addition operation in f, the operation will be executed, consuming two values and producing one new value 5, which is to be transferred to the variable box g. Subsequently, the value 5 is transferred from the box f to the box c and the box h. As soon as the multiplication

operation in the box c gets two arguments, it will be executed, consuming the two values and producing the one value 10 which will be transferred to the variable box d. Thus, Figure 11(b) is the solution for the puzzle given in Figure 11(a).

Note that the data transfer from box a to box c and the transfer from box e to box f is concurrent, and that the operations will be executed when and only when the argument values become available. The multiplication must be performed after the addition in this example, because the former requires the data from the latter. Besides this data dependency, there is no sequencing control assumed in STL. Parallel computation is more basic than sequential computation in STL.

Since there is no cycle in a STL puzzle, once a data object occupies a variable box, changing it to a constant box, the assignment of the value to the box never changes. Therefore, computation in STL is functional and has no side effects.

3.3.3 Consistency

As stated in Section §3.3.2, when a data value is transferred to another box containing a different value, the smallest boxgraph containing the destination box will become *inconsistent*. This is the most primitive form of inconsistency in STL, and the boxgraph containing such conflicting dataflow will be called *primitively inconsistent*. Figure 12 gives examples of primitively inconsistent puzzles. Figure 12(b) is inconsistent because the same conflict as in (a) will arise when 2 or 3 is transferred to the empty box. Figure 12(c) is inconsistent because the number value 5 produced by the addition operation conflicts with the constant 6.

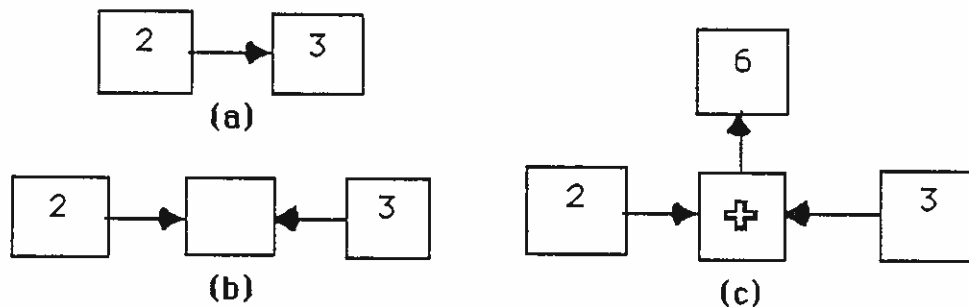


Figure 12: Primitively Inconsistent Boxgraphs

A boxgraph is inconsistent if and only if it is primitively inconsistent or it has an open complex box containing an inconsistent boxgraph or an open operation box containing an icon representing a puzzle whose boxgraph becomes inconsistent when it is executed. A boxgraph is *consistent* if it is not inconsistent. If an inconsistent boxgraph is contained in a closed box, the smallest boxgraph containing the closed box is not necessarily inconsistent. A closed box confines the inconsistency inside the box while an open box allows a propagation of inconsistency out of the box toward the global environment. The inconsistency is broadcasted toward the other components of the boxgraph. Inconsistency is similar to exception in traditional programming languages, and a closed box defines the scope of inconsistency as the begin-end block structure defines the scope of exception in Ada, for example.

The effect of inconsistency is by definition to make the boxgraph non-communicative and non-existing when viewed from the outside. Any dataflow passing through a box containing an inconsistent boxgraph will be terminated at the box. This switching capability of a complex box is the main motivation for introducing the concept of consistency into STL. Since every boxgraph is either consistent or inconsistent, any complex box or any operation box, which contains a named boxgraph, can represent a predicate. Consistency plays the role of the Boolean data type.

Figure 13 illustrates the switching capability of boxgraphs.

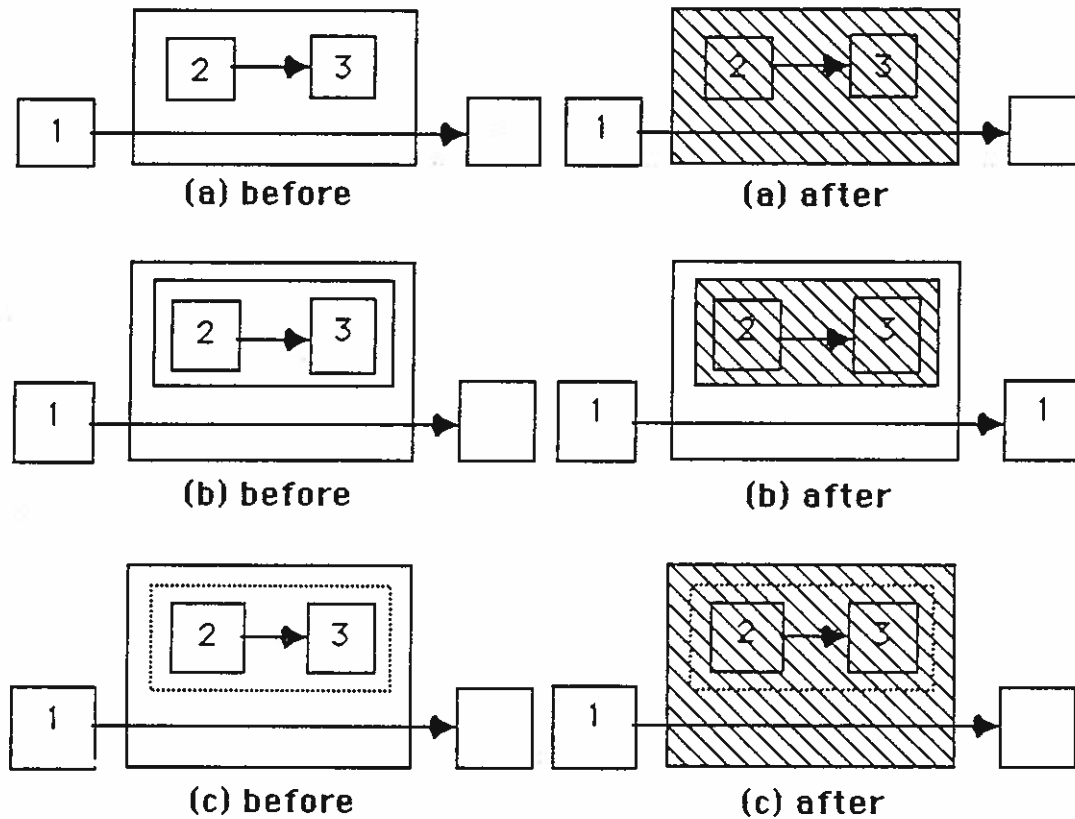


Figure 13: Switching with Consistency

In Figure 13(a) the complex box is inconsistent, therefore the constant data object 1 cannot reach the destination box. Note the MSTL system hatches all inconsistent complex boxes during the execution time. In (b) the inconsistency is contained inside the smaller closed box, and the larger complex box is consistent. Therefore, the constant 1 can reach the destination. In (c) the inconsistency propagates out of the smaller open box, and the larger complex box becomes inconsistent as in (a).

For another example, Figure 14 illustrates how logical operations can be implemented in STL using the switching capability of consistency. Figure 14(a) presents the AND operation computing '0 and 1 = 0' and (b) presents the OR operation for '0 or 1 = 1'.

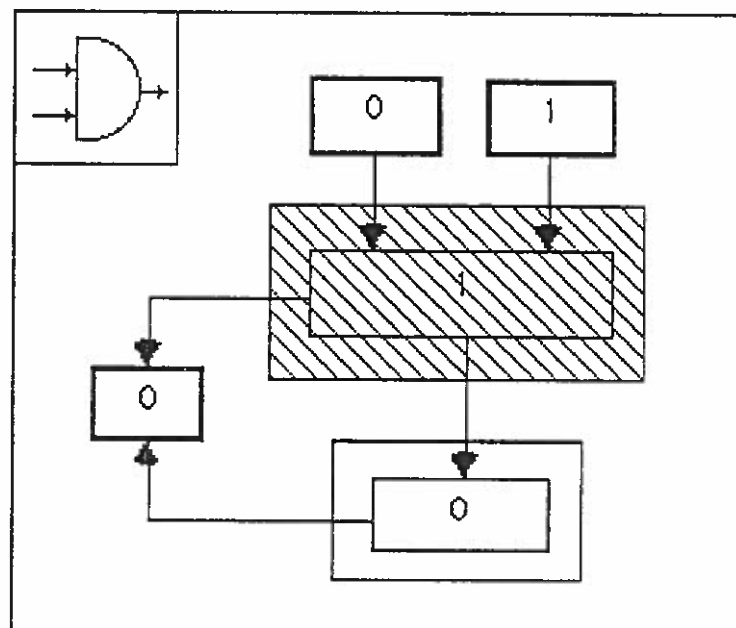
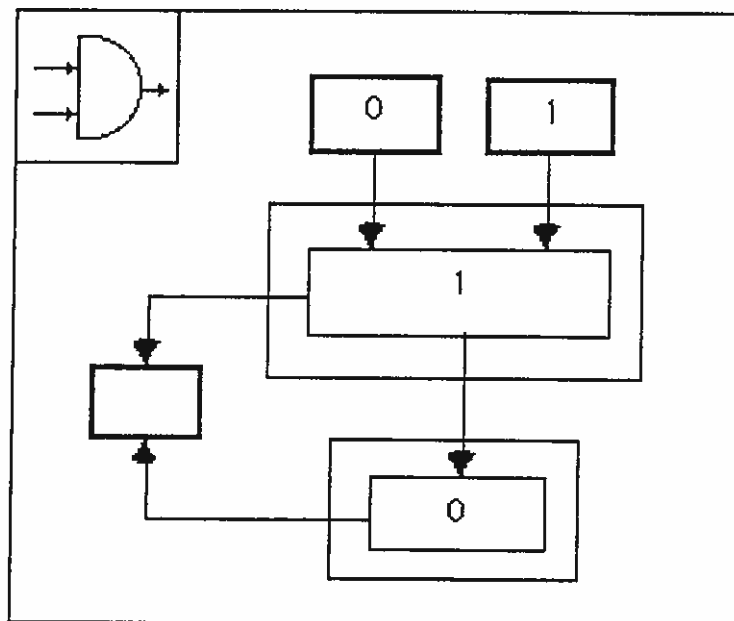


Figure 14 (a): AND Operation

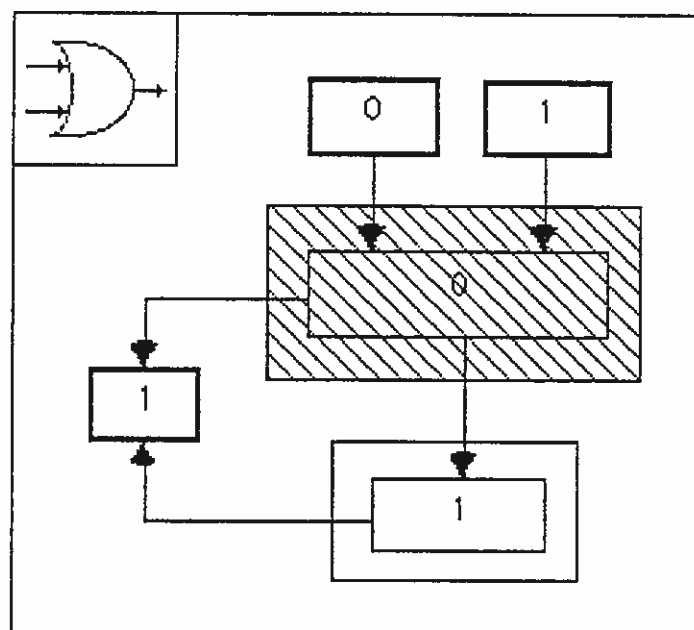
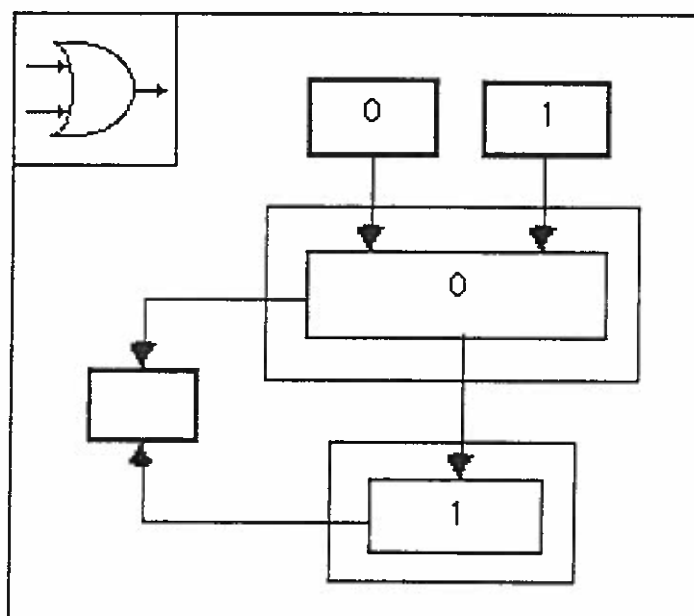


Figure 14 (b): Or Operation

3.3.4 Folding

A boxgraph can be arbitrarily complex. First of all, there is no theoretical limit on the number of boxes in a boxgraph. Secondly, each box itself can be arbitrarily complex because the box may contain another boxgraph entirely, i.e., nesting of boxgraphs is allowed in STL. We differentiate two types of complexity associated with a boxgraph; the number of boxes and the maximum depth of nesting. We call the former the *horizontal* complexity, and the latter the *vertical* complexity. Figure 15 compares two boxgraphs of similar function with different complexity profile.

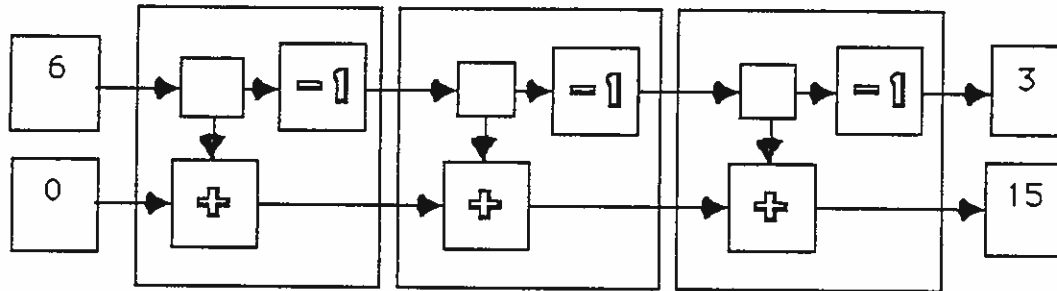


Figure 15 (a): Horizontally Complex Boxgraph

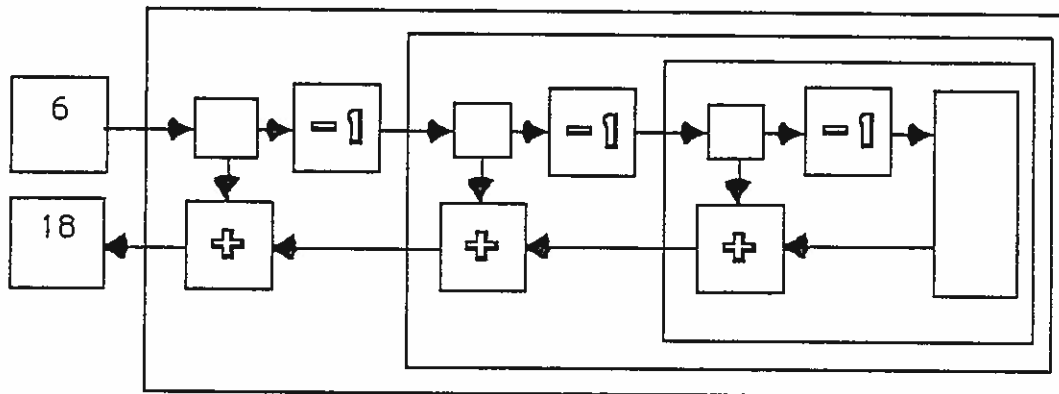


Figure 15 (b): Vertically Complex Boxgraph

For managing such complexities, there are two abstraction mechanisms provided in STL, *folding* and *naming*, each intended to be used for reducing the horizontal complexity and the vertical complexity, respectively. The iterative looping construct in a traditional programming language corresponds to the folding abstraction in STL, and the recursive procedure corresponds to the naming abstraction. In this section we will discuss the folding abstraction and in the next section we will discuss the naming abstraction.

Folding is to collect a spatially (or horizontally) spreading array of similar boxgraphs into one place. It is represented by an iteration box in STL. For example, the boxgraph in Figure 15(a) can be folded into the form shown in Figure 16(a), and the corresponding MSTL syntax is given in Figure 16(b).

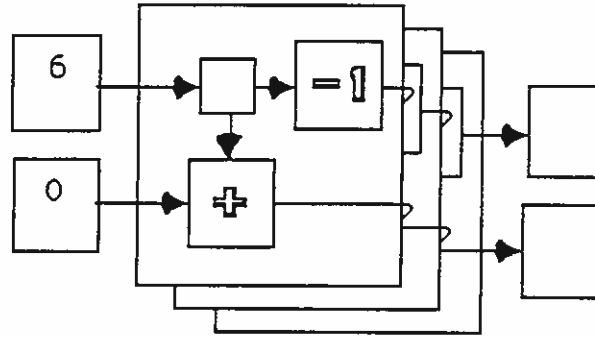


Figure 16 (a): Folding Boxgraphs

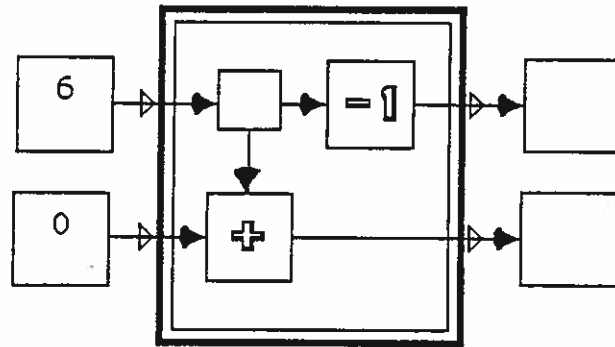


Figure 16 (b): Specification of Folding Using Iteration Box

Note that the iteration box of Figure 16(b) specifies a folding of unbounded number of components, rather than three components as in (a). Methods of limiting the number of components in a folding will be explained later.

There are three different forms of interaction (communication) among the folded components and their environment; *serial*, *parallel*, and *global*.

Sequential iteration: A folding with a sequential iteration provides a serial communication among the components. Sequential iteration is represented by a pair of *sequential ports* (small triangles) attached to the iteration box. Figure 17 shows the general syntax and semantics of sequential iteration, where α is an arbitrary boxgraph. In MSTL when the user enters an incoming sequential port by clicking its location, the system automatically enters the outgoing port at the corresponding location of the opposite side of the iteration box. There may be more than one sequential port on a single iteration box as illustrated by Figure 16(b), but only one arrow may pass through each sequential port.

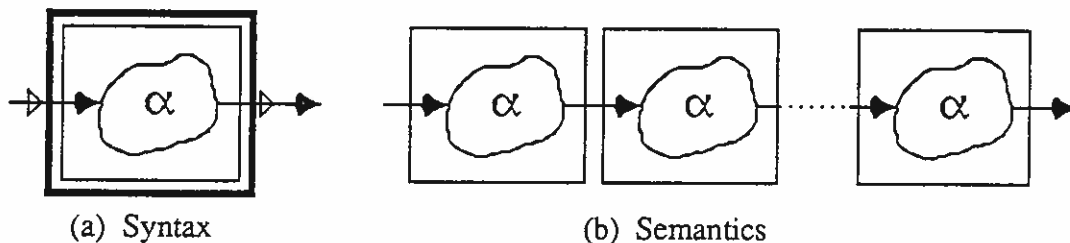


Figure 17: Sequential Iteration

When the STL system executes an iteration box such as the one in Figure 17(a), the system unfolds the array by creating dynamically a new copy of α and transferring the data from the latest component to the new one. The unfolding terminates when the newly created component boxgraph is evaluated as inconsistent. For example the iteration box of Figure 16(b) can be bounded as shown in Figure 18. The boxgraph computes $6+5+4+3+2+1 = 21$.

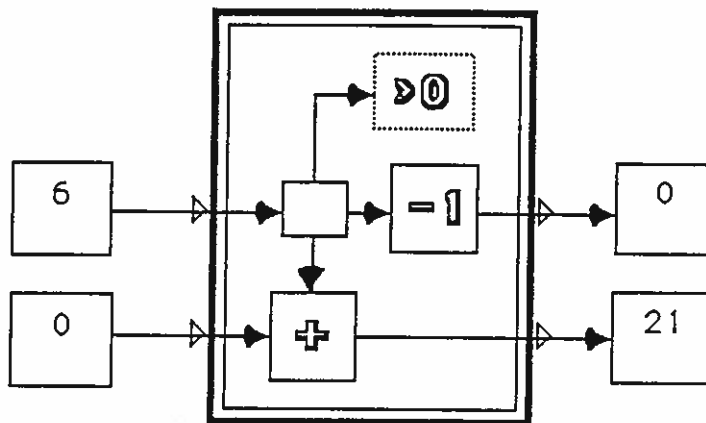


Figure 18: Bounded Sequential Iteration

Note that the data transfer from one component to the next is synchronized at the boundary of the component, limiting the degree of parallelism existing inside the array. However, within each component dataflows are still asynchronous and potentially parallel. Also note that sequential iteration is the only way by which the array components can communicate with each other.

If an iteration box has no communication port, then the unfolding does not terminate. For example Figure 19 is an MSTL puzzle that speaks the current time indefinitely.

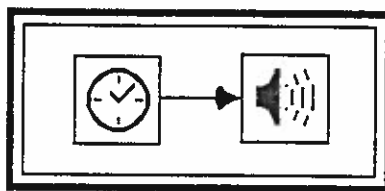


Figure 19: Unbounded Iteration

Global input: An iteration box can receive a data value from an incoming arrow when the arrow does not pass through any communication port. The value will be transferred to every component of the iteration. This corresponds to a global variable in a traditional programming language. The syntax and semantics are given in Figure 20.

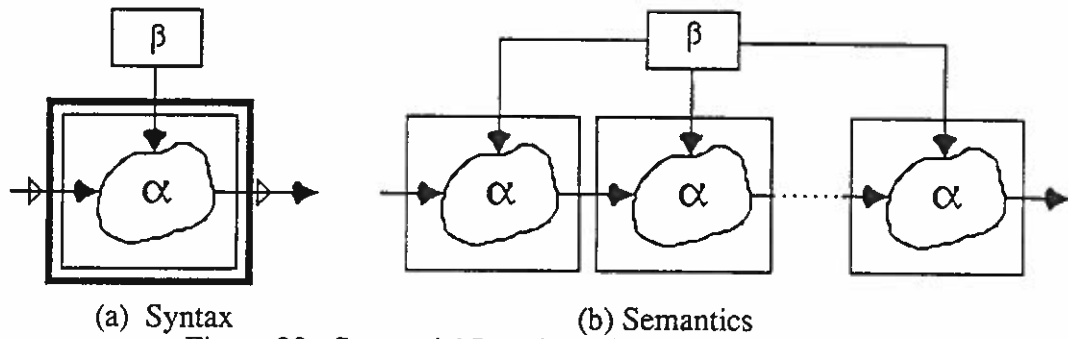


Figure 20: Sequential Iteration with Global Inputs

Parallel iteration: A folding with a parallel iteration provides a parallel communication between two folding arrays. Parallel iteration is represented by a *parallel port* (a striped rectangle) attached to the iteration box. There may be more than one parallel port on a single iteration box, and more than one arrow may pass through each parallel port. Figure 21 shows the general syntax and semantics of simple parallel iteration, where there is only one parallel port on each iteration box, and α and β are arbitrary boxgraphs.

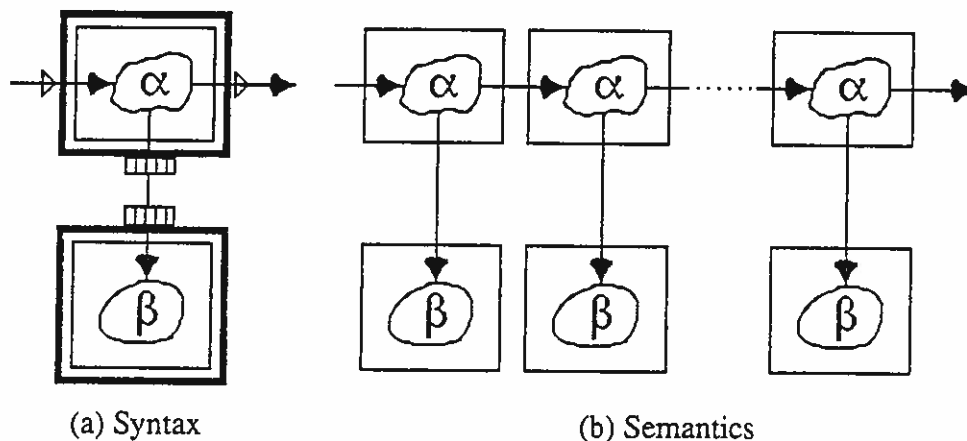


Figure 21: Parallel Iteration

In Figure 21 the α iteration (source) box controls the unfolding of the β iteration (destination) box. The source terminates when an inconsistent α boxgraph is created, and the destination terminates when the source terminates. During the unfolding process, if an inconsistent β boxgraph is created, it will be discarded. This provides a mechanism of incorporating the selection or filtering capability into the β boxgraph. It is illustrated by the example in Figure 22. The left-hand side iteration box in Figure 22 produces a sequence (list) of numbers 1 through 10 and the right-hand side iteration box consumes the sequence by selecting the numbers less than 5 and constructs their product: $1 \times 2 \times 3 \times 4 = 24$.

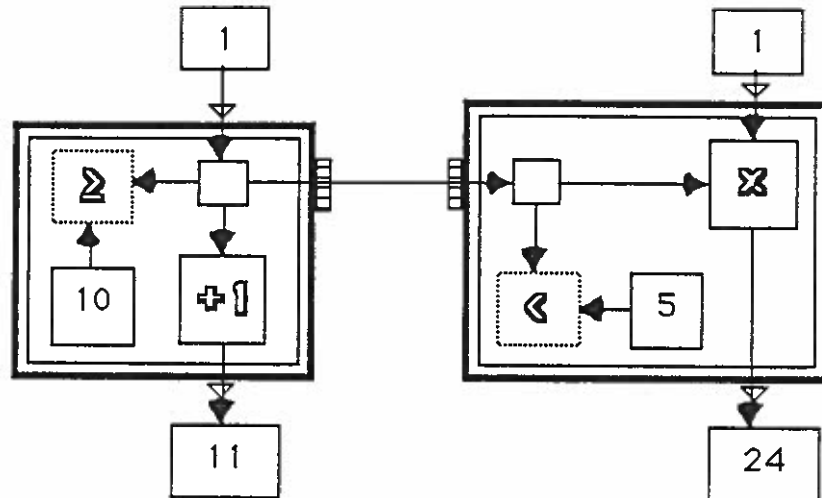
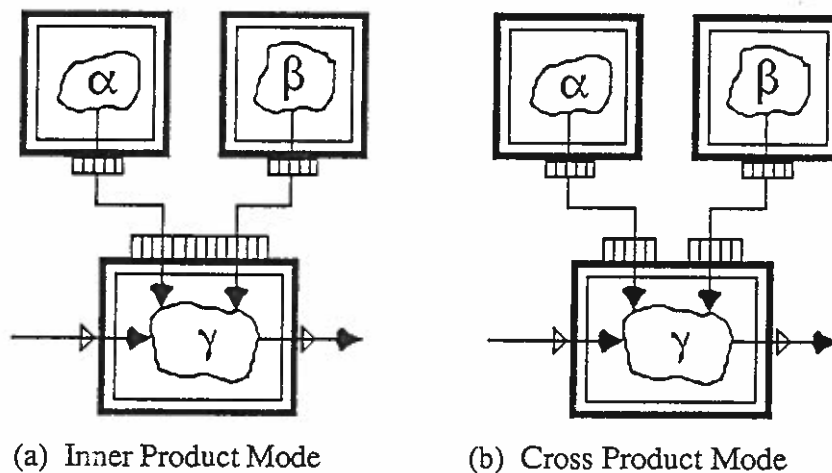


Figure 22: Selection in Parallel Iteration

When two arrows enter a parallel iteration box, they may share the same parallel port or they may enter through separate parallel ports. The former mode of interfacing is called the *inner product* mode and the latter the *cross product* mode. See Figure 23 for the syntactic difference. The syntax for more than two arrows entering a parallel iteration box is similar.



(a) Inner Product Mode

(b) Cross Product Mode

Figure 23: Parallel Iteration Modes

The semantics of inner product mode is illustrated by Figure 24 (a). Unfolding of the destination iteration is synchronized with unfoldings of all the source iterations. If α (or β) iteration has a sequential port, then its components will communicate with each other sequentially in synchronization with the γ iteration. The semantics of cross product can be represented by a nested set of parallel iterations as shown in Figure 24 (b) and (c). The difference between Figure 23 (b) and Figure 24 (b) is similar to the difference between the following two array declarations in Ada:

```
type T23 is array ( $\alpha, \beta$ ) of  $\gamma$ 
```

```
type T24 is array ( $\alpha$ ) of array ( $\beta$ ) of  $\gamma$ .
```

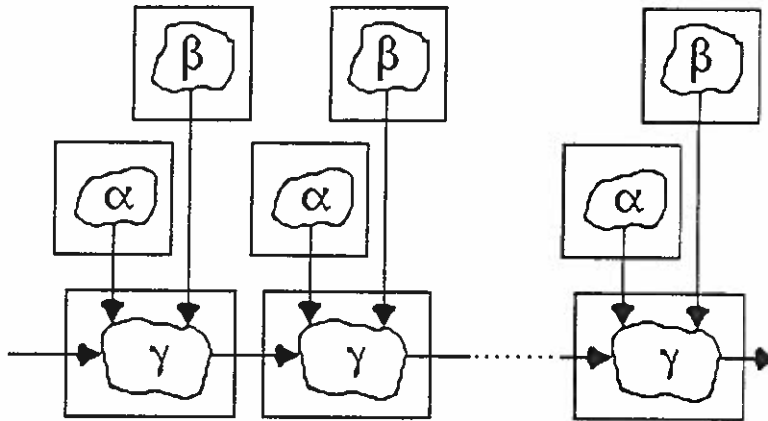


Figure 24 (a): Semantics of Figure 23 (a)

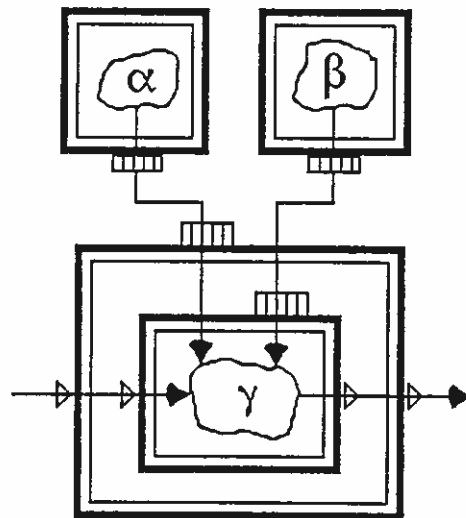


Figure 24 (b): Semantics of Figure 23 (b)

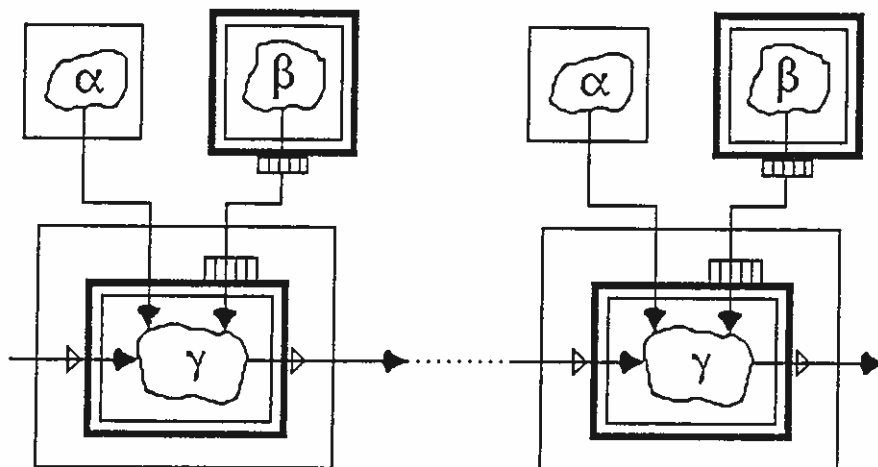


Figure 24 (c): Unfolding of (b)

3.3.5 Naming

A boxgraph is vertically complex when it contains a nested set of boxes as in Figure 15(b). Naming is an abstraction mechanism for managing the vertical complexity. By naming we mean to represent an arbitrarily complex boxgraph by a name (an icon in MSTL) of fixed complexity. It corresponds to the procedure concept in a traditional programming language. Any occurrence of a name icon in a boxgraph can be replaced by the boxgraph associated with the icon without changing the consistency of the original boxgraph.

For example, The boxgraph of Figure 15(b) can be approximately represented by Figure 25(a), where the name is defined by Figure 25(b) in the MSTL syntax. The square at the upper-left corner is the name area that contains an icon. It is to be defined as the name of the boxgraph constructed in the remaining part of the Edit window. The representation is approximate because the boxgraph of Figure 25(a) represents an unbounded nesting of the same boxgraph, while Figure 15(b) has a bounded depth of nesting. An example of naming a bounded nesting will be given later in Figure 27.

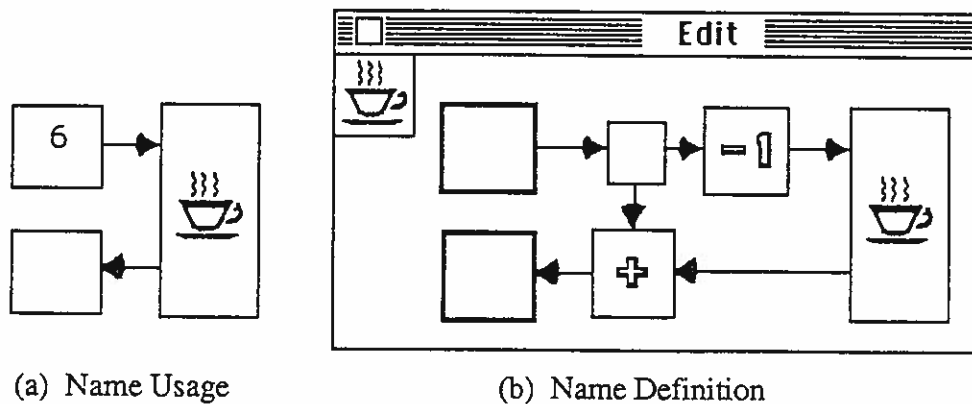
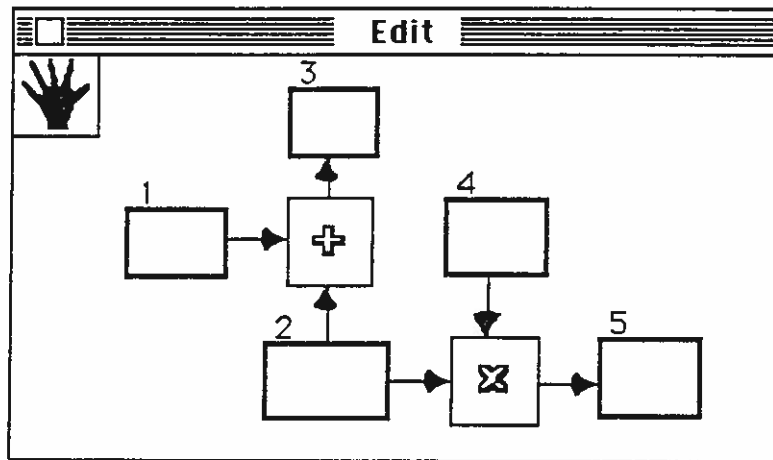


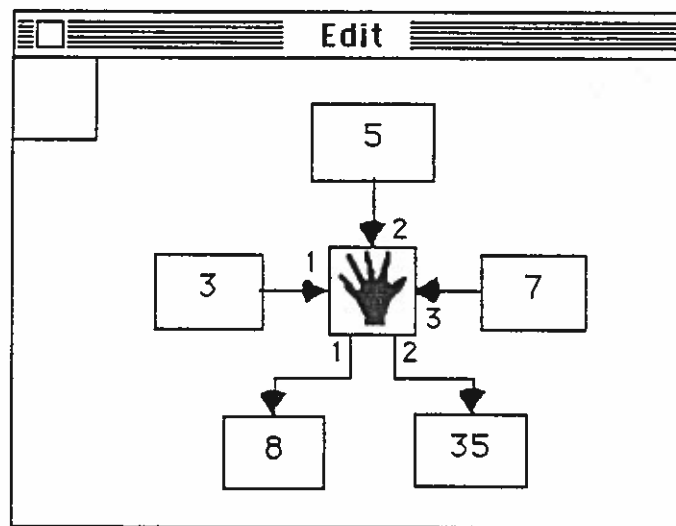
Figure 25: Naming Abstraction

The base boxes in a name definition specify the input and output parameters for the named complex boxgraph. A base box represents an input parameter if it has no incoming arrow, and an output parameter if it has no outgoing arrow. When there is more than one input parameter in a name definition, the association between incoming data values and the input parameters must be established by a *binding rule*.

In MSTL the binding rule is a positional rule in that base boxes in a name definition are lexicographically ordered by the (x,y) -coordinates of the upper-left corners, and arrows incident with an operation box that contains the name are also lexicographically ordered by the (x,y) -coordinates of the intersection points with the box. On Macintosh screen, the x -coordinate increases from left to right and the y -coordinate from top to bottom. In MSTL, the ordering of the base boxes can be shown in the Edit window, as illustrated in Figure 26(a), by executing the menu command "Order Base Boxes".



(a): Ordering of Base Boxes



(b) Ordering of Incoming and Outgoing Arrows

Figure 26: MSTL Binding Rule

When the STL system executes an operation box that contains a name icon, the system creates a new copy of the boxgraph associated with the name, binds the input parameters, executes the boxgraph, and assigns the output parameters to the outgoing arrows of the operation box. If there exists a mismatch between the number of parameters and the number of arrows, the operation box will be evaluated as inconsistent. Since a vertically complex nested boxgraph is expanded dynamically during the execution time, the depth of nesting can be controlled by making the recursive usage of named boxgraph conditional to the input parameters. For example, Figure 27 defines recursively the Σ function which is similar to Figure 25(b) except that the depth of nesting is bounded by the input parameter value. The recursion terminates in this example when the input value becomes non-positive.

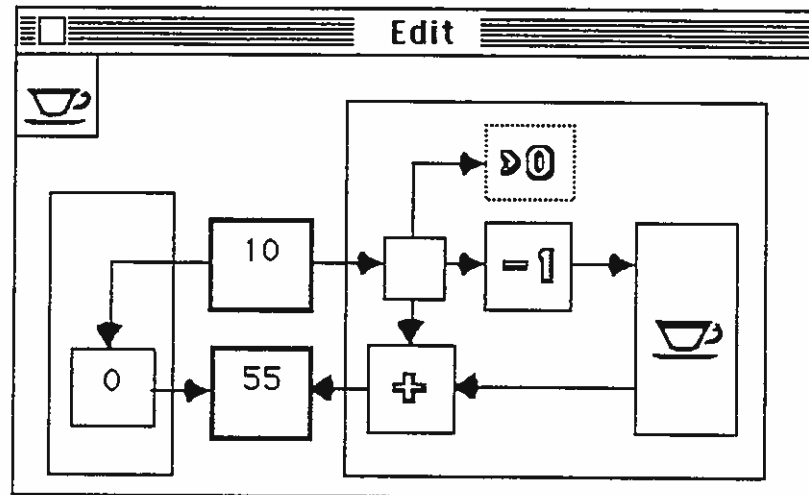


Figure 27: Bounded Expansion of Operation Box

Through naming abstraction, the vertical complexity of a nested boxgraph can be reduced. However, for understanding the semantics of an operation box containing a name icon, it is often necessary to display the definition of the name icon. For that purpose and others, the MSTL system has a *Stop Mark* capability. It corresponds to the break-point operation in a traditional programming system. Figure 28 illustrates the Stop Mark capability using a recursive definition of the factorial function.

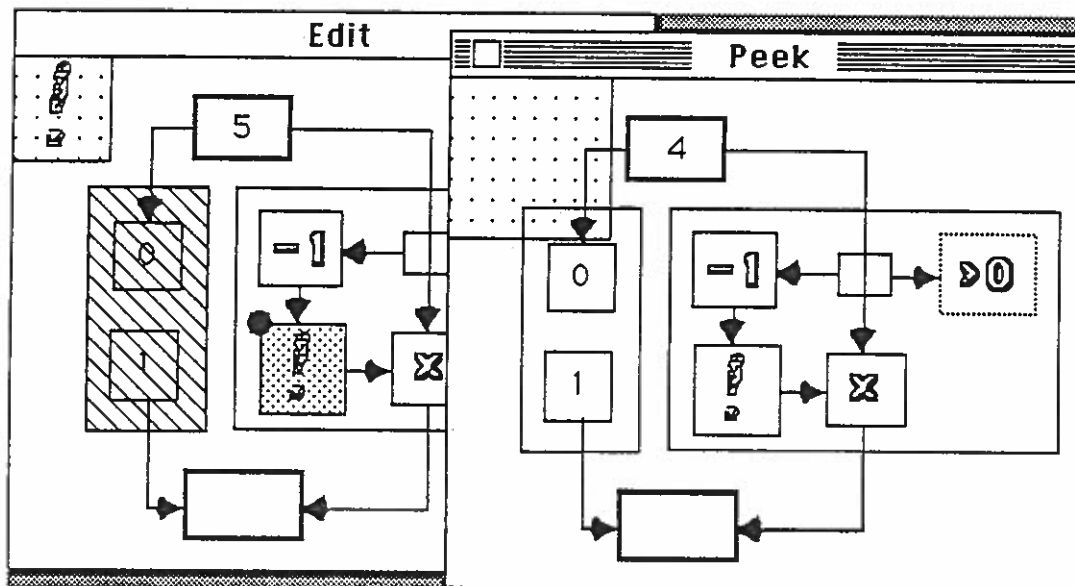
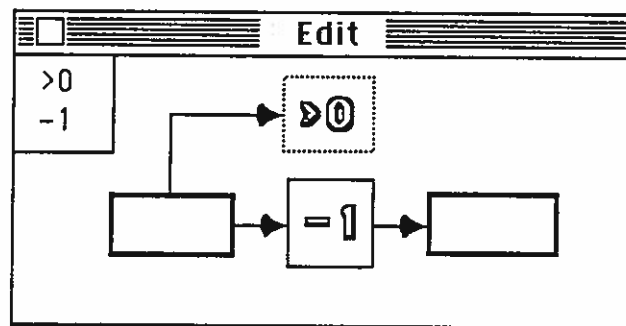


Figure 28: Stop Mark and Peek Window

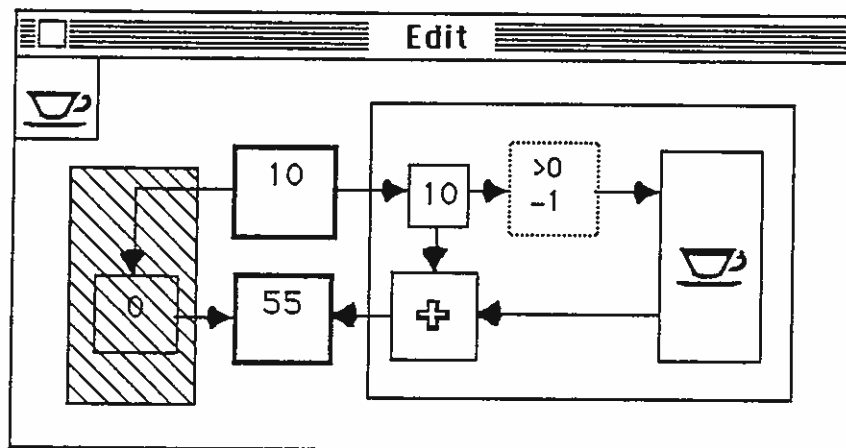
Before solving a puzzle on the current Edit window, the user can set a stopmark on any operation box. The stop mark is indicated by a dot on the upper-left corner of the box. The system stops the execution just before it is ready to execute the marked box. The user has at that point an option of opening a Peek window for the marked box. In the Peek window, the named puzzle (subroutine) will be displayed with the input parameters already being assigned. When the user selects the "Continue" menu command after that, the system continues the execution displaying results of computation on both Edit window (for

main routine) and Peek window (for subroutine). In Figure 28 the Edit window shows that the system stopped before executing the factorial function and the user opened the Peek window for the factorial operation box. The Peek window displays the exact same boxgraph as in the Edit window except for the input value. The dots in the name areas of Edit and Peek windows indicate that the puzzle is currently being executed.

Note that a name icon can be contained either in a closed box or in an open box. When it is contained in a closed box, the closed box is equivalent to a closed complex boxgraph containing the named boxgraph. Similarly, when it is contained in an open box, it is equivalent to an open complex boxgraph. When the named boxgraph is evaluated as inconsistent and the name icon is in an open box, the inconsistency will be propagated into the boxgraph containing the operation box. For example, Figure 29(a) defines a conditional subtraction function and Figure 29(b) uses it to simplify the Σ function of Figure 27. In general a predicate function name is contained in an open box.



(a) Name Definition for Conditional Subtraction



(b) Name Usage of Conditional Subtraction

Figure 29: Open Operation Box

3.3.6 File

When the boxgraph of a puzzle contains at least one base box, the puzzle defines a schema of a *file*. A file in STL is a sequence of known solutions for a puzzle that has base boxes. A solution is a set of data values which makes the puzzle consistent when they are assigned to the base boxes. A solution is a list of data values, and a file is a list of lists. A puzzle in a drawer can save arbitrarily large number of known solutions. When the puzzle consists of base boxes only, and no arrows, it defines a record structure in a traditional programming language, where each base box represents a record field.

An example of simple file definition is given in Figure 30. The puzzle defines a schema for recording golf score. The user fills each base box with proper information through a keyboard and saves the record into the puzzle in a drawer by selecting "Save" menu command. The small box next to the name area indicates how many solutions (records) are currently saved inside the puzzle. Any text outside of the base boxes is a part of the background and should be considered as a comment.

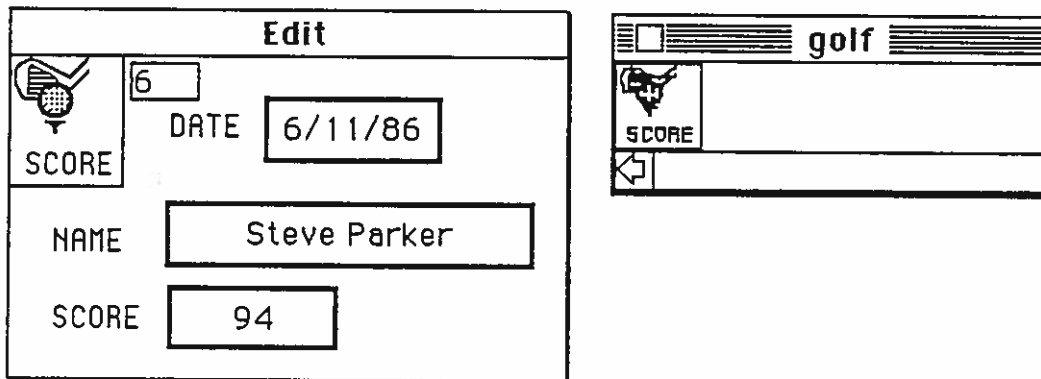


Figure 30: A Simple File Definition

For simple data query there is a browsing mechanism in MSTL to search through a small file, record by record, directly on the Edit window. For selecting a particular record, the puzzle can be modified to include a query condition. Figure 31(a) gives an example of such query specification. The query is "When did Kathy Smith score less than 100?" After the user selects the "Find" menu command, the system completes the boxgraph, as in Figure 31(b), with the first solution in the file that makes the boxgraph consistent. This is a direct method of querying a simple file.

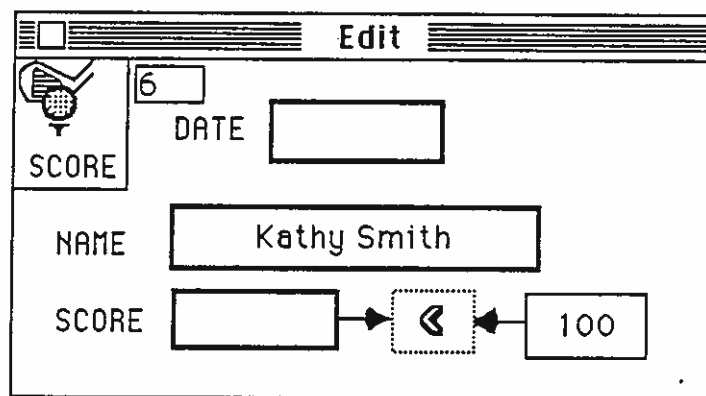


Figure 31(a): Direct Database Query (Specification)

The image shows a window titled "Edit" with a standard Mac OS-style title bar. Inside the window, there are several input fields and labels:

- A small icon of a golf ball with a checkmark is next to a text box containing the number "6".
- A label "DATE" is followed by a text box containing "5/23/86".
- A label "SCORE" is followed by a text box containing "95".
- A label "NAME" is followed by a text box containing "Kathy Smith".
- Below the "SCORE" field, there is a dashed box containing a left-pointing arrow, with another "SCORE" label and a text box containing "100" to its right. Arrows point from the "95" box to the dashed box, and from the dashed box to the "100" box.

Figure 31(b): Direct Database Query (Answer)

There is another method of data query, i.e., through a file box and a structure box. A file box containing a name icon represents the solutions saved in the file of the named puzzle. It is equivalent to a parallel iteration that produces or accepts a sequence of records. In order to represent an individual record a structure box can be used. A structure box has a name area into which the user can insert a puzzle name. The system, then, automatically draw the base boxes of the puzzle, proportionally scaled, inside the structure box. A set of values contained in a structure box is a list data type and can be treated as a single value. A file is a list of lists. The purpose of structure box is the composition and decomposition of a record structure, i.e., a list data structure. Figure 32 illustrates the semantics of a file box in terms of parallel iteration and structure box.

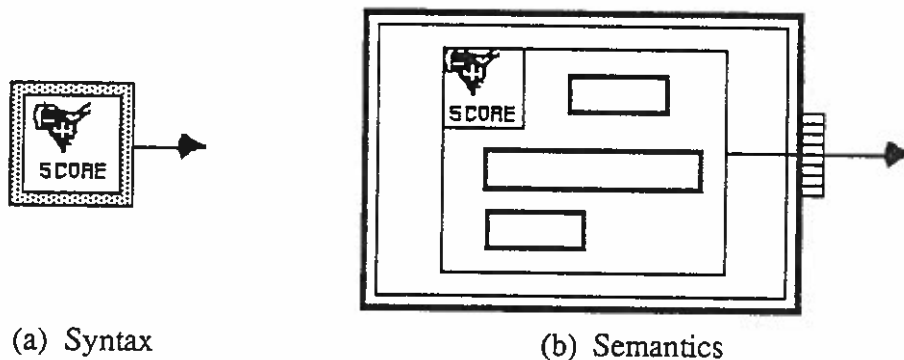


Figure 32: Semantics of File Box

As an example of using a file box the puzzle of Figure 33(a) displays the content of the golf score file on the Score List window as shown in Figure 33(b). Note that the window operation takes a window name as the first parameter and a list value as the second parameter, and only when there is no window open with the same name, the system will create the new window.

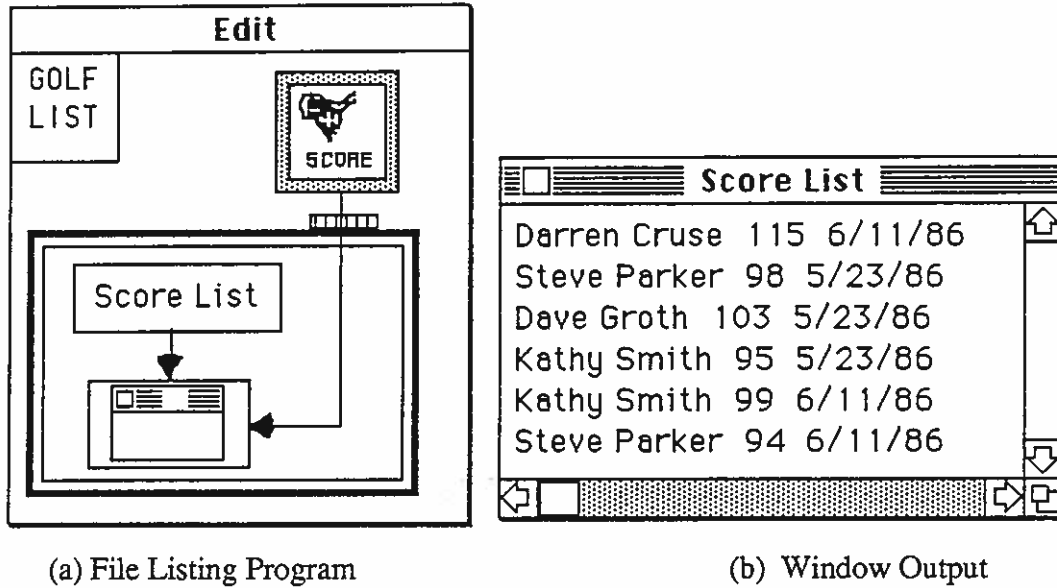


Figure 33: File and Parallel Iteration

Figure 34 illustrates how a structure box can be used to decompose a record structure. The puzzle prints the only records whose score is less than 100.

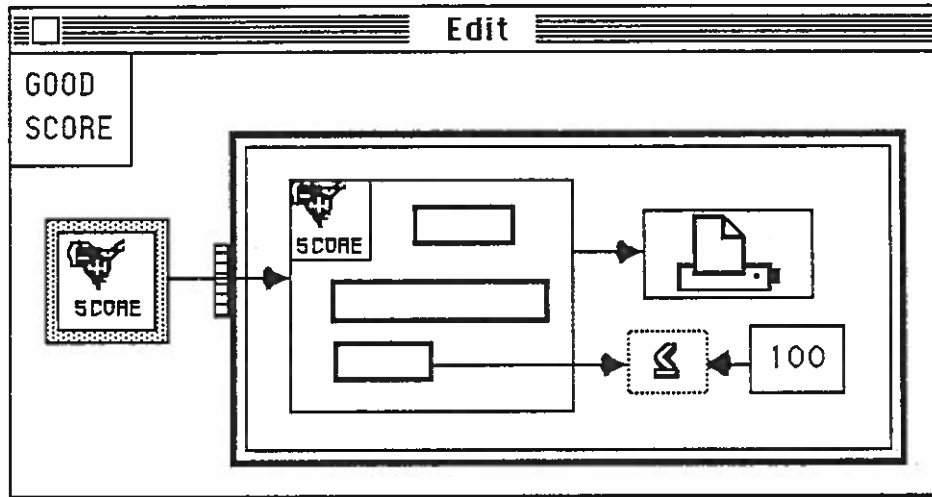
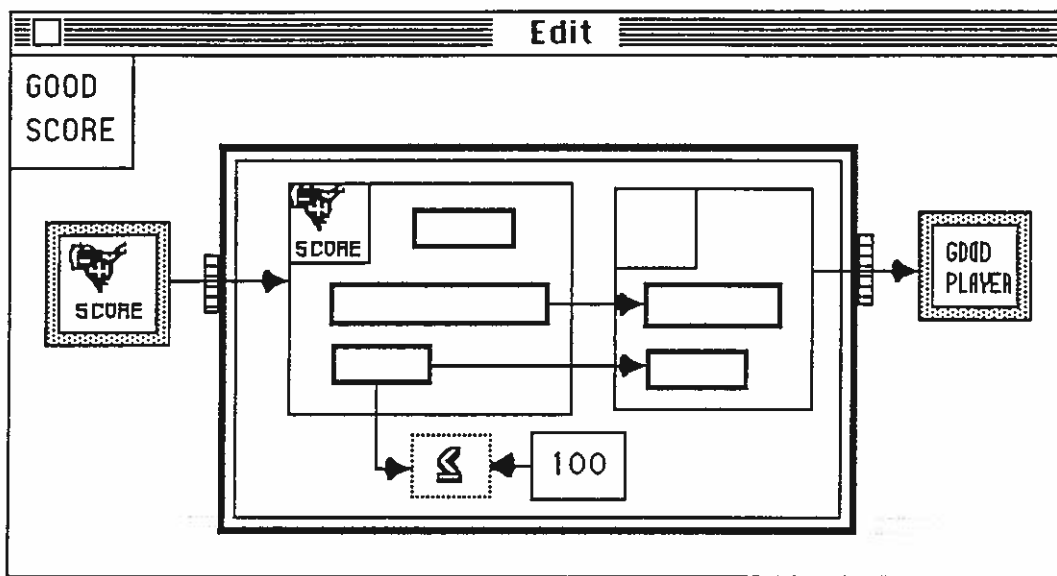
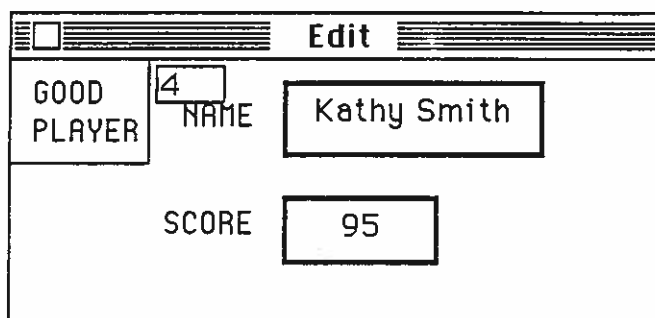


Figure 34: Structure Box for Record Decomposition

A parallel iteration can be used to construct a new file from an old file. For example, Figure 35(a) presents a STL program that constructs a file of good players based on the golf score file. The new file will contain the records of name and score which is less than 100 as shown in Figure 35(b). After the file is created, the user can browse through the file using the direct query method illustrated by Figure 31.



(a): A Program for Construction of New File



(b): The Result of Construction

Figure 35: Construction of New File

Since a file is a relation in the terminology of relational database, the above example shows how the selection and projection operation can be implemented in STL. The join operation requires a parallel iteration with cross product interface. Figure 36 gives an example of the join operation. Assume that two puzzles in Figure 36(a) are already defined. The player-age file contains the records of name and age. Based on the golf score file records the program of Figure 36(b) constructs a file of age and score.

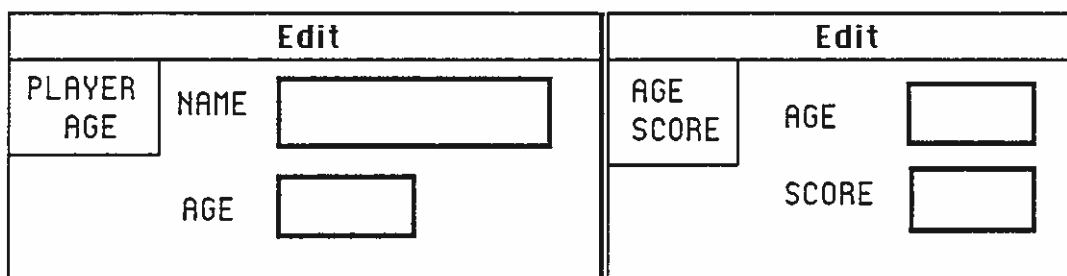


Figure 36 (a): Schemata of New Files

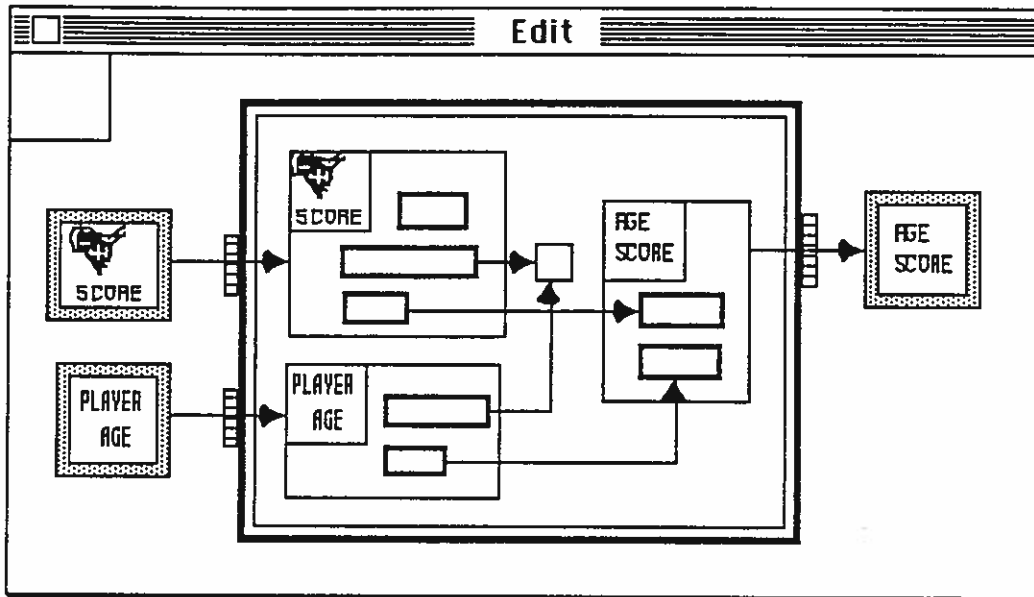
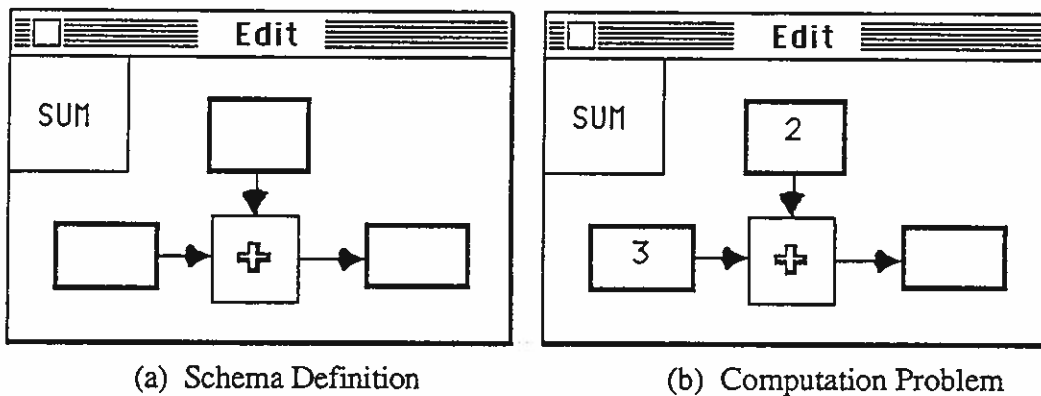


Figure 36 (b): Join Operation

In STL there is no conceptual difference between solving a puzzle by computation and by data query. A puzzle is solved by completion. A simple example will explain the concept. Figure 37(a) is a puzzle to compute the sum of two numbers. It is also a schema of a file whose record structure is an ordered triplet. After solving the puzzle (b) by clicking "Solve" menu command and getting the result (c), it can be saved into the puzzle's file. Now the puzzle (d) can be solved by clicking "Find" to get the same result (c). Note that the puzzle (d) is not computationally solvable in STL.



(a) Schema Definition

(b) Computation Problem

Figure 37: Computation and Data Query

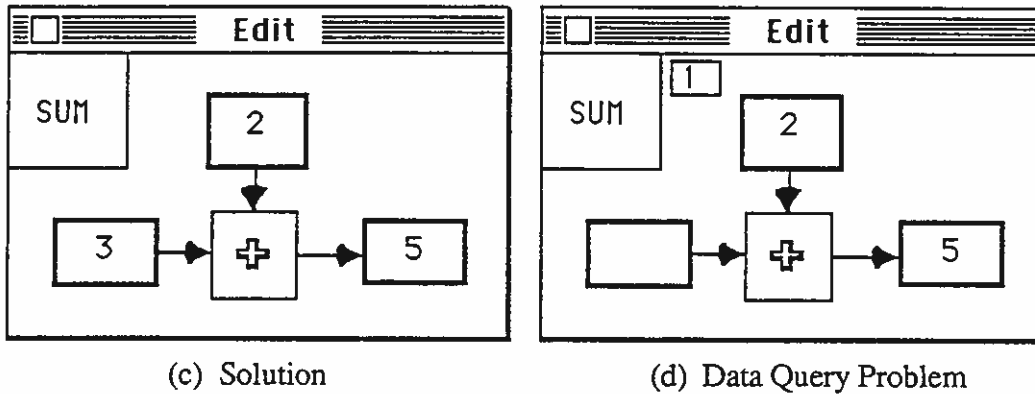


Figure 37: Computation and Data Query

3.3.7 Unquote

Quoting a boxgraph inhibits the system to evaluate the boxgraph. A quoted boxgraph is a data value that can be transferred from one box to another. Unquoting is the inverse operation of quoting, and unquoting a quoted boxgraph is equivalent to evaluating the boxgraph. It corresponds to the apply operation in LISP.

There are two ways of referring to a boxgraph as a data object. One way is to use a quote box containing a boxgraph. The quote box represents its content as a data value. The other way is to use a descriptor box containing an icon name of a puzzle. The descriptor box represents the boxgraph of the named puzzle as a data value. For example, assuming that the puzzle in Figure 38(a) is already defined, both (c) and (d) refers to the same boxgraph of the puzzle in (a). Figure 38(b) will be used later in Figure 40.

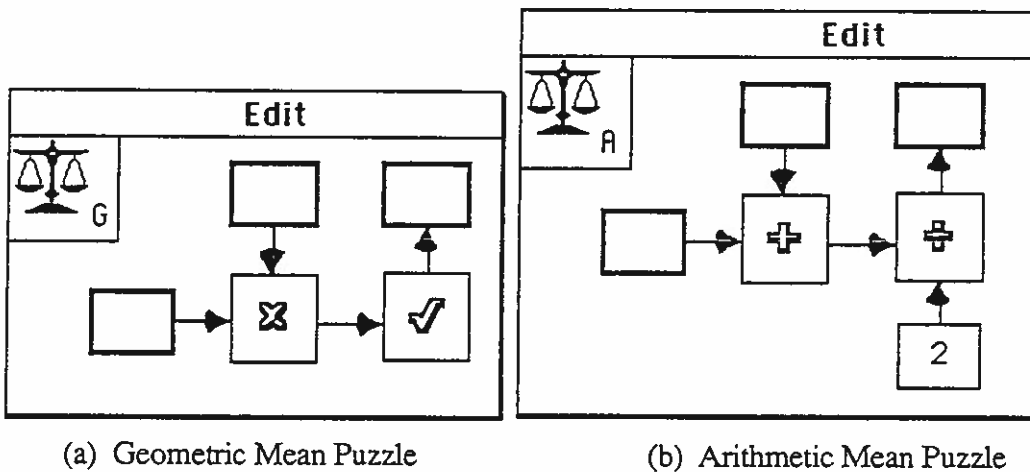


Figure 38: Descriptor Box and Quote Box

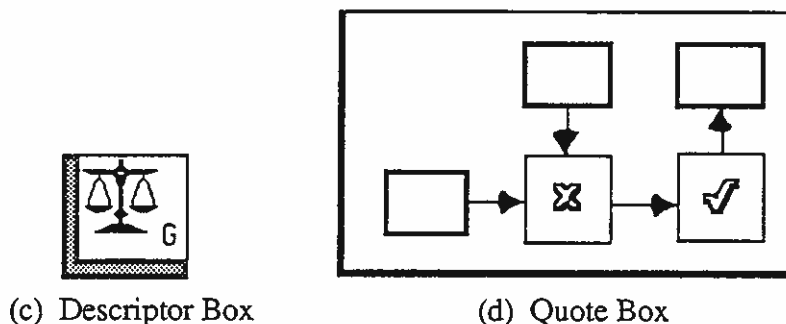


Figure 38: Descriptor Box and Quote Box

An unquote box requires at least one incoming arrow ending at the inside frame for receiving a quoted boxgraph. All other arrows either end at or start from the outside frame of the unquote box. The number of arrows incident with the outside frame should be the same as the number of base boxes (i.e., parameters) contained in the quoted boxgraph. A closed unquote box with a quoted boxgraph is equivalent to a closed complex box containing the same boxgraph except that all base boxes in the boxgraph are connected to the arrows incident to the unquote box. Similarly an open unquote box is equivalent to an open complex box. Figure 39 shows the equivalence for the case of closed unquote box.

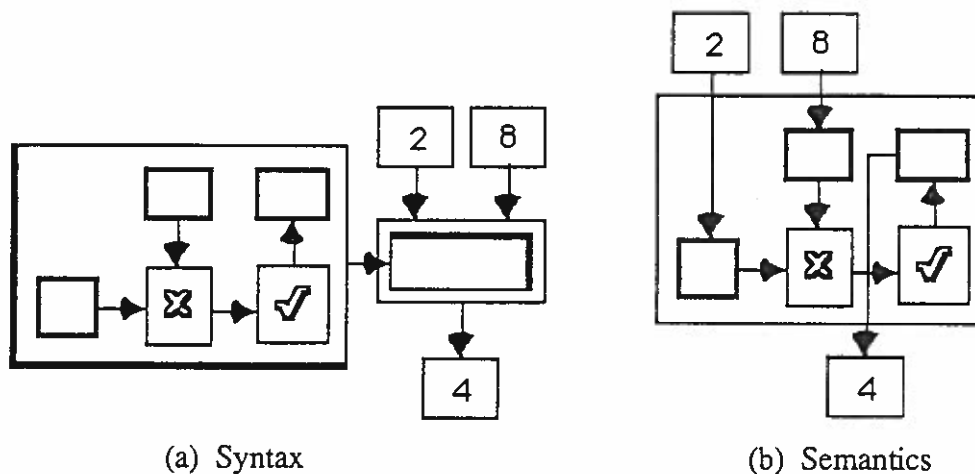


Figure 39: Unquote Box

The main purpose of introducing the unquote operation in STL is to provide the capability of constructing high order functions. Since a boxgraph representing a function can be made as a data value, it can be fed into another function as an argument. It also can be controlled by regular dataflow. A simple application of unquote operation is given in Figure 40. When 'G' is chosen the unquote box computes the geometric mean of two numbers and when 'A' is chosen it computes the arithmetic mean.

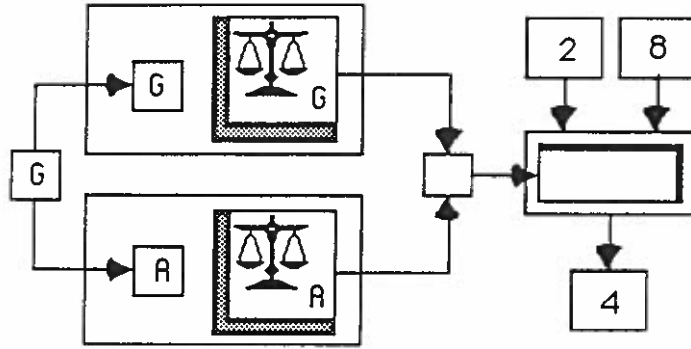


Figure 40: Unquote Application

4. Programming in STL

In this section we will discuss the relationship between STL and traditional programming concepts. First, we will summarize the correspondence between traditional programming constructs and STL constructs. Then, we will discuss the software engineering aspects of STL.

4.1 Programming Constructs

In spite of different syntax and semantics from other programming languages, STL inherits many traditional programming concepts. We will enumerate some of the associations between traditional constructs and STL constructs below, however loose it may be. The list is not meant to be exhaustive.

- Assignment: Dataflow.
- Variable: Empty closed box and empty base box.
Once a value is assigned to a variable, it never changes, i.e., it has the single assignment property.
- Constant: Closed box with data.
- Record: Structure box and base boxes in a named puzzle.
- Array: Iteration box with parallel port.
- Decision: Consistency and dataflow.
There is no Boolean data type. Dataflow is controlled by the consistency of a complex box.
- Sequencing: Data dependency.
No explicit sequencing construct exists.
- Block: Complex box.
Similar to an open subroutine.
- Function: Named puzzle in a closed box.
Parameters are base boxes. The binding rule is positional.
- Predicate: Named puzzle in an open box.
- Iteration: Folding.
- Recursion: Nested naming.
- Task type: Named puzzle.
- Task: Named puzzle in a closed box.
There is no difference between task and procedure in STL.
All activations of functions and predicates are concurrent.
- Synchronization: Data driven.
A box is evaluated as soon as incoming data become available.
- Exception: Inconsistency.
Exception propagation is made possible through an open box.
- Relation: File of a named puzzle.
- Schema: Structure box and base boxes in a named puzzle.
- Selection: Parallel iteration in inner product mode.
- Projection: Parallel iteration in inner product mode.
- Join: Parallel iteration in cross product mode.
- Apply operation: Unquoting.

There is no representation of the following important programming concepts in STL:

- Type checking capability.
- Abstract data type.
- Package.
- Generics or parameterized macros.

4.2 Programming Aspects

A visual programming requires a new approach to software design and implementation. We will discuss here some software engineering aspects of STL programming.

4.2.1 Picture Programs

The most distinct feature of STL is a visual presentation of information. In STL an identifier is a picture and so is a comment. A program itself is a picture and so is a datum. A text is a special form of picture. A database also can contain visual data. Figure 41 give some examples of picture program and database.

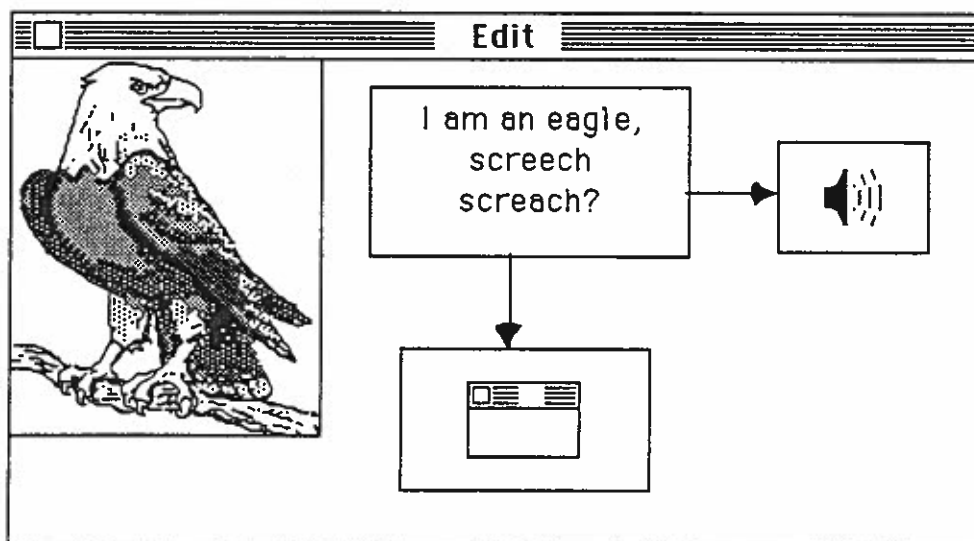


Figure 41(a): Pictorial Program

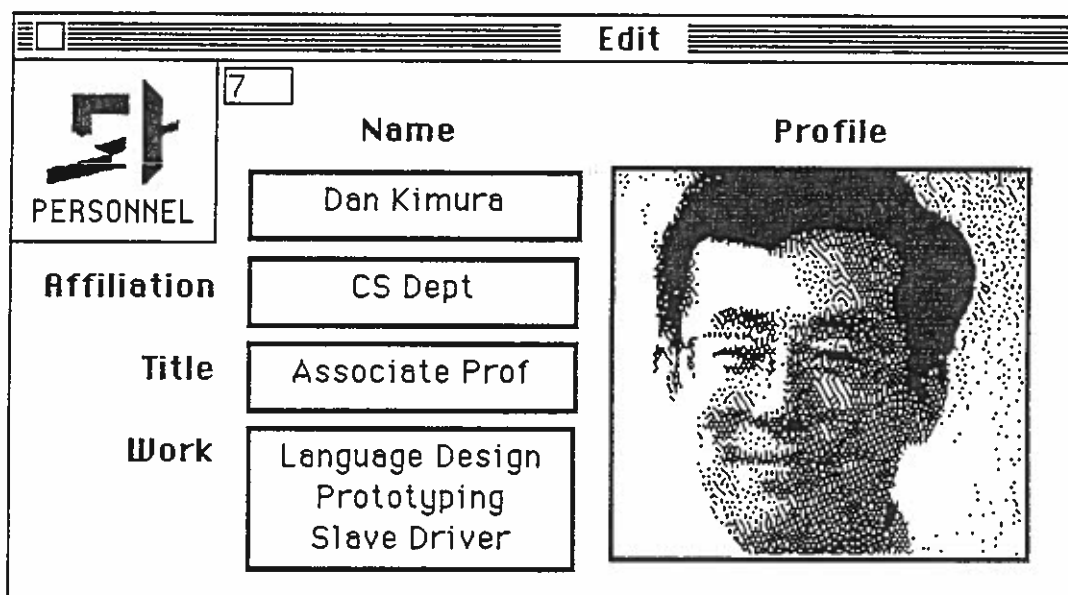


Figure 41(b): Show and Tell Personnel Database

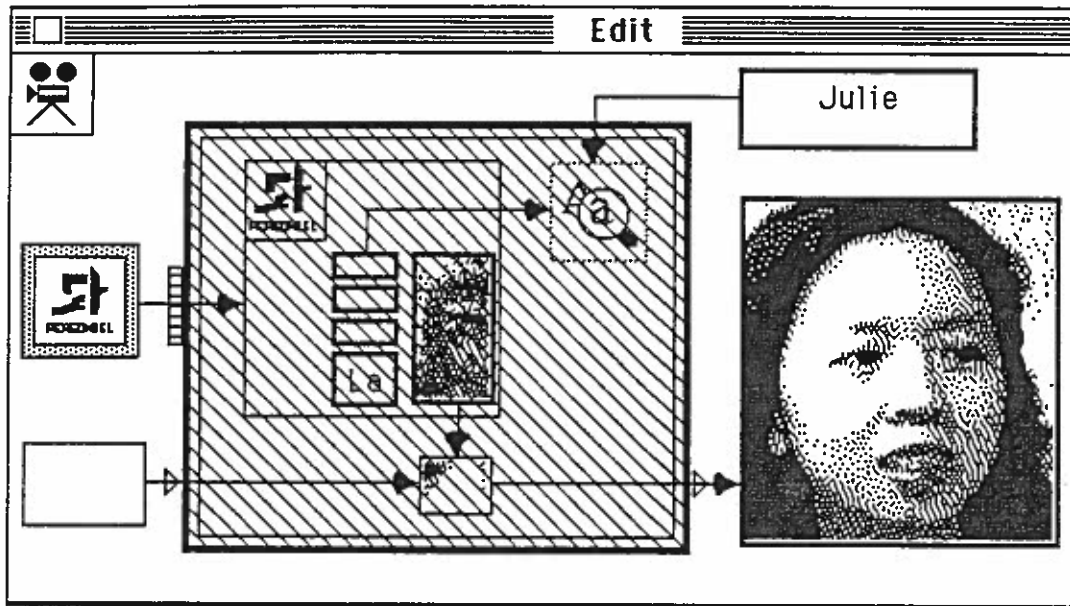


Figure 41(c): Data Query Program

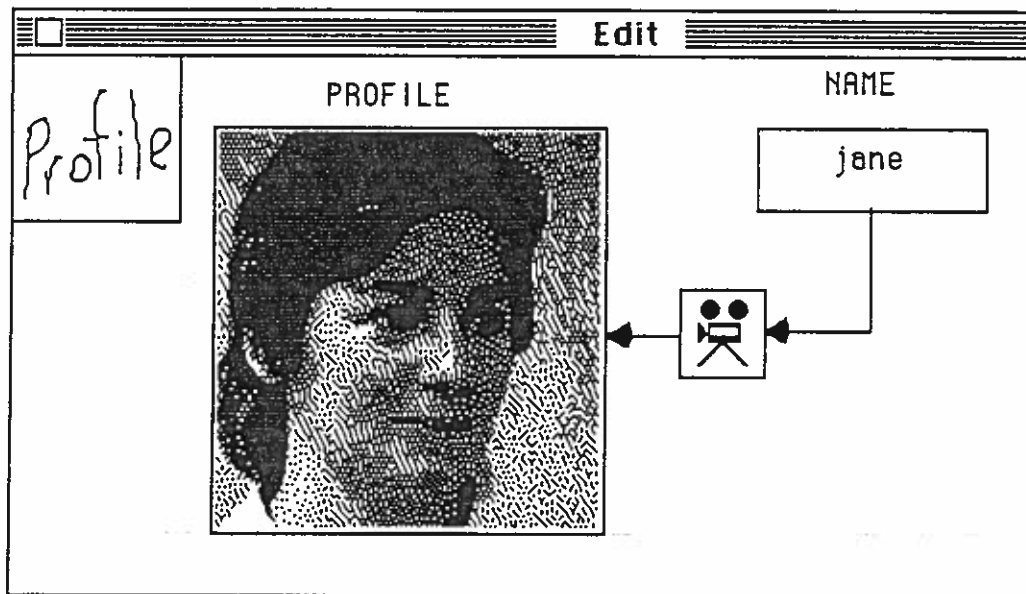


Figure 41(d): Data Query Function

Figure 41(a) is self explanatory. Figure 41(b) is a record of the Show and Tell personnel database. Figure 41(c) is a data query program to find a profile of a person whose name contains a text entered as the input parameter. Figure 41(d) shows how the data query program can be used as a function to generate a person's profile.

4.2.2 Structured Programming

The abstraction mechanisms of STL are limited to control abstraction. Data abstraction is absent from STL. Even though it is generally assumed that a pictorial language is easier to understand, without careful layout, even a pictorial program can be made difficult to understand. For example Figure 42 and Figure 38(b) are logically equivalent puzzles, but Figure 42 is harder to understand. Obviously structured programming in a two-dimensional language requires a different approach from that of a one-dimensional language.

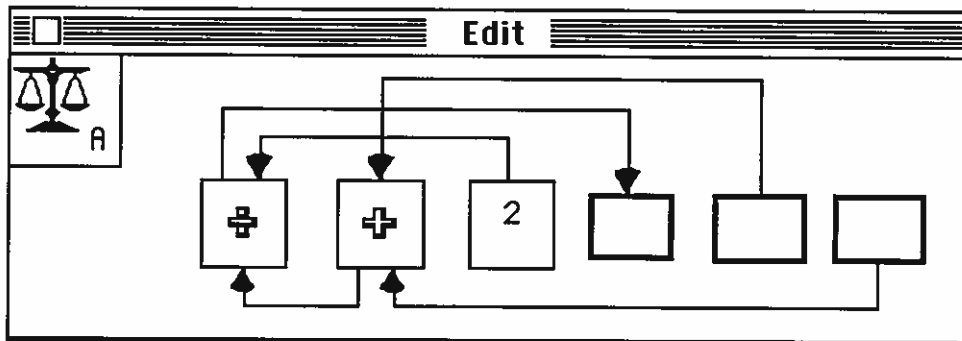


Figure 42: Program Understandability in STL

4.2.3 Assertions

The concept of consistency is fundamental in the STL semantics. It is a declarative concept rather than an imperative one. (See [3] for more detailed discussion of the declarative nature of STL semantics.) Assertion is a declaration of an attribute on computational states. It can be incorporated in STL as a predicate in a closed box. If the predicate fails at some point of computation the box will be hatched indicating the failure, but the computation will continue because the failure is contained in the local closed box. Figure 43(a) gives an example of assertion.

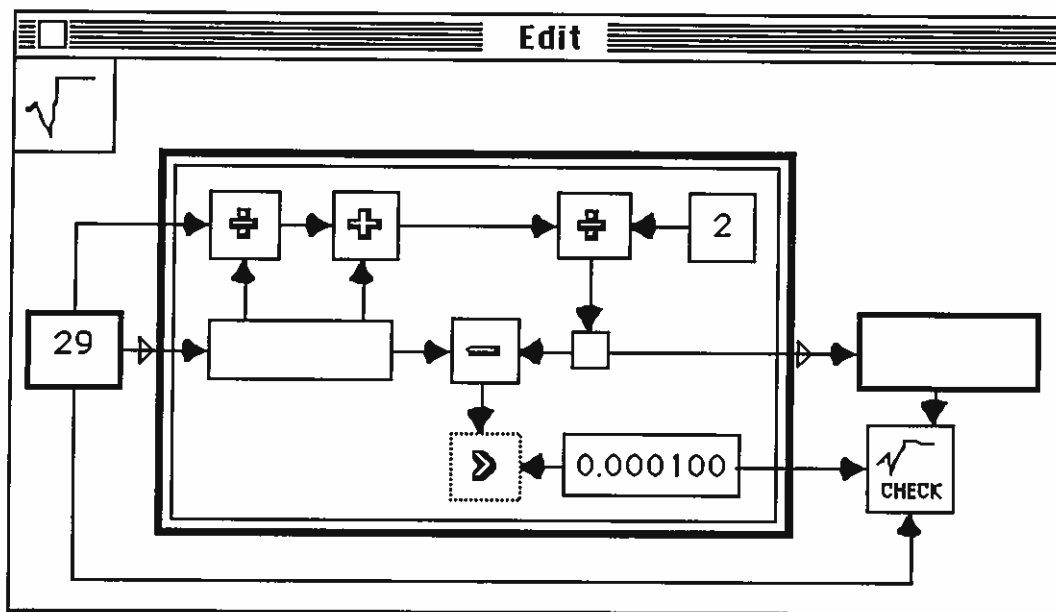


Figure 43(a): Squareroot Function with Assertion

The puzzle of Figure 43(a) computes $y = \text{SQRT}(x)$ with accuracy ϵ where $x=29$ and $\epsilon=0.0001$. It also checks whether the answer is correct or not by testing: $x+2\epsilon y > y^2 > x$. The predicate is defined in Figure 43(b). The computation is correct for $x=29$ as shown in Figure 43(c). If there is an error in the algorithm as in Figure 43(d) where the constant 3 is used in stead of 2, the checking predicate fails and the failure is displayed by hatching the predicate box.

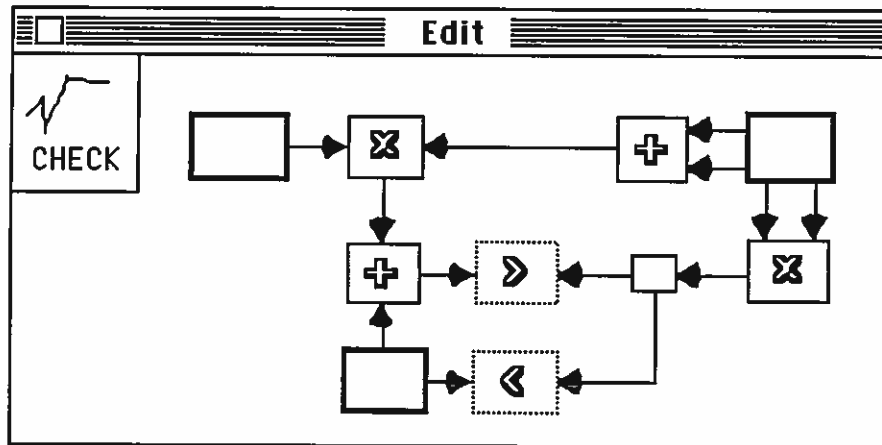


Figure 43(b): Assertion Predicate

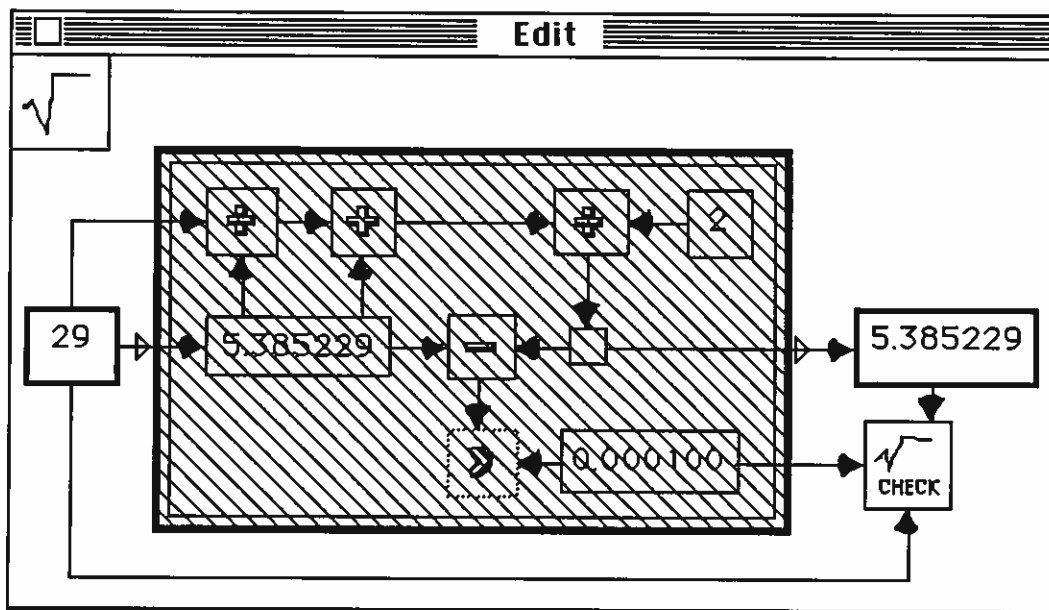


Figure 43(c): Correct Computation

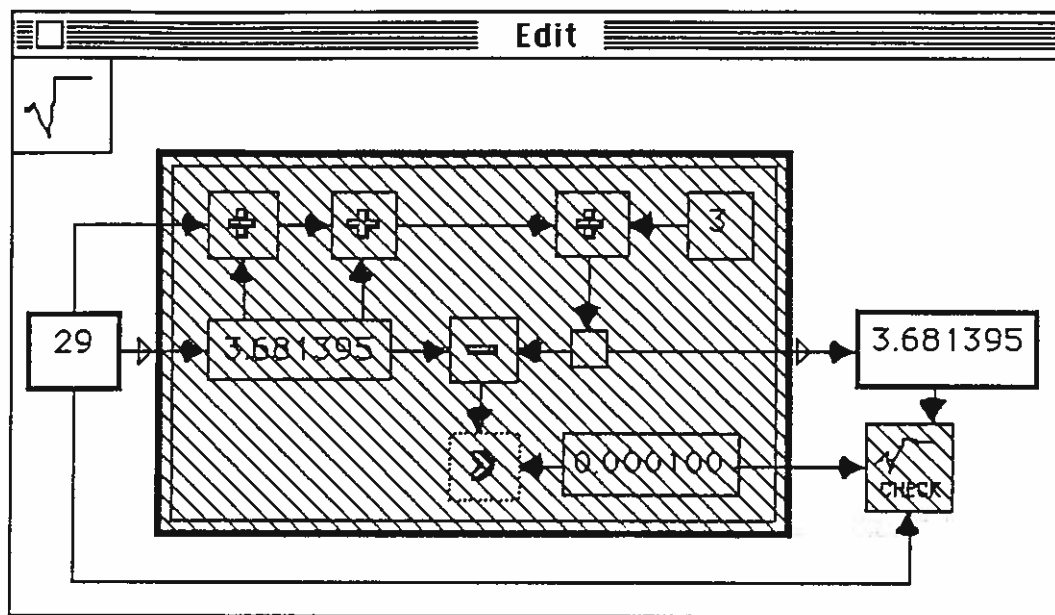


Figure 43(d): Incorrect Computation

5. Conclusions and Future Directions

Visual programming (VP) is a new research area of importance. It is important because it will contribute to make a computer accessible to everyone and will contribute to reduce the software cost, and it will present a new set of research problems. The fundamental issue in VP is the visualization of programming concepts. The issue is how to represent data structures and algorithms in such a way that a computer user can interact with a computation closely and efficiently through high resolution graphics.

We have introduced the term *keyboardless programming* as a paraphrase of VP assuming that when we minimize the usage of a keyboard as a text generator, we maximize the visualization of programming concepts. We have designed and implemented a visual language Show and Tell for keyboardless programming to investigate how far we can achieve this goal. Our results show that it is possible to eliminate entire keyboard usage from programming (except for data entry), if it is for computation and database query.

This does not show, however, that Show and Tell is a better visual programming language. Sometimes VP is tedious even for novice end users, and traditional textual programming can be simpler. For example, arithmetic expressions, in a textual form, are in general much easier to construct, assuming that the programmer knows how to use a keyboard, and easier to understand, than a dataflow visual representation. For another example a data query specification is simpler if we can combine a textual specification with a visual specification, as compared in Figure 44(a) and (b).

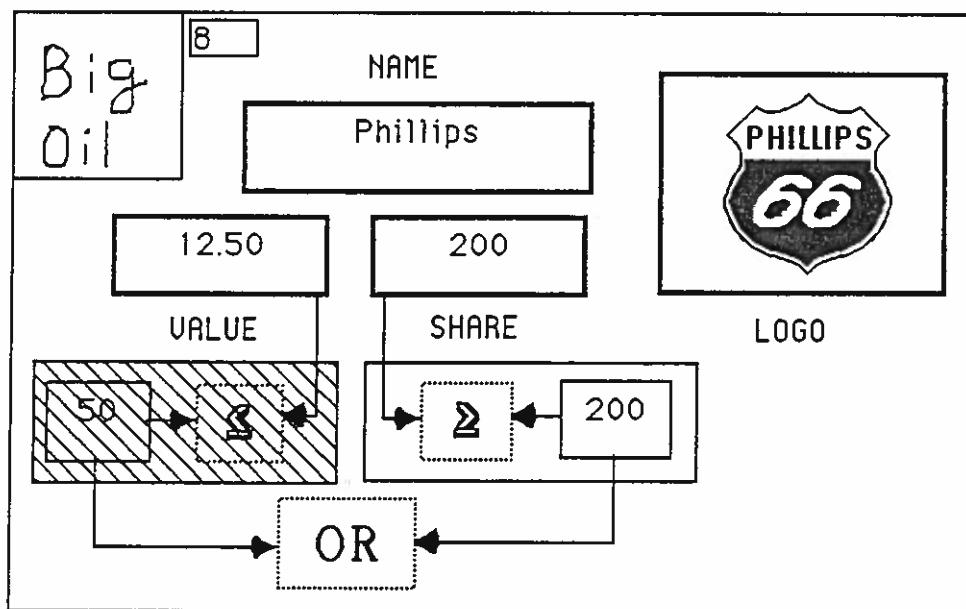


Figure 44(a): Query Definition and Result in MSTL

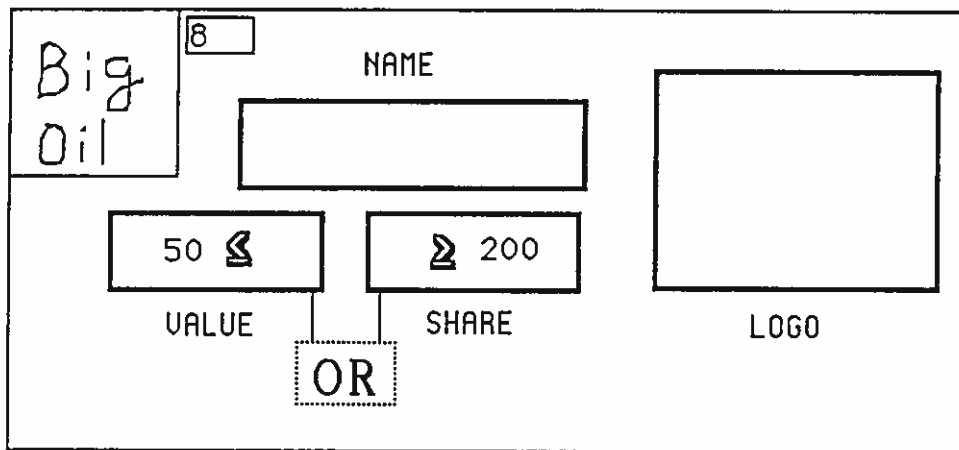


Figure 44(b): Textual Query Definition in Visual Schema

Figure 44(a) is a query definition and a result of query in MSTL. The query portion of this puzzle requires seven boxes and six arrows. In contrast, Figure 44(b), which is based on a possible improvement of MSTL, requires only one box and two arrows. Thus for a simple query definition like this a compromise between textual and visual programming will be meaningful.

We have shown that there are new programming concepts that are unique to visual programming. Folding and parallel iteration are such examples. These are new because they are spatial concepts. The notion of space is not established in traditional programming languages. Another example of spatial constructs in STL is a propagation of inconsistency inside a closed box. Its mode of communication is broadcasting and broadcasting is a space oriented concept. Future investigation of spatial programming concepts will be more urgent in visual programming than in textual programming. In particular, construction of a computation model in which the notion of space plays an important role is a research problem in VP that requires immediate attention.

Another research problem in VP is a formal study of two-dimensional languages. Our proposal of two-dimensional grammar with concatenation and superposition operations, requires more formal and rigorous treatment, in order that a general parsing theory and a new parsing algorithm can be developed. For example, when a two-dimensional formal grammar is established, our systolic approach for parsing, reported in [10], will become a more general parsing theory applicable to other spatial visual languages. Note that systolic parsing is more meaningful for visual programming languages than others. In [11], we show that a major subset of STL boxgraphs can be formally specified by an index grammar.

STL is not a fixed language. It will evolve into another visual programming language after it is thoroughly evaluated. However, some extensions are already under our consideration at the present time. One possible extension of STL is to introduce more variety of arrow types. There are three new arrow types being considered: *control* arrow, *excitatory* arrow, and *inhibitory* arrow.

(1) Control arrow: This introduces sequential control flow into STL directly. The destination box will be executable only after the source box is completed. If a box has many incoming control arrows, it has to wait until all the source boxes are completed. This extension makes it possible to sequence already existing puzzles without modifying them. Figure 45 illustrates the concept of control arrow which is represented by a double lined arrow.

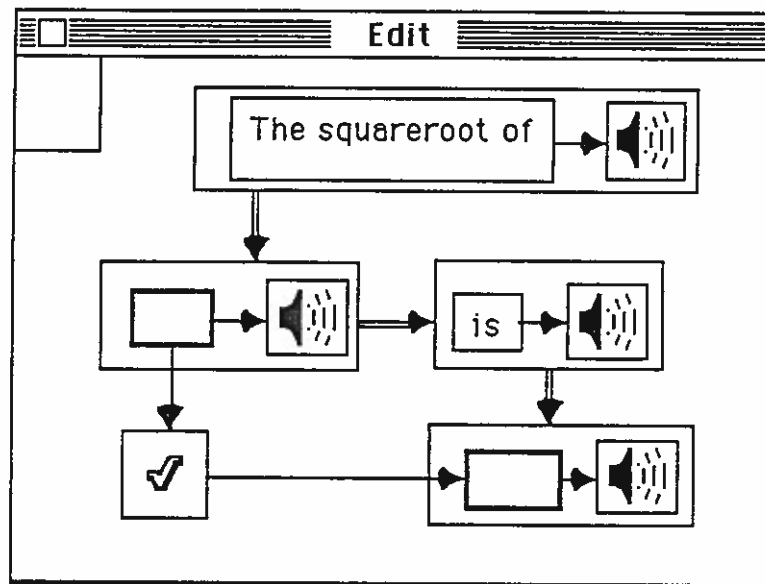


Figure 45: An Example of Control Arrow

(2) Excitory arrow⁺ : This propagates the consistency of a box to another. When the source box is inconsistent the destination box will be unconditionally inconsistent. When the source is consistent the arrow has no effect on the destination box, i.e., the destination may be or may not be consistent, depending on the result of evaluating the destination box. When more than one excitory arrow comes into a box, the box will be inconsistent if one of the source boxes is inconsistent, otherwise the arrows have no effects. This helps to ease some layout problems involving complex boxgraphs.

(3) Inhibitory arrow⁺⁺ : This is similar to excitory arrow. For multiple inhibitory arrows coming into a box, when one of the source boxes is consistent, the destination box will be unconditionally inconsistent. Otherwise the arrows have no effect on the destination box. This simplifies representation of the if-then-else construct in STL. For example the factorial function of Figure 28 can be simplified into the one in Figure 46 in which an inhibitory arrow is represented by a dotted line with a hollow arrow head.

Note that all three arrow types proposed above can be simulated by the current STL. The motivation for the extension is not to enhance the computation power of STL but to enhance the understandability and simplicity of STL programs.

⁺ This arrow type was suggested by Professor Marvin Minsky of MIT,

⁺⁺ This arrow type was suggested by Professor Gyula Mago of University of North Carolina.

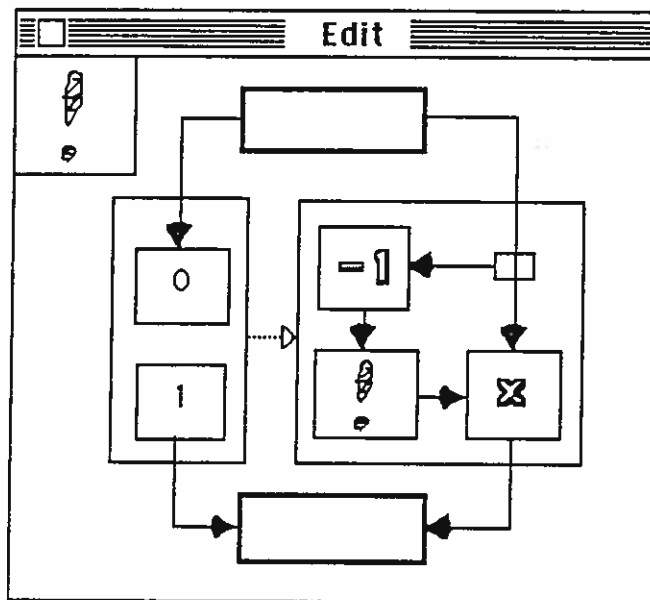


Figure 46: Factorial Function with Inhibitory Arrow

The current version of MSTL is an interpreter of STL. The performance consideration was secondary to the conceptual feasibility of keyboardless programming. Now having established the feasibility, we are addressing the performance issue. As one of the steps to improve the performance of MSTL we designed and implemented a prototype MSTL compiler. See [12] for the detail. The performance of the current prototype compiler is not yet sufficient for business or educational applications. Development of a compiler version of STL is needed for performance enhancement. Particularly the compiler development is essential if and when we extend STL into a system programming language.

The declarative semantics of STL in [3] encourages us to investigate the relationship between STL and other declarative languages. One such investigation can be pursued by extending STL into *Picture LISP* and *Picture Prolog*. While basic components of LISP are already incorporated into STL by design, it is not the case for Prolog. A visual language for logic programming will require new ideas beyond those provided in STL.

Finally a design and implementation of a STL system drawer for *Picture Logo* will demonstrate a contribution of STL in educational application, which was, after all, the initial motivation of developing a visual programming language.

6. References

- [1] Raeder, G. A Survey of Current Graphical Programming Techniques. *IEEE Computer*, Vol. 18, No. 8, August 1985, pp. 11-25.
- [2] Toffler, A. *The third wave*. Morrow, New York, 1980.
- [3] Kimura, T.D. Determinacy of Hierarchical Dataflow Model: A Computation Model for Visual Programming. Technical Report WUCS-86-5, Department of Computer Science, Washington University, St. Louis, March 1986.
- [4] Apple Computer, Inc. *Inside Macintosh, Volumes I, II, and III*. Addison-Wesley, 1985.
- [5] McLain, P. and T.D. Kimura. Show and Tell User's Manual. Technical Report WUCS-86-4, Department of Computer Science, Washington University, St. Louis, February 1986.
- [6] Kimura, T.D. and J.W. Choi. Modular programming in the C language. Show and Tell Project Memo, Department of Computer Science, Washington University, St. Louis, March 1986.
- [7] Davis, A.L. and R.M. Keller. Data Flow Program Graphs. *IEEE Computer*, 15:2 (26-41), February 1982.
- [8] Hebb, D.O. *The organization of behavior*. Wiley, New York, 1949.
- [9] Kimura, T.D. Completion Problem and Its Solution for Context-Free Languages (Algebraic Approach). Moore School Report 72-09, University of Pennsylvania, Philadelphia, PA., May 1971.
- [10] Bojanczyk, A.W. and T.D. Kimura. A Systolic Parsing Algorithm for a Visual Programming Language. To appear in *Proc of FJCC 86*, Dallas, Texas, November 1986.
- [11] Gillett, W.D. and T.D. Kimura. Parsing Two-dimensional Languages. To appear in *Proc. of COMPSAC86*, Chicago. October 1986.
- [12] Choi, J.W. and T.D. Kimura. A compiler for a two-dimensional programming language. To appear in *Proc. of ACM Symposium on Small Systems*, San Francisco, December, 1986.