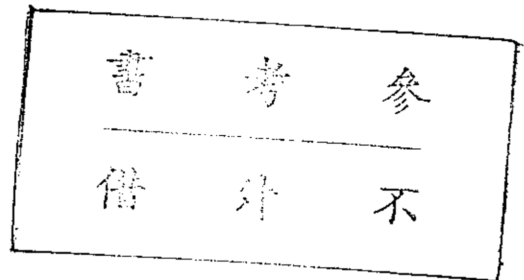
TR-87-004

A Visual Approach to Automatic Program Synthesizer

KUO-YANG CHENG



中研院資訊所圖書室
3 0330 03 000064 5

中央研究院 資訊科學研究所

0064

PART ONE

A Visual Approach to Automatic Program Synthesizer

摘　要

傳統的程式設計環境，著重於如何提供一些方便和有用的工具來讓使用者很方便地發展其應用程式。然而，這對不諳於寫程式的專家而言，似乎無法讓他們有效地發展自已的應用程式。另一方面，在人工智慧語言的程式設計環境下，這些不諳程式設計的專家也很難將他們的專業知識表達得淋漓盡致。這種在發展應用程式時，所面臨應選擇何種對使用者較為友善的設計環境之問題，在一些像辦公室自動化系統、電腦輔助教學系統、和以螢幕為主體的資訊系統上，常常會遇到。為了解決這種選擇和表達上的困擾，本論文提出一種視覺性的程式設計環境。在此環境下，這些不諳於寫程式的專家就可以用視覺的方式，很容易地去發展或執行能表現他們專業知識的應用程式。

本論文研究的目的在探討自動化程式合成系統設計上的一種新嘗試。在此自動化程式合成系統上，本論文採用視覺表格來做為基本的資料或程式媒體，以呈現包括文字、圖形（線段或筆圖）、動畫或規則等之資訊。在此視覺性的程式設計環境下，本論文提供一種視覺表格模式，來將視覺表格之視覺性質與應用程式之圖形架構收納於本系統中。根據此一模式，本系統整合了視覺表格定義語言、圖形軟體與視覺 Prolog 翻譯程式。經由視覺表格語言之運作，使用者的應用程式得以自動地被合成一種內部結構，此結構包含了資訊媒體的性質與應用程式在執行時之流程。至於視覺表格之圖形設計，則由圖形軟體來處理。

為了解釋視覺表格語言合成的內部結構、未來和知識基系統的相連，以及考慮可攜性等因素，本論文亦提出一種建構在Prolog之上的視覺Prolog 翻譯程式，並研擬一組對換演算法，以將視覺表格語言合成之內部結構轉換成可執行的Prolog程式。

ABSTRACT

Traditional programming environment concerns how to provide the programmers some convenient and useful tools to develop their application programs. Under this environment, however, its provision seems not very convenient for Non-Programmer Professionals (NPPs) to develop their own application programs effectively. On the other hand, under AI language programming environment alone, it seems still very difficult for NPP users to handle their specialized knowledge very well. Such a problem arises in the selection of a user-friendly environment encountered in the design of some applications, in particular in the areas of Office Information System, Computer Aided Instructions, and screen-oriented information systems. In order to solve this problem, a visual programming environment is proposed. Under this environment, the NPP users can easily develop and execute their own application programs that simulate their specialized knowledge in visual.

The research purpose of this thesis is to explore a new approach to the design of an Automatic Program-Synthesizing (APS) system for NPP users. In this thesis, visual forms (V-Forms) are used as the fundamental data/programming objects of APS system. The informant presentation data of APS system may contain text, static graphics (line-drawing or bit-map), dynamic graphics (animation), or rules. They are organized as V-Forms. In this environment, a V-Form model is proposed to represent the visual properties of V-Forms and to support a graphical structure of the applications as described by the NPP users.

Based on the proposed V-Form model, a V-Form Definition Language (VDL), a prototype graphics utility, and a V-Prolog interpreter are integrated into the APS system. Through the VDL, an user-designed application can be self-synthesized into a consistent internal structure which holds the properties of knowledgable objects and the execution flow of the desired application. The prototype graphics utility is used to support the graphical entry of V-Forms. When applying a mapping algorithm which transforms the self-synthesized internal structure into Prolog programs, the user-designed application becomes executable by the V-Prolog interpreter.

TABLE OF CONTENTS

CHAPTER	1.	INTRODUCTION	1
	1.1	WHY VISUAL PROGRAMMING	1
	1.2	ORGANIZATION OF THIS THESIS	3
CHAPTER	2.	VISUAL PROGRAMMING ENVIRONMENT	5
	2.1	INTRODUCTION	5
	2.2	RELATED RESEARCHES	5
		2.2.1 Form Language	6
		2.2.2 QBE/OBE	8
		2.2.3 FORMAL	9
		2.2.4 State Transition Diagram Language	10
	2.3	THE VISUAL PROGRAMMING ENVIRONMENT	11
	2.4	OVERVIEW OF THE SYSTEM CONFIGURATION	12
		2.4.1 System Architecture	12
		2.4.2 Main Features of Visual Programming Approach	15
CHAPTER	3.	ON VISUAL LANGUAGE	18
	3.1	INTRODUCTION	18
	3.2	BACKGROUND	18
		3.2.1 Closures	19
		3.2.2 Phrase-Structure Grammars/Languages	20
		3.2.3 Inferred Grammar	22
		3.2.4 Context-Free Programmed Grammar	23
		3.2.5 Non-Procedural Language	24
	3.3	THE DEFINITION OF VISUAL LANGUAGE	25
		3.3.1 Related Researches	25
		3.3.2 Formal Definition of Visual Language	26
	3.4	WHY A V-FORM APPROACH	28
	3.5	V-FORM MODEL	30

3.5.1 External Structure 30
3.5.2 Internal Structure 35

CHAPTER 4. V-FORM DEFINITION LANGUAGE 37
4.1 INTRODUCTION 37
4.2 THE SKELETON PHASE 38
4.2.1 BNF Syntax of the Skeleton Phase 39
4.2.2 Primitive Operations of the Skeleton Phase 40
4.2.2.1 Initialization 40
4.2.2.2 Window Operations 40
4.2.2.3 Flow Manipulation 42
4.2.2.4 Type Redefining 44
4.2.2.5 Instance Generation 45
4.2.2.6 File Handling 45
4.2.2.7 Miscellaneous 46
4.3 THE EDITING PHASE 47
4.3.1 BNF Syntax of the Editing Phase 47
4.3.2 Primitive Operations Of the Editing Phase 49
4.3.2.1 Content Management 49
4.3.2.2 Procedure Processing 50
4.3.2.3 Flow Manipulation 51
4.3.2.4 Miscellaneous 51
4.4 A PROTOTYPE GRAPHICS UTILITY 53
4.5 AN ILLUSTRATIVE EXAMPLE 55
4.5.1 Problem Statement 55
4.5.2 Start up 60
4.5.3 Interactive Sequence 61

CHAPTER 5. INTERPRETING V-FORMS UNDER PROLOG 63
5.1 INTRODUCTION 63
5.2 BACKGROUND 64
5.2.1 Predicate 64
5.2.2 Unification 65
5.2.3 Backtracking 65

5.2.4 CProlog Interpreter 66
5.3 THE V-PROLOG INTERPRETER 67
5.3.1 The Machine-Independent System Predicates 68
5.3.1.1 Procedure Evaluation 68
5.3.1.2 Control Strategy 69
5.3.1.3 Database Management 69
5.3.1.4 Window Manipulation 70
5.3.2 The Machine-Dependent System Predicates 70
5.3.2.1 Screen Management/Vedio Control 71
5.3.2.2 System Communication 71
5.3.2.3 Miscellaneous 72
5.4 INTERPRETING ENVIRONMENT 72
5.4.1 Mapping Algorithm 72
5.4.2 An Example 76

CHAPTER 6. CONCLUDING REMARKS 80

APPENDIX A. INSTALLATION OF APS SYSTEM 83

APPENDIX B. AN EXAMPLE ON INTERACTIVE SEQUENCE 87

REFERENCES 94

FIGURE

2.3.1	Visual Process vs. Traditional Programming Process	12
2.4.1	Basic Architecture of APS System	13
2.4.2	System Architecture	14
3.5.1	The First V-Form F1 of an Application CAI_Course	31
3.5.2	The Hierarchical Structure for the Application CAI_Course	33
3.5.3	The Template for the CAI_Course	33
3.5.4	The Internal Structure of the V-Form F1 of fig. 3.5.1	36
4.2.1	Block Diagram of VDL Interactive Sequence	38
4.2.2	After Using EXTEND and a series of OPEN commands on VFORM:Yes of fig. 3.5.3	43
4.3.1	An Example Showing the Usage of ALIAS	53
4.4.1	A Typical Screen Layout of the Graphics Utility	54
4.5.1	The Application CAI_Course	56
4.5.2	Screen Layout of the Proposed APS System	60
4.5.3	Screen Layout of VDL	61
4.5.4	Control Flow of CAI_Course	62
5.3.1	The Structure of V-Prolog	67
5.4.1	A Typical Screen During Execution	79

TABLE

1.1.1	Categories About The User Applications	2
4.2.1	Syntax of the Primitive Operations in VDL Skeleton Phase	39
4.3.1	Syntax of the Primitive Operations in VDL Editing Phase	48
5.4.1	The Productions of the Mapping Algorithm	73

CHAPTER 1

INTRODUCTION

1.1 Why Visual Programming

The improtance for the study of visual language has been recognized in the United States and Japan in recent years [3,14,19,24,25,30,31]. The major issues of the study are to provide a programming environment for Non-Programmer Professionals to develop their own application programs visually. This kind of applications can be seen widely in the design of Office Information Systems [7,16,17,23,28] and others such as Computer Aided Instructions [1,15,21]. In these applications, the visual language is used to describe visional data: a menu, a screen layout, an engineering drawing, a typeset report, and the abstract data type such as hierarchy, condition statements, and rule-based knowledge.

When encountered in the design of such an application program, one may ask whether the traditional programming environment or the AI programming environment should be used? As we know, both programming environments require knowledge about the programming techniques which are still the time-consuming, detail-intensive, and error-prone chores. In addition, as the computer has brought into the office environment and the classroom, there emerges a group of

non-programmers who have no time or no interest to program but urgently need to write their own application programs from time to time. Also, according to the survey of Rockard and Flannery [22], when classified by primary purpose of end-user applications, about 50-56% of the end-user applications in table 1.1.1 are not of the pre-designed categories. Users fall into these categories can not derive the benefits of computerization without the help of **programming**. This leads to the problem of the so called "application program backlog".

Table 1.1.1 Categories about the user applications

Category	Percentage
Operational Processing	9%
Report Generation	14%
Inquiry/Simple Analysis	21%
Complex Analytical Assistance	50%
Miscellaneous	6%

To solve the "application program backlog" problem, a new environment called the visual programming environment, which is different from the traditional and AI programming environments, is proposed to let the users with or without programming knowledge to develop and run their own application programs. Under this environment, there is a V-Form Definition Language (VDL) which allows the users to focus on expressing their specialized knowledge without relying on the data structure or information processing techniques and yet can effectively carry out their tasks. Usually, these tasks under the visual programming environment can be processed in terms of V-Forms' definition and their

relations which can be interpreted by any AI language. Hence, an V-Prolog interpreter is proposed as the interpreter of the "programs" synthesized by the V-Form Definition Language. In this way, rules and data processing capabilities can be handled well in the visual programming environment.

1.2 Organization Of This Report

This report is organized as follows:

Chapter 2 discusses the properties of a visual programming environment and its differences to the traditional and AI programming environments. Some related researches concerning form language and user-friendly interfaces are discussed. Finally, derived from the concept of visual programming environment, a suitable system architecture is proposed.

Chapter 3 gives the formal definition of visual language and the properties of visual forms (V-Forms). Also, a V-Form model which explicitly reveals the visual properties of V-Forms to the user and implicitly supports the internal structure of V-Forms to the system is proposed.

Chapter 4 discusses the syntax of a V-Form Definition Language which allows the users to describe their applications visually. It will be shown that the V-Form Definition Language is a three-dimensional, non-procedural language for defining and manipulating V-Forms on the screen in a manner of what-you-sketch-is-what-you-get. Also, a prototype

graphics utility is designed to provide the graphical information of V-Forms in the sketch. Finally, an CAI example is given to illustrate how VDL is functioning.

Chapter 5 presents a method that interprets V-Forms under a logic programming language -- Prolog. After processing VDL, a consistent internal structure that holds the definition and relations of V-Forms for each application can be self-synthesized and the behavior of the internal structure can be interpreted. In consideration of portability, extensibility, and sometimes in need of unification and backtracking during the interpretation of such an internal structure, a CProlog interpreter (available on VAX 11/785 under VAX/VMS at NTU Computer Center) with a slight modification is employed as the target interpreter. Finally, a mapping algorithm which transforms the internal structure into a Prolog program is discussed.

CHARTER 2

VISUAL PROGRAMMING ENVIRONMENT

2.1 Introduction

In the following, we shall divide our discussion in visual programming environment into three sections. The first section provides the discussion of the related researches in the aspect of visual interface design and their effect on our approach.

The second section gives a detailed explanation about a proposed visual programming environment and its difference to traditional and AI programming environments.

The third section introduces a basic three-level architecture of V-Form Definition Language, from which an overall system architecture is designed. Finally, the advantages and features of this approach are discussed.

2.2 Related Researches

Recently, in the United States and Japan, there exists a number of user interface systems which provide both programmers and non-programmers

a guiding to describe their applications. Among them, Form Language [27] by Sugihara, QBE/OBE [30,31] by Zloof, FORMAL [24,25] by Shu, and the State Transition Diagram Language [14] by Jacob are well known. In the following, we shall summarize their primary features without going into the detailed designs and implementations.

2.2.1 Form Language

The approach to the design of a form language for office use was proposed by Sugihara, et al. Their research was basically to design a form language which allows users to define and manipulate office forms visually. This language consists of two components: a form definition language and a form manipulation language to define and manipulate forms, respectively.

Using the form definition language, users can describe the forms processed in the office environment. On the other hand, users can use form manipulation language to create, retrieve, modify, and browse form instances (filling values into forms). With this form language, the forms and activities in the office can be simulated.

The most important features of the form language that are employed in this report includes:

1. Sugihara proposes a new form model to exploit the visual properties of forms. A form, in their model, is defined as a pair of a form type and a form instance. Once a form type is defined, different values can be

filled to obtain different kinds of form instances. As a matter of fact, V-Form model proposed in this report as discussed in charter 3 is basically the extension of their form model.

2. The screen is partitioned into several windows by the user. Each window represents a subform (forms within a form, just like columns or fields within a table) to be displayed. Users can arrange the windows so that an office form can be faithfully reproduced. In many visual applications, this is considered as a good approach toward user interface design. The V-Form proposed in this report is basically the same as office forms, except that V-Form includes more entities.

3. A form has a form heading, a heading has many columns, and a column has several fields, so that the relation of forms is a hierarchical structure. We shall also use the hierarchical structure but add in the control flow to V-Forms.

In summary, the form language proposed by Sugihara is well suited to office environment, as they put their conclusion in the following sentence:

> "This is the step toward the development of user-friendly interfaces of office information system."

2.2.2 QBE/OBE

QBE (Query By Example) is an IBM product released in 1978. It is centered on office and business applications and is widely used in the areas such as distribution, finance, government, manifacturing, processing, and utilities. On the other hand, OBE (Office procedure By Example) is the extended version of the QBE.

The features of QBE/OBE are:

1. QBE/OBE emphasizes on relational DBMS interactive query and data mainteinance. The operations are easy to learn and easy to use. Users only need to describe their applications directly to the computer.

2. The fundamental object of QBE is a two-dimensional, one-level skeleton table which also provides the programming environment. Initially, users are given a blank table skeleton. Then, after key in the appropriate name into the table name field, the table heading is generated and the users can now "program" it by entering appropriate QBE/OBE commands.

QBE/OBE is well suited to office automation as stated by Zloof [31]:

"We suggest that the two-dimensional programming approach of Query-by-Example ... is suitable for non-programmers who wish to interactively automate their applications ... Consequently, users with no knowledge of

> any formal programming language can, in a matter of several hours training, formulate QBE programs to retrieve, modify, define, and control the database. Psychological testing of QBE users has shown it to be a very friendly language."

2.2.3 FORMAL (Form-ORiented MAnipulation Language)

FORMAL provides a powerful visual-directed facility for non-programmers to develop their data manipulation applications on computers. To accomplish their goal, a forms-oriented approach is employed. Their approach consists of three aspects: a form data model, a form-oriented language, and filling forms with instances.

The form data model proposed by Shu defines the forms as a named collection of instances (or records) with the same data structure. The components of a form can be any combination of fields and groups. Fields is the smallest unit of data that can be referenced in a user's application, while group is a sequence of one or more fields and/or subordinate groups. The group is also called the subform of a form. This form-subform concept is also employed in Sugihara's approach.

The form-oriented programming language is a two-dimensional, non-procedural description language which uses forms as both the fundamental data object and program structure. Users can program within the form visually. The concept of what-you-sketch-is-what-you-get is also introduced.

In summary, as said in the paper of Shu:

> "FORMAL is a forms-oriented and visual-directed application language, designed and implemented to provide the non-programmers with powerful capabilities to computerize a wide range of data processing tasks."

2.2.4 State Transition Diagram Language

The state transition diagram language (STDL) uses diagram to describe algorithms to the computer. A visual programming environment for this language is currently being implemented on a SUN workstation [14]. The main features of STDL are:

1. Uses graphical representations to represent data objects as well as abstract objects.

2. The temporal sequence during the dialog between a user and the system are emphasized. This concept is important in many application areas, such as computer aided instruction. Therefore, the basic concept of temporal sequence during user/system dialog will be used in our approach.

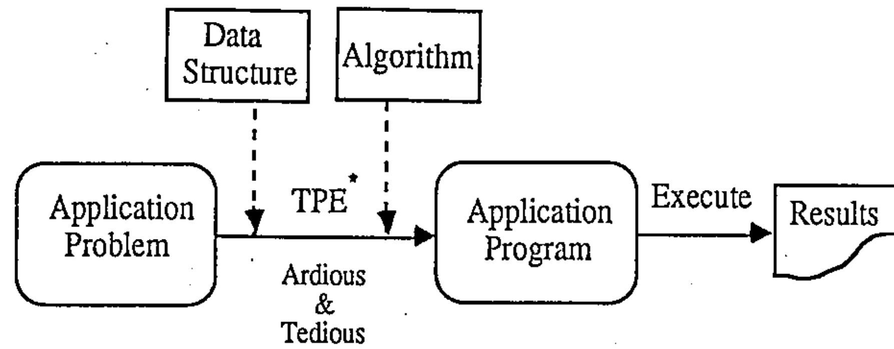
3. One diagram can call upon another diagram. This is one of the important features of this language. In this way, procedure call in traditional programming language can be simulated. In our approach, this gives the idea of procedure embedded concept within a form.

2.3 The Visual Programming Environment

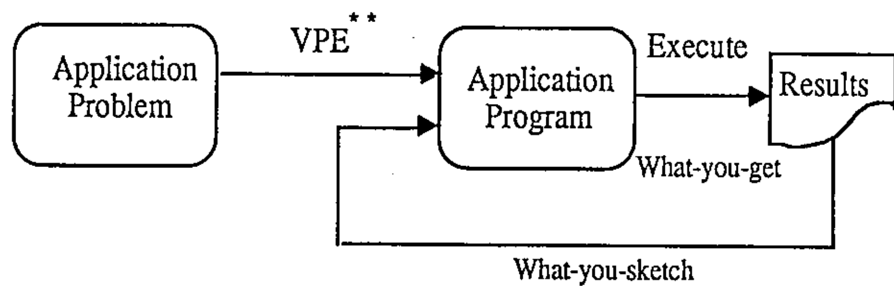
Traditional programming environment concerns how to provide some kinds of tools for the users, programmers, to develop their application programs. In this environment, users must take care of the data structure, keep in mind what the results should be, and write the detailed textual instructions that must adhere strictly to the syntax rules. Under this situation, the background and convenience of non-programmers are often ignored for the sake of machine efficiency. On the other hand, under AI programming environment alone, it seems still very difficult for Non-Programmer Professionals (NPPs) to express their specialized knowledge very well. Hence, a new programming environment is needed such that users with or without any programming knowledge background can describe their specialized knowledge to the computer. This newly emerging technique is called the **visual programming environment.**

The visual programming environment allows users to develop their applications in a manner of what-they-sketch-is-what-they-get. In this way, users only need to describe their specialized knowledge as a process of how to get the desired output results. The difference of this visual process and the traditional programming process is shown in fig. 2.3.1.

Notice that, during the development of an application program, the user can see the results echoed on the screen and determine to see if these results are what he desired. This visual process is just like painting on a scratchpad which is so easy that a user is no longer afraid of writing his/her own application programs.



(a)



(b)

Fig. 2.3.1 Visual Process vs. Traditional Programming Process

* TPE: Traditional Programming Environment
** VPE: Visual Programming Environment

2.4 Overview Of The System Configuration

2.4.1 System Architecture

The basic architecture of Automatic Program-Synthesizing (APS) system consists of three hierarchical levels similar to that of ANSI/X3/SPARC Information system framework and SPECDOQ [17] as

shown in fig. 2.4.1. They are called the external, logical, and internal level. External level, the highest, handles V-Forms from the user's point of view. The V-Forms defined by the user are synthesized by VDL into logical level which is the internal structure of V-Forms. Finally, a mapping is invoked to translate the self-synthesized internal structure into internal level which is in the form of predicate logic [18] and can be directly interpreted by a Prolog interpreter [2,4,6,20,26].

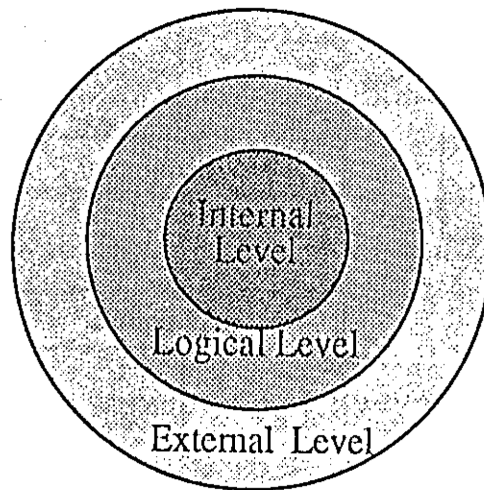


Fig. 2.4.1 Basic Architecture of APS System

The corresponding components of our approach with respect to the basic architecture are shown in fig. 2.4.2. The informant presentation data to be processed may include text, graphics, and rules. They are formated in V-Forms. Based on a V-Form model, a V-Form Definition Language is used as an interface language to define the V-Forms. After processing VDL, a complete internal structure can be self-synthesized. Finally, a mapping algorithm is used to translate this internal structure into an executable Prolog program. During the interpretation of the Prolog program, V-Forms with text or rule type are directly interpreted by

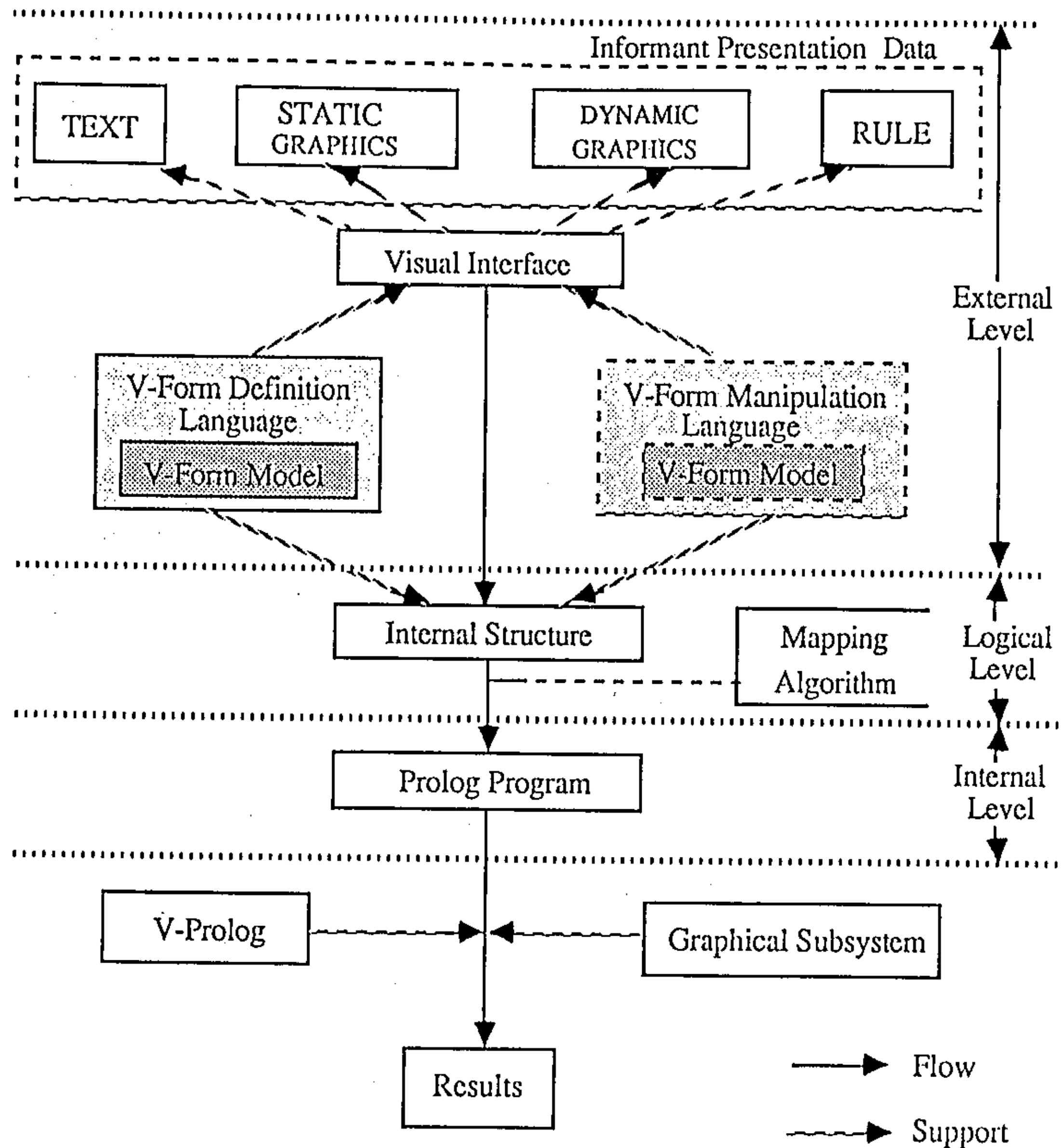


Fig. 2.4.2 System Architecture

V-Prolog, while V-Forms with graphic type are delivered to graphical subsystem for execution. In this way, the user's application programs can be developed visually. Furthermore, a V-Form Manipulation Language can be invoked to manipulate the designed applications. The overall system is called VIsual Programming Synthesizer (VIPS) as described

elsewhere [5]. For more information about V-Form Manipulation Language, please consult the part 2 of this report.

2.4.2 Main Features of Visual Programming Approach

There are some design methods for visual programming languages [14,19,24,25,27]. However, a systematic approach toward a theoretical sound methodology is still under developing. Here, the visual programming environment is proposed to be interactive and application-oriented with the following features:

1. **Easy to Use.** At the current stage, the communication between user and system is through prompting a command menu on the screen. However, facilities are reserved so that various user friendly interfaces such as icons, pointing devices, or mouses can be easily appended if the hardware is available. For novices, it takes only a few hours' training to be familiar with our system.

2. **Visual-Directed Objects.** V-Forms are used as the fundamental objects of our approach because their features are more akin to the users' view point. From the visual interface, users can define their application objects on the screen in a manner of what-they-sketch is what-they-get.

3. **Non-Procedural Programming Nature.** Instead of writing a series of instruction code to tell **how**, users

need only to tell the computer **what** the results should be. This will make the programming an easy and artistic job.

4. **For Non-Programmers.** The application programs are automatically generated by VDL, so the users can concentrate on expressing their specialized knowledge to the computer.

5. **Variety of V-Forms.** Multiple sources can be described in the V-Forms, e.g. text, graphics, or rules. Voice can also be included if needed. Furthermore, procedures or actions can be embedded within a V-Form. This makes the V-Form an active media which is powerful to describe many user applications.

6. **Portable.** The source of this system is coded in standard Pascal with the machine-independent factor loaded from a file. Porting it from one environment to another is simply by changing the parameters of the file. Furthermore, the internal structure at the logical level is uniform and consistent. Hence, to transport the proposed system from one computer to another is simply by recompiling the sources, changing the parameter file, and a slight modifying the mapping algorithm.

7. **Extensible.** The interpreter is contructed on a CProlog interpreter available on VAX/VMS as well as some added extensions (we call this interpreter as the

V-Prolog). Some reservations are kept so that the system maintainer can easily add the available tools to extend the capability of the proposed system without changing or modifying the source programs.

CHAPTER 3

ON VISUAL LANGUAGE

3.1 Introduction

The study of visual language approach has just begun in the United States and Japan in recent years. However, a formal definition on visual language is still underdeveloped. In this chapter, we try to define visual language from theoretical viewpoint of formal language.

Visual forms (V-Forms) which are the basic element of data/programming objects can be regared as the terminal nodes of a grammar that generates visual languages. A V-Form model which forms the basis of the visual language can be regard as a parsing tree.

3.2 Background

Formal language theory [11] has been developed extensively, and has several discernible trends which include applications to syntatic analysis of programming language, program schemes, and relationships with natural language. In order to define the visual language approach formally, we will give a brief introduction about context-free language,

context-sensitive language, regular language, context-free programmed language, inferred grammar, Kleene closure, and non-procedural language as follows.

3.2.1 Closures

Let Σ be a finite set of symbols, λ be a null string, and let L_1, L_2 be a set of strings from the reflexive and transitive closure [12] of Σ. The concatenation of L_1 and L_2, denoted by L_1L_2, is the set $\{xy \mid x \in L_1, y \in L_2\}$. That is, the strings in L_1L_2 are formed by choosing a string in L_1 and following it by a string in L_2, in all possible combinations.

[Definition 3.2.1] **Kleene Closure** [8]

Let $L^0 = \{\lambda\}$ and $L^i = LL^{i-1}$, $\forall\, i \geq 1$. The Kleene closure of L, denoted by L^*, is the set

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

[Definition 3.2.2] **Positive Closure**

The positive closure of L, denoted by L^+, is the set

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

In other words, L^* denotes the words constructed by concatenating any

number of words from L, while L^+ is the same but with the word λ excluded. Note that, L^+ contains λ if and only if L does.

3.2.2 Phrase-Structure Grammars/Languages

[Definition 3.2.3] **Phrase-Structure Grammar** [9]

A phrase-structure grammar G is a four-tuple $G = (V_N, V_T, P, S)$ in which:

1. V_N and V_T are sets of the nonterminal and terminal vocabularies of G, respectively.

2. P is a finite set of rewrite rules or productions denoted by $\alpha \rightarrow \beta$, where $\alpha, \beta \in V$ $(= V_T \cup V_N)$ with α involving at least one symbol of V_N.

3. $S \in V_N$ is the starting symbol of a sentence.

Chomsky divided the phrase-structure grammars into four types according to the forms of the productions: type 0 is unrestricted grammar, type 1 is context-sensitive grammar, type 2 is context-free grammar, and type 3 is regular grammar.

[Definition 3.2.4] **Type 0 (Unrestricted) Grammar**

For type 0 grammar, there is no restrictions on the productions, i.e. either α or β may have any strings. This type of grammar is too general to be useful.

[Definition 3.2.5] **Type 1 (Context-Sensitive) Grammar**

For context-sensitive grammar, the productions are restricted to the form:

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$$

where $A \in V_N$, $\alpha_1, \alpha_2 \in V^*$, $\beta \in V^+$. The languages generated by context-sensitive grammars are called the context-sensitive language.

[Definition 3.2.6] **Type 2 (Context-Free) Grammar**

For context-free grammar, the productions are of the form:

$$A \rightarrow \beta$$

where $A \in V_N$, $\beta \in V_T$. Notice that, context-free grammar allows the nonterminal A to be replaced by the string β independently of the context in which A appears. On the other hand, productions of context-sensitive grammar permit replacement of nonterminal A by the string β only in the "context" α_1-α_2. The languages generated by context-free grammars are called the context-free language.

[Definition 3.2.7] **Type 3 (Regular) Grammar**

For regular grammar, the productions are restricted to the form:

$$A \rightarrow aB \quad \text{or} \quad A \rightarrow b$$

where A, B $\in V_N$ and a, b $\in V_T$. Note that all A, B, a, and b are single symbols. The languages generated by regular grammars are called regular language.

3.2.3 Inferred Grammar

The problem of learning a grammar based on a set of sample sentences is called the grammatical inferences. Let G be an unknown grammar, S_t be a finite set of sentences or strings, L(G) be the language grnerated by G, then the inferred grammar is a set rules for describing the given finite set of strings S_t from L(G) and predicting other strings which in some sense are of the same as the given set [9].

However, the predicted strings from S_t may be in the set $\{+y_i \mid i = 1, 2, ..., t, y_i \in L\} \cup \{-y_j \mid j = 1, 2, ..., t, y_j \in V_T^* - L\}$, where $+y_i$ contains only strings from L and $-y_j$ contains only strings from V_T^* - L. The set $\{+y_i \mid i = 1, 2, ..., t, y_i \in L\}$ is called the positive sample of L(G) and is denoted as $L^+(G)$, while the set $\{-y_j \mid j = 1, 2, ..., t, y_j \in V_T^* - L\}$ is called the negative sample of L(G) and is denoted as $L^-(G)$ which is the parasitical product of G.

3.2.4 Context-Free Programmed Language

[Definition 3.2.8] **Context-Free Programmed Grammar** [9]

A context-free programmed grammar G is a five-tuple G = (V_N, V_T, J, P, S), where V_N, V_T, and P are finite sets of nonterminals, terminals, and productions, respectively. S is the starting symbol, $S \in V_N$. J is a set of production labels. The productions of G are of the form:

$$(r) \quad A \rightarrow \beta \quad S(U) \quad F(W)$$

where $A \rightarrow \beta$ is called the core, $A \in V_N$, $\beta \in V^*$, (r) is the label, $r \in J$. U is the success field and W the failure field. U, $W \in J$.

The context-free programmed grammar operates as follows:

Step 1. Production (1) is applied first.
Step 2. If one tries to apply production (r), $r \in J$ to rewrite $A \in V_N$ and the current string α contains A,
then
(γ) $A \rightarrow \beta$ is applied and the next production selected from the success go-to field U.
else
production (r) is not used and the next production is selected form the failure go-to field W.
Step 3. if the applicable go-to field contains ∅ then derivation halts.
else goto Step 2.

For instance, the context-free programmed grammar $G = (V_N, V_T, J, P, S)$ with $V_N=\{S, B, C\}$, $V_T=\{a, b, c\}$, $J = \{1, 2, 3, 4, 5\}$, and P :

(r)	Core	S(U)	F(W)
1	$S \rightarrow aBC$	{2, 3}	{∅}
2	$B \rightarrow aBB$	{4}	{∅}
3	$B \rightarrow b$	{3}	{5}
4	$C \rightarrow CC$	{2, 3}	{∅}
5	$C \rightarrow c$	{5}	{∅}

will generate the context-free programmed language $\{a^n b^n c^n \mid n = 1, 2, ...\}$

3.2.5 Non-Procedure Language

For convenience in our discussion, we say that a programming language is non-procedural if and only if the programming method is a process of specifying **what-to-get** instead of telling how-to-get the desired results.

Obviously, the process of specifying **what** is much more easier than that of telling **how**. Thus, a non-procedural language is essential for NPP users.

3.3 THE DEFINITION OF VISUAL LANGUAGE

3.3.1 Related Researches

The definition of a visual language drawn researchers' attension began at the first workshop on visual language sponsored by the IEEE Computer Society at Hiroshima, Japan on Dec. 1984. Since then, intensive discussion on this topic has been seen in many published papers in several journals. [3,5,14,19,24,25,27]. In the following, some of their definitions about the visual language are illustrated in words:

1. In "Call For Paper" of the IEEE computer Society Workshop on Visual Language:

   "Visual Language:

   1) A computer language with prominent visual components, such as icons or computer graphics.
   2) A computer language specifically designed for use with visual problem, such as image analysis."

2. In the paper "A State Transition Diagram Language for Visual Programming" of Jacob [14]:

   "The visual programming language provides a natural way to describe the graphical object."

Since the above definitions are rather abstract, we will give a formal definition about the visual language in the following section.

3.3.2 Formal Definition of Visual Language

[Definition 3.3.1] **Visual Grammar**

A visual grammar, G_V, is a four-tuple $G_V = (V_N, V_T, P, S)$ in which:

1. V_N and V_T are the nonterminal and terminal vocabularies of G_V, respectively. The union of V_N and V_T constitutes the total vocabulary V of G_V, $V_N \cap V_T = \emptyset$.

2. P is a set of inference rules (thinking processes) for describing the objects from the one sketched by the user and predicting the results which in some sense are of the same nature as the sketched objects.

3. $S \in V_N$ is the starting symbol.

The language generated by visual grammar G_V is:

$$L(G_V) = \{v \mid v \in V_T^*, \text{ such that } S \overset{*}{\underset{G_V}{\Rightarrow}} v\}$$

where $\overset{*}{\Rightarrow}$ is the reflexive and transitive closure of the relation $\Rightarrow$. As

described in section 3.2, the information sequence generated by G_V may contain a set of positive samples and a set of negative samples, i.e.

$$
\begin{aligned}
L(G_V) &= L^+(G_V) \cup L^-(G_V) \\
&= \{\text{positive samples}\} \cup \{\text{negative samples}\}
\end{aligned}
$$

In visual language approach, what we need is the objects sketched by the users. Hence, a visual language should be:

[Definition 3.3.2] **Visual Language**

Let D be the domain of what-you-sketch, and y be a sketched object, $y \in D$. Suppose y can be generated by the visual grammar G_V, then a visual language L_V is

$$L_V = \{y \mid y \in D, \text{ and } y \in L^+(G_V)\}$$

In other words, a visual language is a formal language that contains only objects sketched by the user.

As we know, there are three types of language, i.e., regular, context-free, and context-sensitive that can be generated by an inferred grammar in the limit [9]. Among them, context-free languages are not powerful enough to describe the programming applications, while context-sensitive languages are very complex for analysis. Therefore, the visual language L_V, which is used to get the desired applications in a set of what-they-sketch-is-what-they-get operations, is used to implement the

context-sensitive programming applications in a manner of context-free programmed language that can simulate the behavior of the visual grammar G_V. In other word, the visual programming itself is an interactive language which allows the users to describe the inference algorithm of G_V by themselves.

3.4 Why A V-Form Approach

To accomplish our goal, visual-forms (V-Forms) are adopted as the fundamental objects of our visual language approach. Here, the forms dealt with contain text, static graphics (line drawing and bit-map) , dynamic graphics (animation) , and rules. They are the basic elements in the visual programming applications. In order to distinguish them from those in the office environment, we call them the V-Forms.

There are several reasons for adopting V-Forms as the fundamental data/programming objects of our visual language approach. First, V-Forms are more akin to the user's viewpoint. As we know, people are more familiar with forms than any other objects. For example, filling a form is more easier than writing an article. Hence, it will be easy for users to use the V-Form-oriented system.

Second, forms are the most natural interface between a user and the data [18]. In the last few years, there exists a vast amount of research focusing on "forms" in the interests of office environment. For instance, Ellis [7] uses forms as the template for document which are logical images

of business paper forms. Kitagawa [17] presents an architecture and implementation of a form document management system which allows users to perform handling of form documents such as their creation, storage, retrieval, editing, cut, and paste, as if they were conventional paper documents in the office information systems. Shu [24, 25] has provided the users a forms-oriented programming language to describe their data processing activities as a form process or a series of form processes. Tsichritzis [28] has introduced forms as an abstract and generalization of Business paper forms. Furthermore, Zloof [30,34] has extended the query language and database system to deal with the forms. It seems that forms are the most natural interface between the non-programmers and the computer system.

Third, V-Forms are the most acceptable programming nature for non-programmers. In traditional programming environments, the programming objects manipulated by them are geared toward the internal/computer representations, users can only figure out in their mind what the results will be. On the other hand, in visual programming applications, it needs only the program-developing process in what-they-sketch-is-what-they-get manner of operations. To make this process feasible for non-programmers, we use V-Forms as the programming objects such that the results are directly corresponding to those described by the user. Thus, the programming nature is shifted from **procedural** to **non-procedural.**

Fourth, upon filling the contents of V-Forms, there may be some embedded actions/procedures to be taken. The actions/procedures, either conditional or unconditional, can be attached to the V-Forms. This

concept is similar to the **demons** [29] in the frame system. Hence, V-Forms can serve as a more complex, flexible knowledge representation method for future extension.

From the above four reasons, it is believed that the contents of V-Forms are more fruitful than the forms in the office environment. Hence, we use V-Forms-oriented approach as the convenient data/programming objects.

3.5 V-FORM MODEL

3.5.1 External Structure

VDL allows users to open several windows on a screen. Each window is treated as a V-Form. The informant presentation data in a V-Form may includes text, static graphics (line-drawing or bit-map), dynamic graphics (animation), or rules. They are all represented in V-Forms.

Let Π be a set of codes to be displayed/processed, Δ be a set of alphanumerics, β be a set of graphic codes, Ω be a set of drawing attributes, and $\Sigma = \Pi \cup \Delta \cup \beta \cup \Omega$. Then the domain of a V-Form y, $DOM(y) \in \Sigma^*$, where * denotes the Kleene closure [12]. All V-Forms of an application are subset of D and are considered as the positive sample of $L(G_V)$.

A V-Form system is a pair of a V-Form type F and a V-Form instance I which are defined below. A V-Form type describes the skeleton structure of a V-Form system which contains a list of structurally related V-Forms. A V-Form type consists of a scheme S and a template T_S for S.

[Definition 3.5.1] Scheme

A scheme S is the logical structure of a V-Form. It can be recursively defined as:

<S> ::= <M> | <A>
<A> ::= <S> | <S> , <A>
<M> ::= <Type> : <Identifier>
<Type> ::= TEXT | GRAPG | BIT-MAP | ANIMATION
| RULE | VFORM

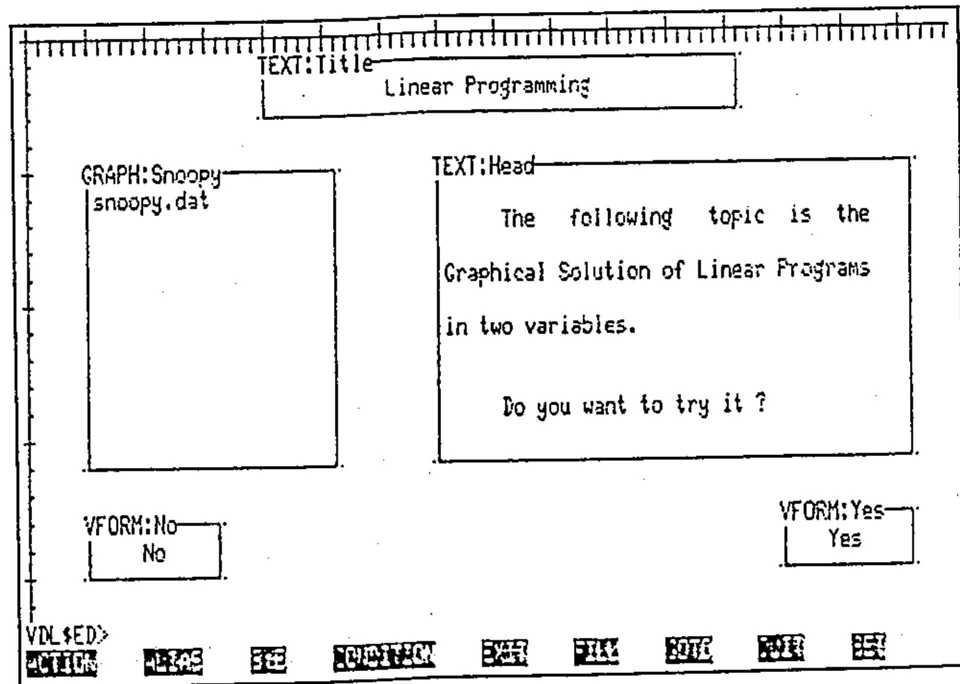


Fig. 3.5.1 The first V-Form F1 of an application CAI_Course

For example, consider the first V-From of an application CAI_Course shown in fig. 3.5.1. The sheme for this V-Form is defined in the following expressions:

CAI_Course = [TEXT:Title, GRAPH:Snoopy, TEXT:Head,
VFORM:Yes, VFORM:No]

Where TEXT:Title, GRAPH:Snoopy, and TEXT:Head are V-Forms that contain no other V-Forms (these primitive V-Forms are also called the **atoms**), while VFORM:Yes and VFORM:No are V-Forms that may contain a set of V-Forms.

The VForms-within-VForm concept is similar to that of the fields-within-record in a table structure. Hence, a hierarchical structure is formed. The V-Forms within a V-Form are also called the **sub-VForms** of that V-Form. Fig. 3.5.2 shows the hierarchical structure of the scheme for the application CAI_Course, where a box represents an atom, a circle represent a V-Form that contains other V-Forms.

[Definition 3.5.2] **Template**

A template T_S for the scheme S is the visual structure that represents the two-dimensional display format and visual properties of V-Forms which are independent of the contents held by V-Forms.

For instance, the template for the scheme S of fig. 3.5.1 is given in fig. 3.5.3.

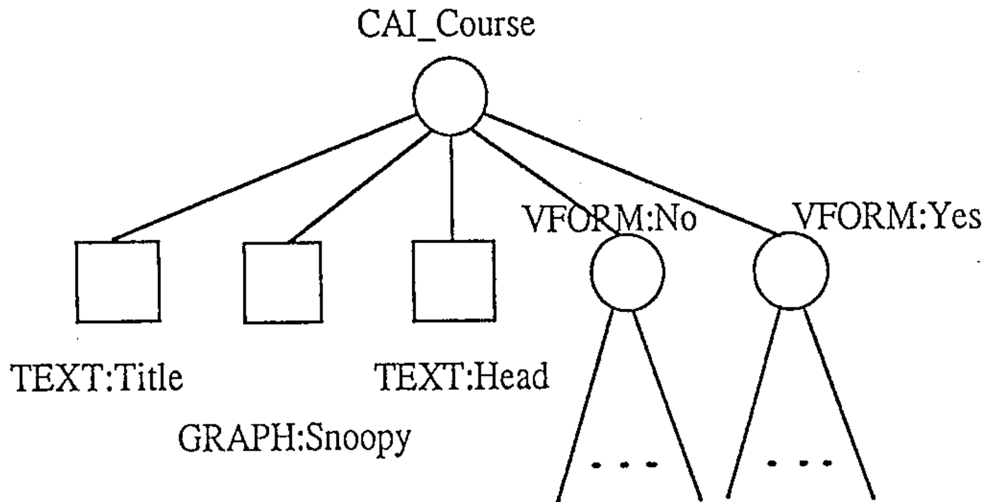


Fig. 3.5.2 The hierarchical structure for the application CAI_Course

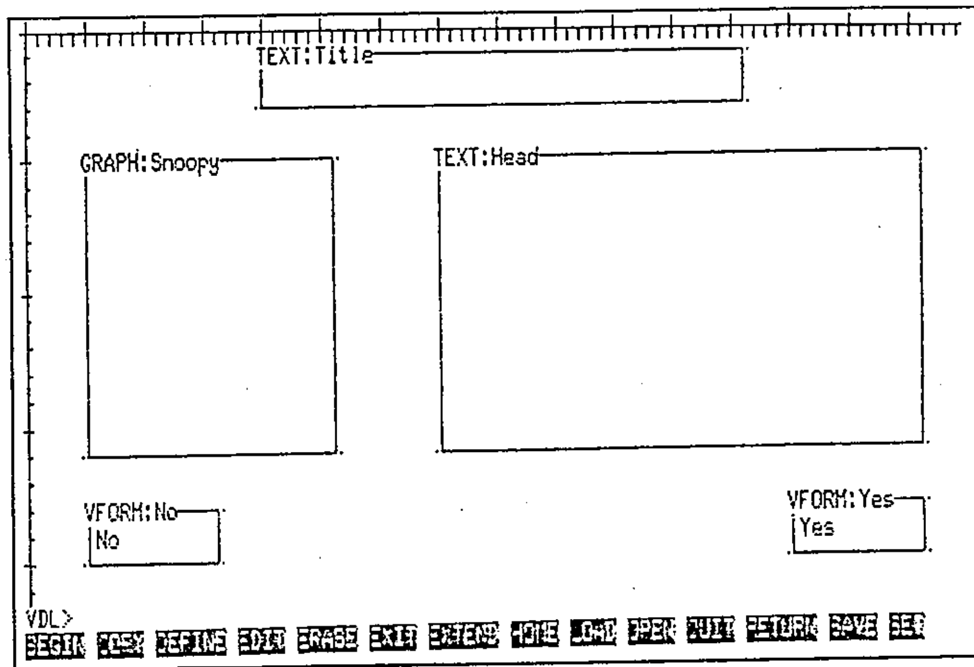


Fig. 3.5.3 The Template for the CAI_Course
(only the first V-Form of that application is shown)

The template also describes the control flow of the V-Form system implicitly. The control flow of a V-Form system is similar to the concept of flow-charting in traditional programming environment and

is the execution flow of user's application program. For example, when VForm:Yes (sub-VForm of CAI_Course) of fig. 3.5.1 is selected, then the display will be switched to that of its descendent V-Forms.

[Definition 3.5.3] The **Value** of a V-Form

The value of a V-Form y of type t is x, such that $x \in DOM(y_t) \subseteq \Sigma^*$, where

$$DOM(y_t) = \begin{cases} (\Delta \cup \Pi)^* , \text{if } t = \text{TEXT, RULE} \\ (\beta \cup \Omega)^* \text{ if } t = \text{GRAPH, BIT-MAP, ANIMATION} \end{cases}$$

For example, the value of V-Form Title of type TEXT in fig. 3.5.1 is "Linear Programming".

[Definition 3.5.4] The **Characteristics** of a V-Form

The characteristics of a V-Form include the displaying attributes (NORMAL, BOLD, FLASH, or INVERSE vedio) , the alias (give an alternative name to a given V-Form) , and the "demons" (embedded action/procedure) of a V-Form.

[Definition 3.5.5] **V-Form Instance** (I)

The V-Form instance for a V-Form type (F) is defined as a mapping which assigns a value to each atom of F and assigns optionally a characteristic to each V-Form of F.

For instance, one of the V-Form instance for the template of fig. 3.5.3 is shown in fig. 3.5.1. As stated above, a V-Form type can be filled with different contents to obtain different kinds of V-Form instances. Therefore, a V-Form type is similar to a **language**, from which users can write many **programs** which are V-Form instances in this model.

3.5.2 Internal Structure

The external structure of this V-Form model gives the user an opportunity to express their specialized knowledge in terms of V-Forms, while the internal structure gives an internal/computer representation about the V-Forms defined by the user.

According to the external structure defined in section 3.5.1, a V-Form system consists of a group of V-Forms which may be either atoms or sub-VForms of a particular V-Form. Furthermore, "demons" can also be embedded in a V-Form. Hence, an internal structure with four kind of nodes is proposed to implement the atoms, V-Forms, and demons. They are called the system node (SN), the form control node (FCN), the mode node (MN), and the action node (AN). System node keeps all information of a V-Form system. Form control node keeps all the characteristics of a V-Form (either atom or non-atom). Mode node contains the value and display attribute of an atom. Alias and the content of the embedded procedure are also kept in MN. Finally, action node specifies the embedded components, such as conditions and actions.

An atom is represented by a FCN which has a pointer pointing to a MN. A non-atom V-Form is also represented by a FCN but has a pointer

(in FCN) pointing to a list of FCN which describes its sub-VForms. The list of FCN, also called a form control table (FCT), describes the V-Form structure within a window. For example, the internal structure of fig. 3.5.1 is shown in fig. 3.5.4

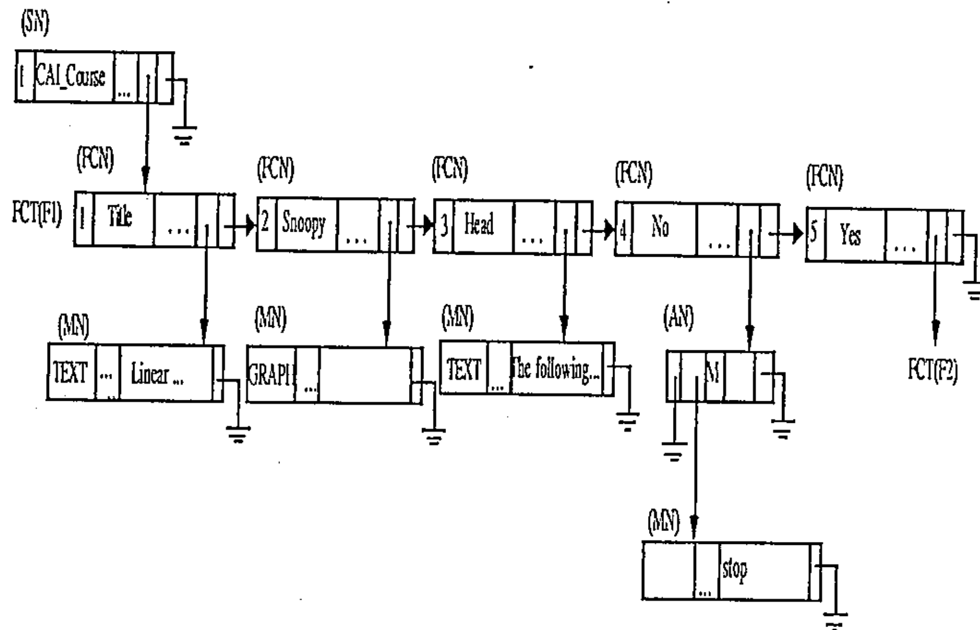


Fig. 3.5.4 The internal structure of the V-Form F1 of fig. 3.5.1

CHAPTER 4

V-FORM DEFINITION LANGUAGE

4.1 Introduction

V-Form Definition Language (VDL) is a pictorial language used to describe the logical and visual structure of V-Forms. It is a three-dimensional non-procedural language to communicate with users interactively. Once the user has prepared his/her own applications in a set of well-written sheets of V-Forms, he/she can invoke V-Form Definition Language to manage the screen at will.

As stated in Chapter 3, a V-Form consists of a V-Form type F and a V- Form instance I. Hence, VDL is divided into two phases, namely, a skeleton phase and an editing phase to define the V-Form type and V-Form instance, respectively. The details about this two phases are given in this chapter. An illustrative example on the CAI application is also given to illustrate how the V-Form Definition Language is performed.

It is worthwhile to mention that a prototype graphics utility is designed to serve as the graphical entry of non-text type V-Forms only. However, a commercial graphical packages is preferred and is suggested to be attached to our system, if there is one available.

4.2 The Skeleton Phase

In this phase, the user can directly manipulate the screen visually. The role of the skeleton phase is shown in fig. 4.2.1. The skeleton phase is used to describe the template of a V-Form system and after processing this phase, a V-Form type is obtained which is delivered to the editing phase for filling in the V-Form instances.

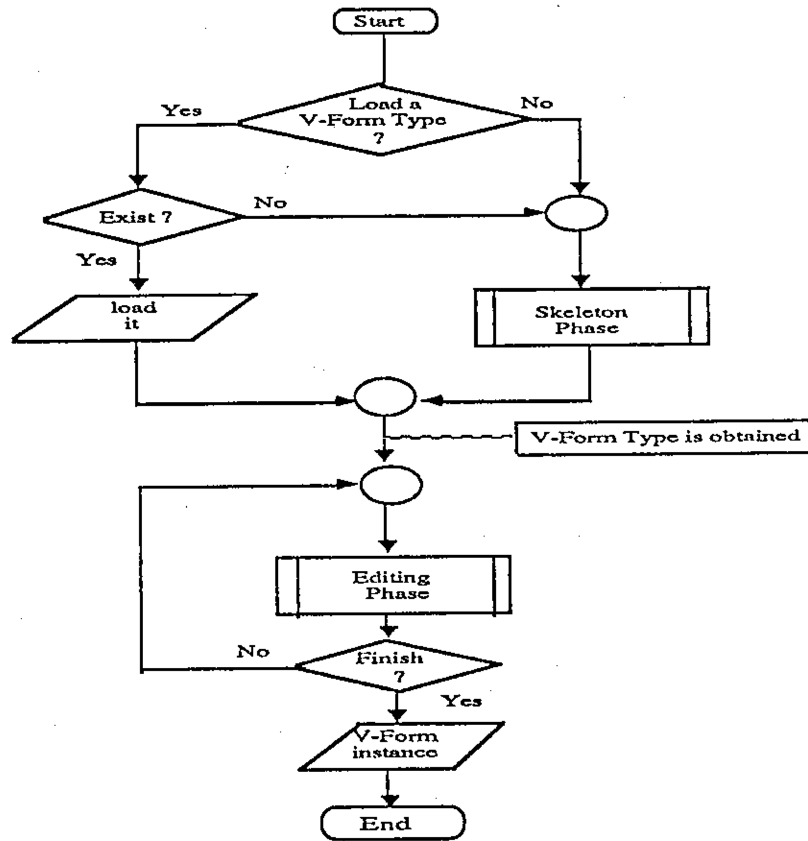


Fig. 4.2.1 Block Diagram of VDL Interactive Sequence

For convenience, all the following interactive sequences are illustrated in line-mode commands. But actually in our system, a mouse or a pointing device can be used as a menu-driven device.

4.2.1 BNF Syntax of the Skeleton Phase

There are fourteen primitive operations in the skeleton phase which are summarized in Table 4.2.1, where the bracket denotes optional.

Table 4.2.1 Syntax of the primitive operations in VDL skeleton phase

```
<VDL>          ::= <START> [<COMMANDS>] <END>
<START>        ::= <BEGIN> <SYSTEM-NAME>
<COMMANDS>     ::= <COMMAND> [<COMMANDS>]
<COMMAND>      ::= <COPY> | <DEFINE> | <EDIT> | <ERASE>
                 | <EXTEND> | <HOME> | <LOAD> | <OPEN>
                 | <RETURN> | <SAVE> | <SET>
<COPY>         ::= COPY [{PROCEDURE | FLOW} [OF]]
                   <VFORM_NAME> TO <VFORM_NAME>
<DEFINE>       ::= DEFINE <VFORM_NAME> AS <TYPES>
<TYPES>        ::= TEXT | GRAPH | BIT-MAP | ANIMATION | RULE
                 | VOICE | VFORM
<EDIT>         ::= EDIT <VFORM_NAME>
<ERASE>        ::= ERASE <VFORM_NAME>
<EXTEND>       ::= EXTEND <VFORM_NAME> [MARGIN AT
                   <POSITION> WITH <SIZE>]
<POSITION>     ::= <INTEGER> <INTEGER>
<SIZE>         ::= <INTEGER> <INTEGER>
<HOME>         ::= HOME [<SYSTEM_NAME>]
<LOAD>         ::= LOAD <FILE_NAME>
<OPEN>         ::= OPEN <TYPES> AT <POSITION> WITH <SIZE>
                   [AS <VFORM_NAME>]
<RETURN>       ::= RETURN
<SAVE>         ::= SAVE <FILE_NAME>
<SET>          ::= SET NEXT <VFORM_NAME> [GOTO]
                   <GROUP_NAME>
                 | SET PROC <VFORM_NAME> [TO]
                   <STATEMENTS>
                 | SET TOTAL <GROUP_NAME>
<END>          ::= EXIT | QUIT
```

In table 4.2.1, SYSTEM_NAME is the name of the V-Form system. VFORM_NAME is either the name of a V-Form given during the process of OPENing or a default name given by VDL. GROUP_NAME gives a method to name the whole screen which may

contain a group of V-Forms. The above three names are all identifiers. On the other hand, the FILE_NAME is the legal file specification allowed by the operating system of the user's computer environment.

4.2.2 Primitive Operations of the Skeleton Phase

According to the function performed, the above fourteen primitive operations can be further classified into seven categories, namely, initialization, window operation, flow manipulation, type redefining, instance generation, file handling, and miscellaneous.

4.2.2.1 Initialization

The command BEGIN is used to identify a V-Form system and start the "programming" of V-Forms. It causes VDL to get a system node, initialize variables for a new V-Form system, and assign the given name to the system node, i.e.

```
procedure VDL$Begin;
   begin
      get a system node;
      initialize the variables needed;
      SN.name ← name;
   end;
```

The name defined by BEGIN can also be used as the file name of the defined V-Form type and V-Form instance (will be described later).

4.2.2.2 Window Operation

There are two commands in this category, namely, OPEN and

ERASE. As the name stands, the command OPEN is used to open a window on the screen, while the command ERASE is used to erase a window. When the command OPEN is given , a user-supplied name and a system-given unique id are attached to the V-Form opened. The unique id is used as the key to the searching process which is needed during erasing or editing. Hence, even different V-Forms can be assigned the same name and still no confusion. The algorithms for OPEN and ERASE are as follows.

```
procedure VDL$Open;
   begin
      get a Form control node;
      if type ∈ {text, graph, bitmap, animation, voice, rule} then
         begin
            get a mode node;
         end;
      assign the characteristics supplied by the command to the relative
         fields of the nodes gotten;
   end;

procedure VDL$Erase;
   begin
      search a V-Form on the screen with the name the same
         as that required;
      if the V-Form is found then
         begin
            dispose the nodes (FCN and optional MN or AN);
         end
      else report the error;
   end;
```

For instance, after giving the following interactive commands, we have the V-Forms as shown in fig. 3.5.3.

```
VDL> BEGIN CAI Course
VDL> OPEN TEXT AT 1 20 WITH 40 1 AS Title
VDL> OPEN GRAPH AT 5 5 WITH 20 10 AS Snoopy
VDL> OPEN TEXT AT 5 35 WITH 40 10 AS Head
VDL> OPEN VFORM AT 18 5 WITH 10 1 AS No
VDL> OPEN VFORM AT 18 65 WITH 10 1 AS Yes
```

4.2.2.3 Flow Manipulation

There are three commands in this category, namely, EXTEND, RETURN, and SET. The EXTEND command can be used to stretch the V-Form node so that we can define the sub-VForms of a V-Form node. The stretching process is similar to the extension in the z-direction (the screen is considered as the x-y plane). This is the reason why we call the VDL a three-dimensional language. Furthermore, the screen margin can also be redefined by EXTEND command. This facility is useful in some multiwindow applications. For example, in CAI, part of the screen can be reserved for demonstration of theorem, the other part of the screen can be used as interactive region for teaching, just like a blackboard in the classroom. The algorithm for EXTEND is as follows.

```
procedure VDL$Extend;
 begin
    search a V-Form on the screen with the name the same as that
       required;
    if the V-Form is found then
       begin
          clear up the screen with the specified margin; {if not
             specified, the default is the one specified previously.
             Initially, a full screen is specified}
          push the V-Form to return on a stack;
       end
    else report the error;
 end;
```

For instance, the command

VDL> <u>EXTEND Yes</u>

will clear up the screen, ready for users to define the sub-VForms of VForm:Yes. After giving a series of OPEN commands similar to those in section 4.2.2.2, the screen of fig. 4.2.2 is obtained.

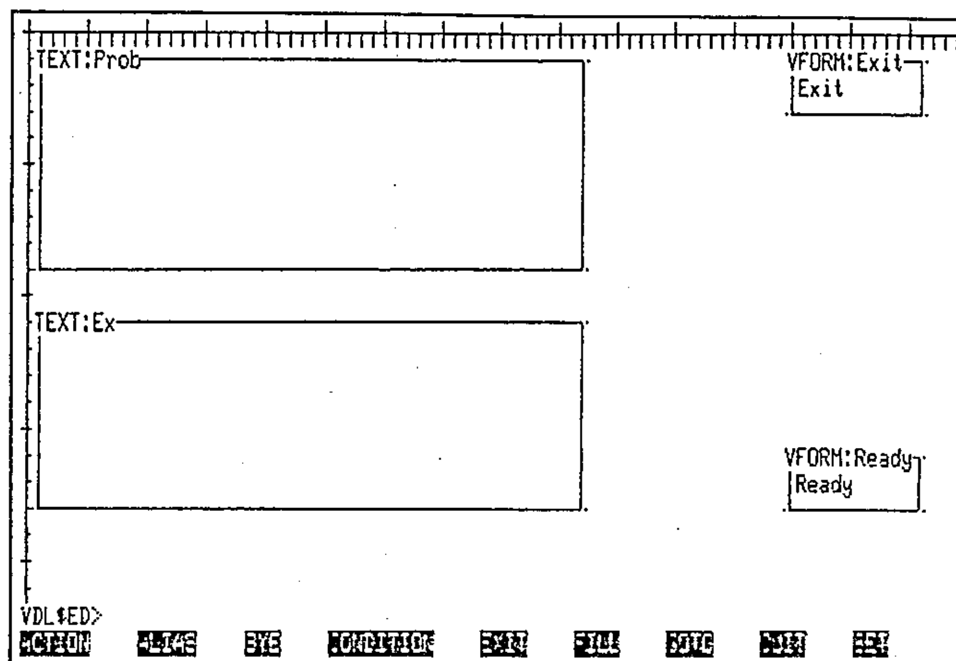


Fig. 4.2.2 After using EXTEND and a series of OPEN commands on VForm:Yes of fig. 3.5.3

The RETURN command can be used to return to the previous V-Form. The algorithm for RETURN is as follows.

```
procedure VDL$Return;
   begin
      pop the stack;
      display the screen which contains the popped V-Form;
      reset the screen margin to the screen newly displayed;
   end;
```

For instance, the selection of

VDL> <u>RETURN</u>

in fig. 4.2.2 will display fig. 3.5.3 again.

The SET command has three options: SET NEXT, SET PROC, and SET TOTAL to give the next destination, assign the embedded procedure of a V-Form, and group the V-Forms on a screen, respectively. The algorithm for SET is as follows.

```
procedure VDL$Set;
  begin
    case options of
      'SET NEXT':
        begin
          assign the destination V-Form's address, from the
            group table, to the field PhyAddr;
        end;
      'SET PROC':
        begin
          get an action node;
          get a mode node;
          store the procedure to the MN;
          assign the address of MN to the field ActPtr of AN;
        end;
      'SET TOTAL':
        begin
          save the address of the first entry of the screen and the
            group name into a group table;
        end;
    end; {case}
  end;
```

For instance, you can group the screen of fig. 3.5.3 as the name F1 by giving the command:

VDL> SET TOTAL F1

after the interactive command shown in section 4.2.2.2.

4.2.2.4 Type Redefining

This category of operations contains only one command: DEFINE. It can be used to change the type of a V-Form. The algorithm for DEFINE is as follows.

```
procedure VDL$Define;
   begin
      case old_type of
         'TEXT',    'GRAPH',  'BIT-MAP',  'ANIMATION',
         'VOICE', 'RULE' :
            begin
               if new_type = 'VFORM' then
                  begin
                     dispose the attached MN;
                     set field MF of FCN to F;
                  end
               else
                  begin
                     set field type of MN to new_type;
                  end;
            end;
         'VFORM':
            begin
               if new_type ≠ 'VFORM' then
                  begin
                     get a MN;
                     set field MF of FCN to M;
                     set field PhyAddr of FCN to the address of MN;
                     set field type of MN to new_type;
                  end;
            end;
      end; {case}
   end;
```

4.2.2.5 Instance Generation

The command EDIT is for instance generation which assigns values to a V-Form node. When the command EDIT is given, the interactive mode is switched to the editing phase and the command menu of this phase is prompt in the bottom of the screen. The details about editing phase will be discussed in section 4.3.

4.2.2.6 File Handling

There are two command in this category, namely LOAD and SAVE. LOAD command reloads a prewritten V-Form type. As mentioned

in chapter 3, a V-Form type is similar to a language, while V-Form instances are programs derived from that language. Hence, the LOAD command is used to reload a "language" for "programming".

SAVE command is just the reverse of LOAD. It saves the V-Form type just constructed to a file for further usage.

4.2.2.7 Miscellaneous

The remaining two command: COPY and HOME are included in this category. The COPY command saves the designing effort. It has three options: COPY PROCEDURE, COPY FLOW, and COPY to duplicate the embedded procedure, make the flow of many V-Forms to the same destination, and combine the above two options, respectively. The algorithm for COPY is as follows.

```
procedure VDL$Copy;
   begin
      case options of
         'COPY PROCEDURE':
            begin
               set the field ProcPtr of the destination V-Forms to that
                  of the source V- form;
            end;
         'COPY FLOW':
            begin
               set the fields MF and PhyAddr of the destination
                  V-Forms to the corresponding fields of the source
                  V-Form;
            end;
         'COPY':
            begin
               do the works specified by the above two cases;
            end;
      end; {case}
   end;
```

The command HOME forces the displaying of the first screen defined. It provides a method to return, at any level, to the top of the V-Form syetem.

Furthermore, there is another command in this category which is not the member of the fourteen commands in skeleton phase. It is called RECOVER. During the process of application developing under VDL, there may exist some kind of interruptions such as power failure, reset, or hardware interrupt. This annoying situation will discourage the user because he/she must redo what he/she had done. To remedy this situation, all the interactive commands/data given by the user, whether in skeleton phase or in editing phase, are recorded in a file named VDL.DIA (stands for VDL DIAlog). Upon abnormal leaving of VDL, the command RECOVER can be used **as the first command** to perform recovery. During normal leaving, by giving EXIT to save the defined V-Form instance or by giving QUIT to abort, the file VDL.DIA will be deleted.

4.3 The Editing Phase

4.3.1 BNF Syntax of the editing phase

There are nine primitive operations in the editing phase which are summarized in Table 4.3.1.

In table 4.3.1, a regular expression [12] over the alphabet Σ is

recursively defined as follows:

1) $\varnothing$ is a regular expression and denotes the empty set.
2) λ is a regular expression and denotes the set $\{ \lambda \}$, where λ is the empty string.
3) For each symbol s in Σ, s is a regular expression and denotes the set $\{s\}$.
4) If r and s are regular expressions denoting the sets R and S, respectively, then $(r+s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup S$, RS, and R^*, respectively.

Table 4.3.1 Syntax of the Primitive Operations in VDL Editing Phase

```
<EDITOR>          ::= <EDIT> [<ED_COMMANDS>] <ED_EXIT>
<ED_COMMANDS>     ::= <ED_COMMAND> [<ED_COMMANDS>]
<ED_COMMAND>      ::= <ACTION> | <ALIAS> | <CONDITION>
                    | <FILL> | <GOTO> | <SET_ATTR>
<ACTION>          ::= ACTION [{IS | ARE }] <statements>
<ALIAS>           ::= ALIAS <Regular expression>
<CONDITION>       ::= CONDITION [{IS | ARE}] <statements>
<FILL>            ::= FILL <Regular exspression> ^Z
<GOTO>            ::= GOTO <VFORM_NAME>
<SET_ATTR>        ::= SET [ATTRIBUTE] <ATTRIBUTE>
<ATTRIBUTE>       ::= BOLD | NORMAL | FLASH | INVERSE
<ED_EXIT>         ::= BYE | EXIT | QUIT
```

The <statements> in the command CONDITION and ACTION specify the conditions/actions to be taken. At the present stage, the <statements> is expressed in the form of the Horn clause [18]. Of course, natural language other than the restricted first order predicate can be accepted in the future if the technique has been fully developed.

4.3.2 Primitive Operations of the Editing Phase

Based on the function performed, the above nine primitive operations of editing phase is further divided into four categories, namely, content management, procedure processing, flow manipulation, and miscellaneous. They are described below.

4.3.2.1 Content Management

There are two commands in this category: FILL and SET. The FILL command is used to fill values into each atom. The algorithm for FILL is as follows.

```
procedure EDT$Fill;
   bgein
      if the type of the edited V-Form ≠ 'VFORM' then
         begin
            set the cursor to the left-upper corner of this V-Form;
            while not end_of_file do
               begin
                  get a character;
                  margin control; {control the fillable margin of the
                     V-Form}
                  put the character into the content of a MN;
               end;
         end
      else report the error;
   end;
```

The command SET is used to set the displaying attribute to NORMAL, FLASH, BOLD, or INVERSE vedio. Let us consider the template as shown in fig. 3.5.3. The following interactive commands will generate the V-Form F1 of fig. 3.5.1.

```
VDL> EDIT Title
VDL$ED> FILL  {cursor is now at the left-upper corner of the
               V-Form TEXT:Title, user can now key in text
               'Linear Programming' and control-Z}
VDL$ED> SET BOLD
VDL$ED> EXIT
VDL> EDIT Snoopy
VDL$ED> FILL
_File Name where the graphic codes stored: snoopy.dat
VDL$ED> EXIT
VDL> EDIT Head
VDL$ED> FILL
        .
        .
        .
VDL$ED> EXIT
VDL>
```

4.3.2.2 Procedure Processing

If a V-Form has procedures embedded, then the procedure processing primitives: CONDITION and ACTION can be used to specify the conditions and actions, respectively. The algorithm for CONDITION is as follows.

```
procedure EDT$Condition;
   begin
      if the condition already exist then
         begin
            display the existing condition;
            request for change to or add on a new condition;
            do the appropriate action according to user's reply;
         end
      else
         begin
            prompt 'Condition:';
            accept and save user's input condition;
         end;
   end;
```

The algorithm for ACTION is the same as that of CONDITION except that all the keyword "condition" is changed to "procedure".

For instance, when VFORM:No of fig. 3.5.3 is selected, then the execution of this application is terminated. This situation can be defined by:

```
VDL> EDIT No
VDL$ED> PROCEDURE
_Procedure: Stop
VDL$ED> EXIT
VDL>
```

4.3.2.3 Flow Manipulation

The flow manipulation command GOTO is used to specify the destination of going to. The condition for this control transfer can be specified by the command CONDITION. The algorithm for GOTO is as follows.

```
procedure EDT$Goto;
  begin
    set the field of MF to F;
    set the field ActPtr of AN to the address of the destination
      V-Form;
  end;
```

This command is quite similar to that of SET NEXT at skeleton phase except that users are more freely to define the flow in both phases.

4.3.2.4 Miscellaneous

The other four commands: ALIAS, BYE, EXIT, and QUIT fall into this category. The ALIAS command gives the user an alternative viewing on the V- Form which will override the name of that V-Form when at the stage of execution. The algorithm for ALIAS is as follows.

```
procedure EDT$Alias;
  begin
    if the type of the V-Form is 'VFORM' then
      begin
        set cursor to the left-upper corner of this V-Form;
        get a MN;
        set the field AliasPtr to the address of the MN;
        read in the alias;
        store the alias into the content of the MN;
      end
    else report the error;
  end;
```

In contrary to the FILL command which operates on V-Forms with type not equal to "VFORM", the alias command operates on V-Forms with type "VFORM". For example, consider one of the screens of the application CAI_Course as shown in fig. 4.3.1.

The content "(1) (0,0)" is the alias of VFORM:1 which is defined by the ALIAS command as follows:

```
VDL> EDIT 1
VDL$ED> ALIAS        {cursor is now set to the left-upper corner of
                     VForm:1, user can fill in '(1) (0,0)'}
VDL$ED> EXIT
VDL>
```

Finally, the command EXIT saves the edited data and leaves the editing phase, the command QUIT aborts the modified data and leaves the editing phase, while the command BYE saves the data changed and leaves both the editing phase and VDL. (Of course, the V-Form instance is also saved by the BYE command).

```
TEXT:Quiz                                       GRAPH:Fig
Answer the following question:                  Fig3.dat

Suppose x >= 0, y >= 0,
        x + y <= 120, x - y >= 60.

For ax + by = F(x,y),
    a,b are any real numbers.

Which of the following can NEVER be the
maximum?

VFORM:1        VFORM:2        VFORM:3         VFORM:4         VFORM:5
(1) (0,0)      (2) (0,60)     (3) (70,45)     (4) (30,90)     (5) (120,0)

VDL$ED>
```

Fig. 4.3.1 An example showing the usage of ALIAS

4.4 A Prototype Graphics Utility

The prototype graphics utility is designed only to provide the graphical entry of V-Forms for demonstration purpose. It uses the graphic primitives of VT240 made by Digital Equipment Corporation. The drawing mode provided by this utility are LOCATE, LINE, ERASE, TEXT, BLOCK, CIRCLE, EXIT, and QUIT as shown in fig. 4.4.1, where the cross is automatically generated by the graphic primitive "R(P(I))" which means "report position interactively".

The window in fig. 4.4.1 is specified by the user which is the same as the relative position and size of V-Form in the V-Form system. In LOCATE mode, the cross can be moved to any position within the

window by the cursor key. In LINE mode, lines can be ploted at will by the cursor key. The ERASE mode supports an eraser for the user to erase the redundancy plotted. The TEXT mode supports the text entry. While BLOCK and CIRCLE modes draw a block or a circle by giving two diagonal points or a center and a point on the circumference, respectively. After completion, the EXIT command saves the plotted vectors into the file specified by the user.

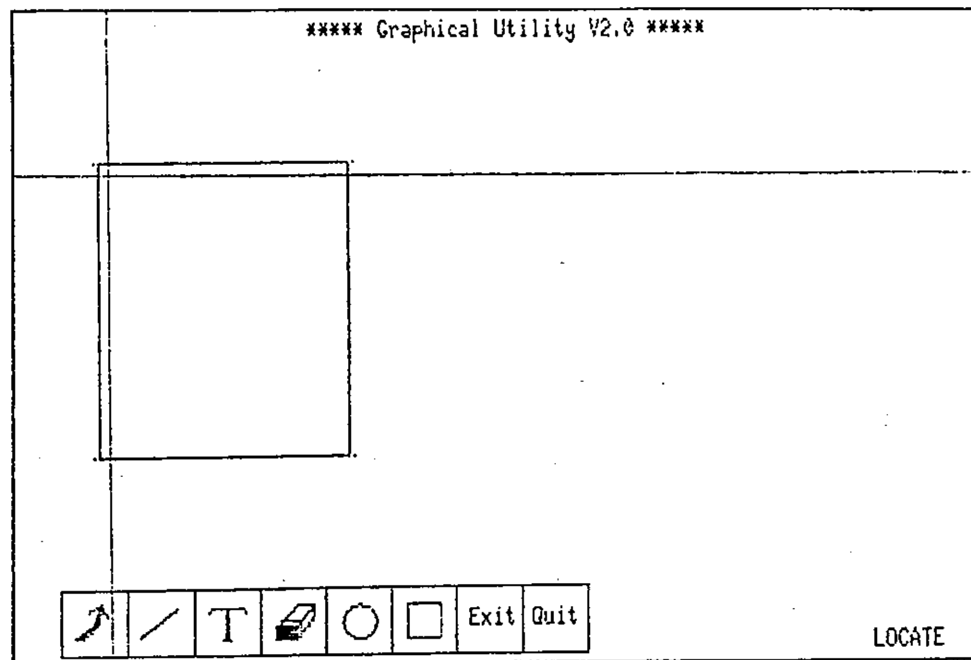


Fig. 4.4.1 A typical screen layout of the graphics utility

LOCATE mode is the default as shown in the right-down corner of fig. 4.4.1. A double-click on RETURN key moves the cross to the menu and allows the user to change the graphic mode to the one required by using the cursor key.

4.5 An Illustrative Example

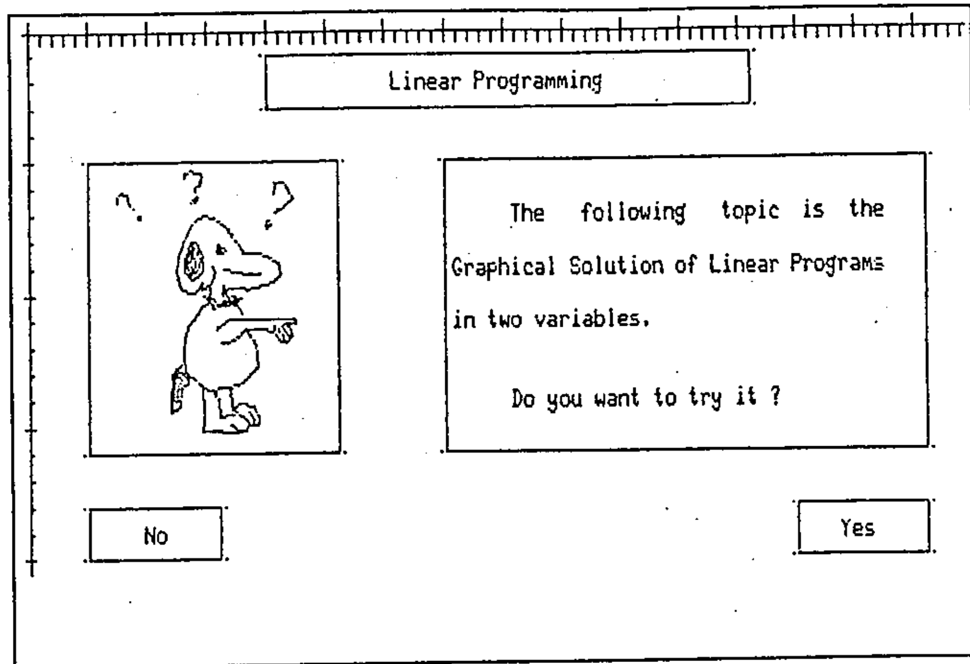
4.5.1 Problem Statement

Consider a CAI Courseware which teaches the student the concept of Graphical Solution to Linear Programs in two variables. At the beginning, a screen is displayed to show the topic of the problem as shown in fig. 4.5.1(a).

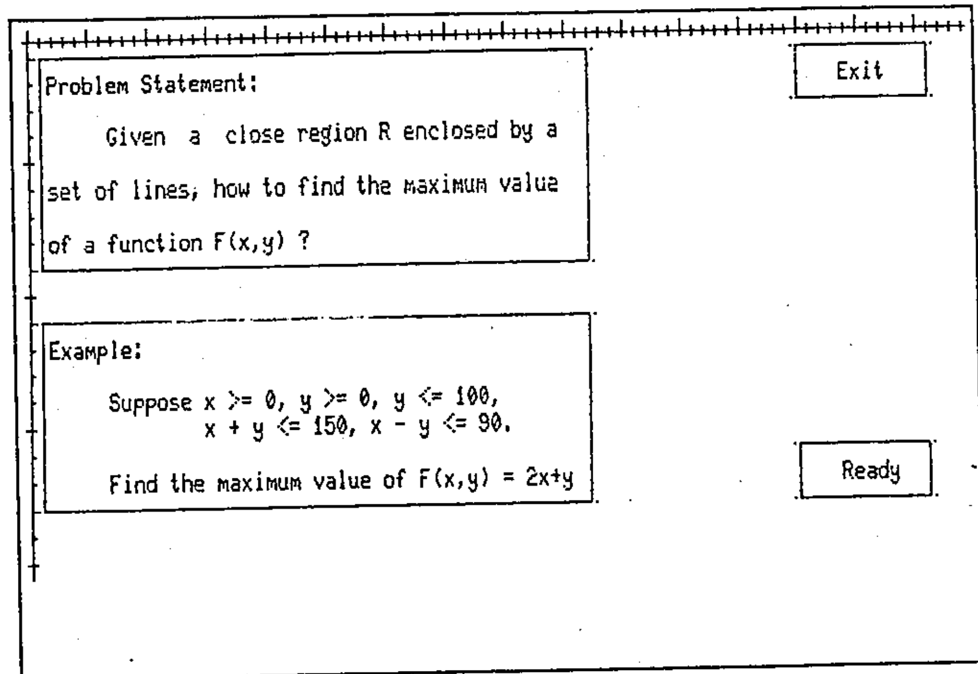
If the user want to use this courseware to explore the concept of graphical solution, then the touch block "Yes" on the screen is selected and switched to next screen which shows the problem statement and gives an example of finding the maximum value of a function on a region as shown in fig. 4.5.1(b).

After a sequence of trying as shown in fig. 4.5.1(c)(d), a message is given to ask the user whether he/she is ready for quizzing. If he/she is not ready, then fig. 4.5.1(c) is displayed for reviewal. Otherwise, fig. 4.5.1(f) is displayed. As we know, "(3) (70,45)" is the correct answer. If it is selected, then fig. 4.5.1(h) is displayed. If any of the other four answer is chosen, then fig. 4.5.1(g) is displayed to give an explanation.

Notice that, in fig. 4.5.1(d), the values 45 and 78 are given by the user when "Yes" of fig. 4.5.1(c) is selected, while the value 168 is computed automatically. Of course, these procedures must be specified by the courseware designer.

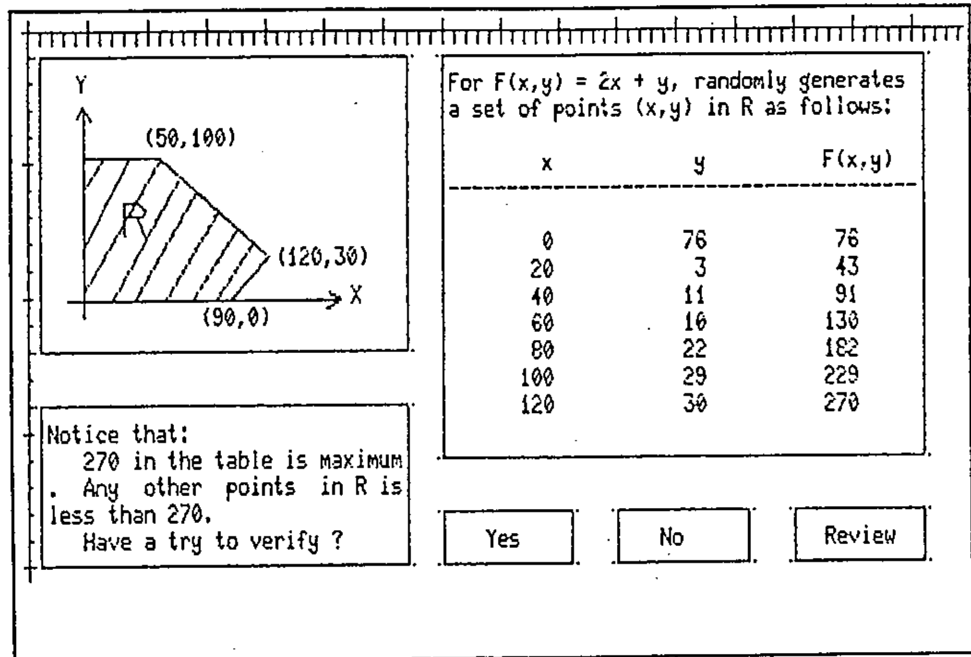


(a) F1

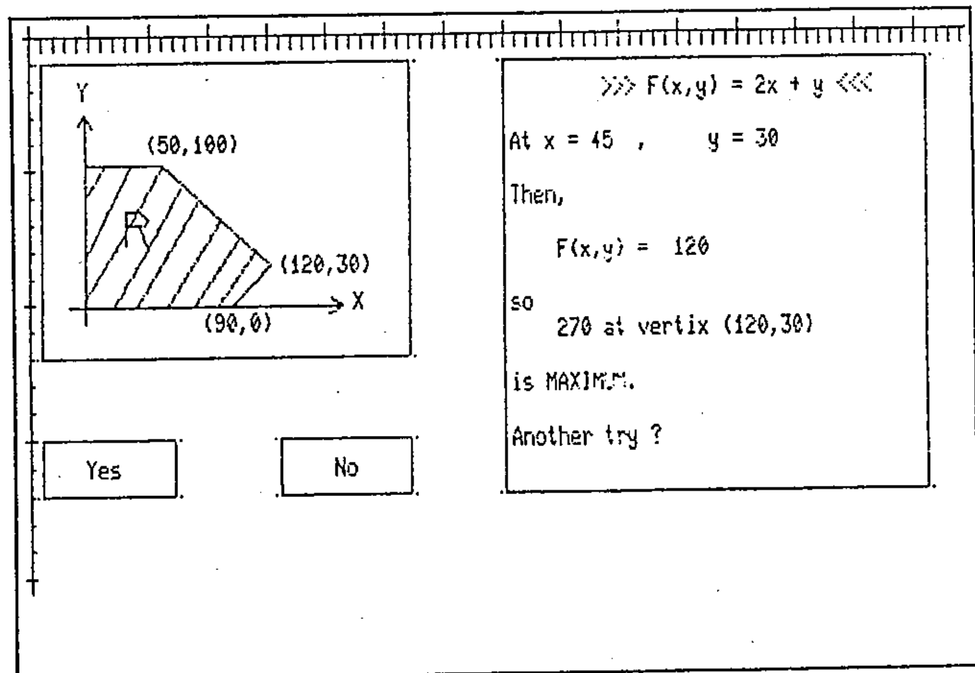


(b) F2

Fig. 4.5.1 The application CAI_Course



(c) F3



(d) F4

Fig. 4.5.1 The application CAI_Course (continued)

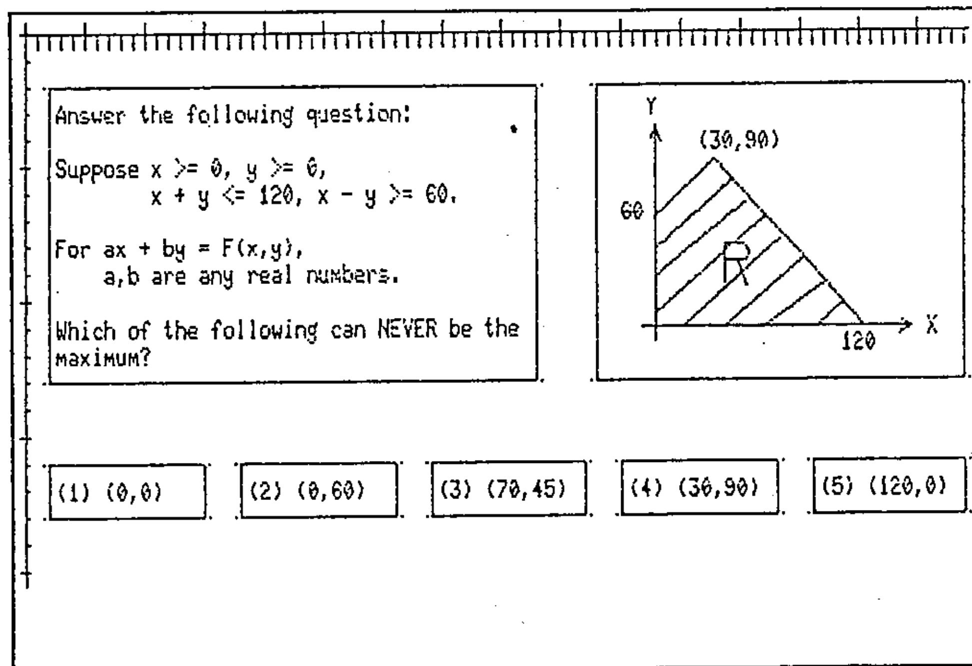
In the following, we will give you a quiz.

A R E Y O U R E A D Y ?

Yes

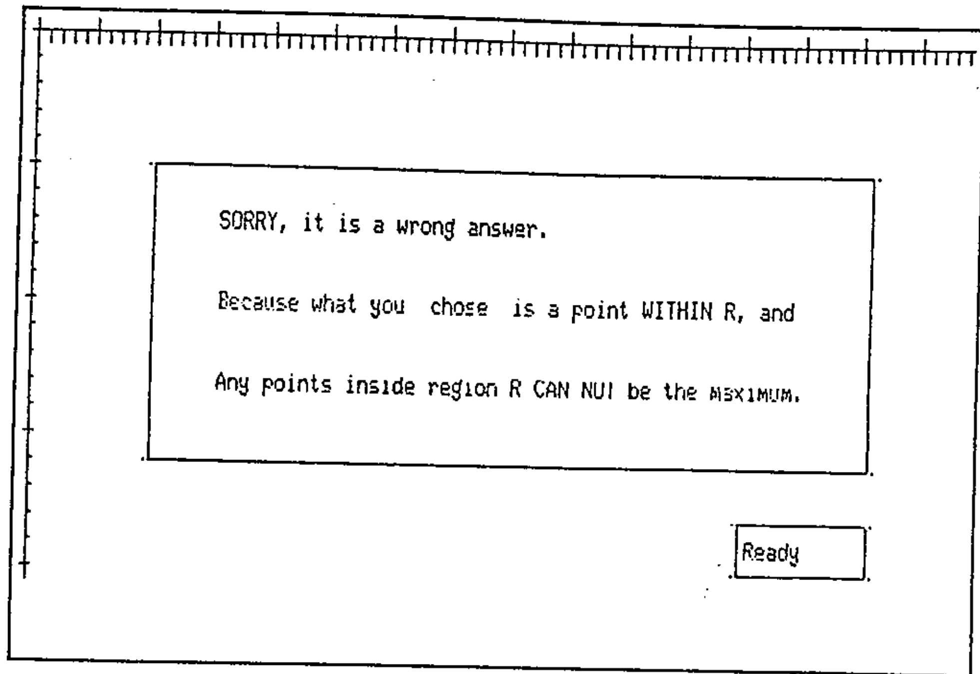
No

(e) F5

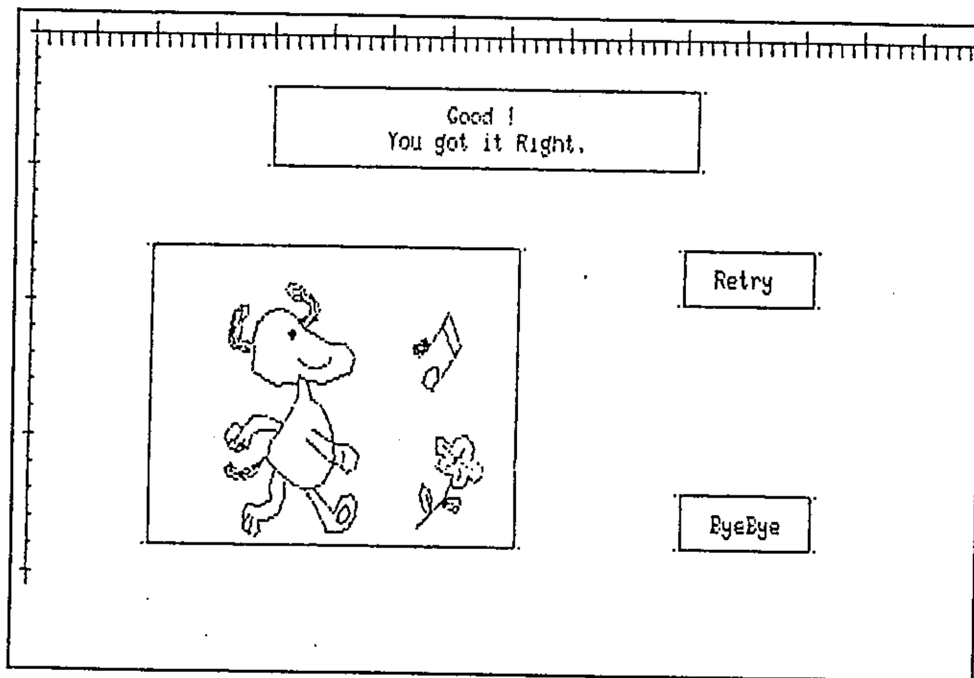


(f) F6

Fig. 4.5.1 The application CAI_Course (continued)



(g) F7



(h) F8

Fig. 4.5.1 The application CAI_Course (continued)

4.5.2 Start up

To develop this courseware, first, a set of well-written sheets of courseware is prepared. Second, the Automatic Program-Synthesizing (APS) system, which is an integration of graphical utility, VDL, V-Prolog, and some useful commands, is invoked as shown in fig. 4.5.2.

Third, a VDL is invoked by giving the command VDL after the prompt in fig. 4.5.2 and fig. 4.5.3 is displayed.

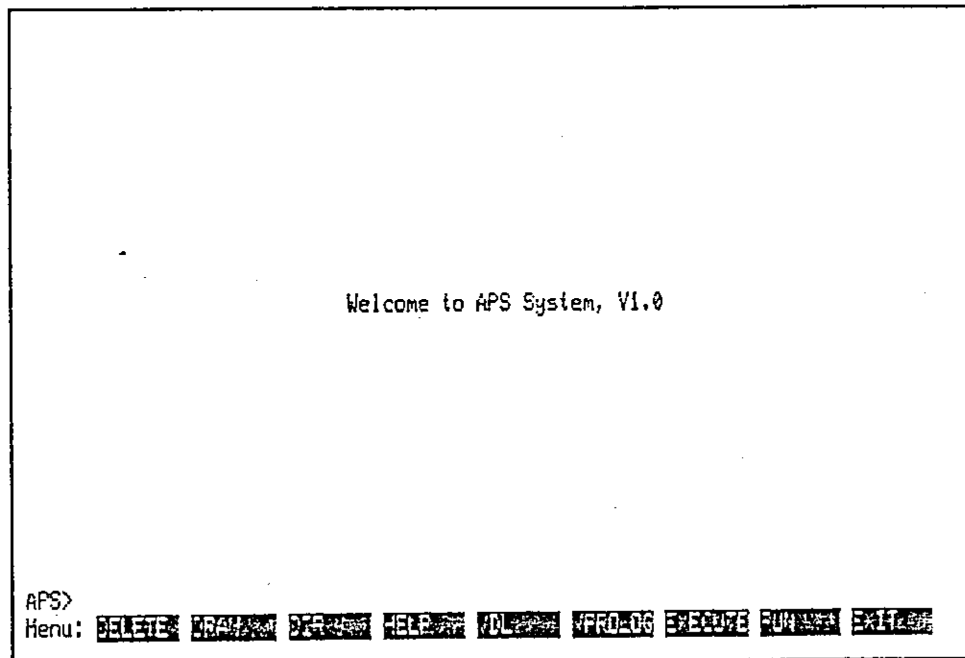


Fig. 4.5.2 Screen layout of the proposed APS system

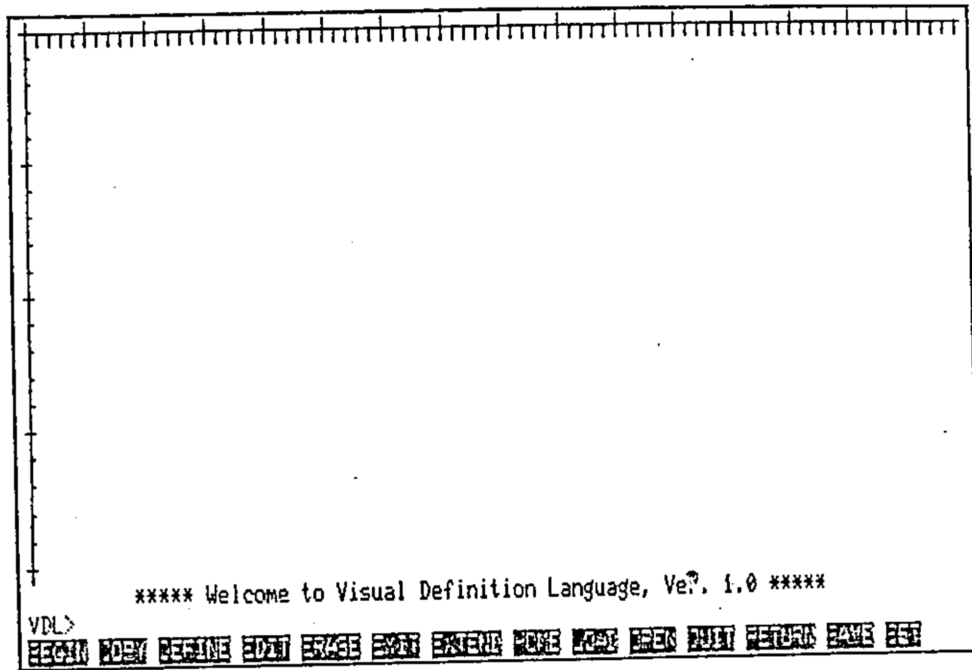


Fig. 4.5.3 Screen layout of VDL

Fourth, the graphics utility as described in section 4.4 can be invoked after the completion of VDL by the command

APS> DRAW

Finally, the mapping and execution of the developed V-Form system can be done under the command EXECUTE and RUN which will be discussed in chapter 5.

4.5.3 Interactive Sequence

The interactive sequence for the courseware CAI_Course is given in appendix B. After the completion of VDL, the internal structure is obtained. Furthermore, the control flow, which gives the flow of the sheets of courseware in this example, is automatically synthesized as shown in fig. 4.5.4.

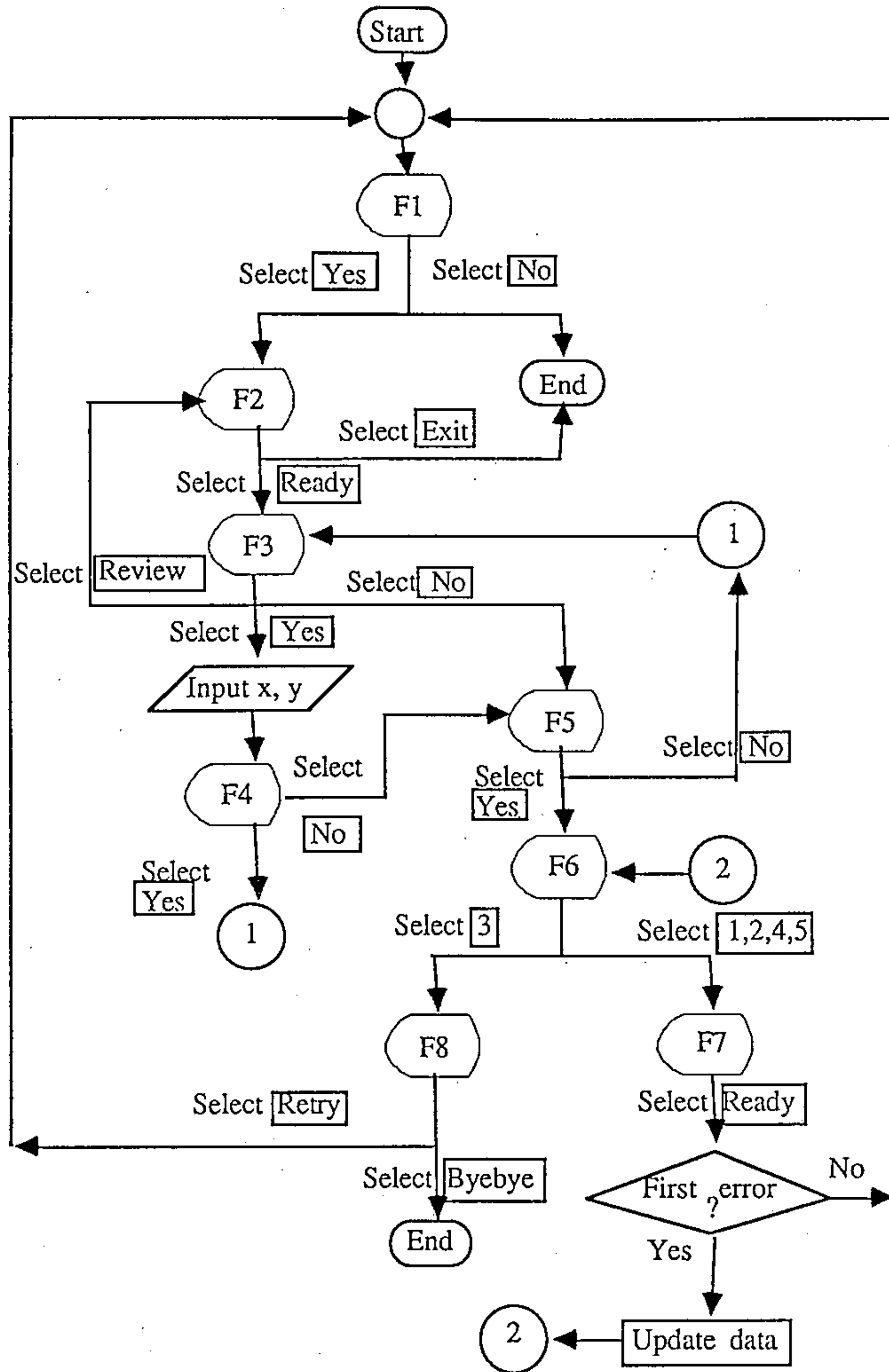


Fig. 4.5.4 Control flow of CAI_Course

CHAPTER 5

INTERPRETING V-FORMS UNDER PROLOG

5.1 Introduction

Using the primitive operations of VDL, we can complete the sketching of an application program. For example, the CAI courseware which demonstrates the Graphical Solution to Linear Programs in two variables as shown in section 4.5 can be obtained. In consideration of the portability, extensibility, and sometimes in need of unification and backtracking during the execution of a V-Form instance, a current existing CProlog interpreter with a slight modification is adopted to interpret the self-synthesized internal structure.

With this approach, a mapping algorithm is needed to translate the internal structure into Prolog facts and rules. According to this algorithm, the system proposed by this report can be easily ported to another computer for execution if there has an available Prolog interpreter.

5.2 Background

5.2.1 Predicate

Computer programming in Prolog consists of:

1. Declaring some **facts** about object and their relationships,
2. Defining **rules** about objects and their relationships, and
3. Asking **questions** about objects and their relationships.

where, in BNF form:

```
<FACT>       ::= <TERM> .
<RULE>       ::= <TERM> :- <TERM_LIST> .
<QUESTION>   ::= ?- <TERM_LIST> .
<TERM>       ::= <Atom> [<Arguments>]
<Atom>       ::= <Lowercase_letter> <Alphanumerics>
<Arguments>  ::= ( <TERM_LIST> )
<TERM_LIST>  ::= <Object> [<Terms>]
<Terms>      ::= , <Object> [<Terms>]
<Object>     ::= <Atom> | <Variable> | <Term>
<Variable>   ::= {<Uppercase_letter> | _ } <alphanumerics>
```

These three elements: fact, rule, and question constitute the Prolog program. After supplying all the facts and rules about objects and their relationships, Prolog system will enable a computer to be used as a storehouse of facts and rules, and it provides ways to make inferences from one fact to another such that question can be answered, theorem can be proved, and goal can be achieved. The name of the relationship, i.e. the <Atom> in the above BNF form, is called the predicate. For example, in the fact

likes(john,mary).

which means "John likes Mary", the relationship **likes** is called the predicate.

5.2.2 Unification

When a question is asked, Prolog will search the facts through the storehouse that you have typed in before. It looks for facts that match the fact in the question. Two facts are said to be matched if and only if their predicates are the same and each of their corresponding arguments are the same.

In the formal definition [3], the process of matching is called the unification.

5.2.3 Backtracking

Consider a Prolog question where a conjunction of goals arranged from left to right, separated by commas, are to be matched.

$$?- \ g_1,\ g_2,\ g_3,\ \ldots,\ g_{i-2},\ g_{i-1},\ g_i,\ \ldots,\ g_n.$$

Suppose g_k can be successfully matched for $1 \leq k \leq i-1$, but the unification of g_i fails. In this case, Prolog tries to re-satisfy the goal g_{i-1} by searching the storehouse to find an alternative unification. If this unification succeeds, the matching on g_j for $j \geq i$ proceeds. Otherwise,

Prolog tries to re-satisfy g_{i-2}. This unification process will go on repeatly until the goal g_n is satisfied or the re-satisfication of g_1 is failed. In the former case, the entire conjunction of question succeeds and the answers are given by Prolog. While in the later case, the entire conjunction fails. This behavior, where Prolog repeatedly attempts to satisfy and re-satisfy goals in a conjunction, is called the backtracking [6].

5.2.4 CProlog Interpreter

CProlog [20] was developed at EdCAAD, Department of Architecture, University of Edinburgh. It is written in C Programming Language and is complete in view of the build-in function support. CProlog is almost the same as the core-Prolog described in the text by Clocksin [6] and runs reasonably fast.

CProlog offers the user an interactive programming environment with tools for incrementally building programs, debugging programs by following their executions, and modifying parts of the programs without having to start again from scratch. In addition to the integer arithmetic build-in functions as described in core-Prolog, CProlog also provides real arithmetic operations. Furthermore, the scientific functions such as logarithm, square root, sine, cosine, arc sine, arc cosine, are also supported which will be useful in developing the application programs.

5.3 THE V-PROLOG INTERPRETER

The V-Prolog (Visual Prolog) interpreter is based on the CProlog with a slight modification to explore the visual properties of V-Forms. The structure of V-Prolog is shown in fig. 5.3.1.

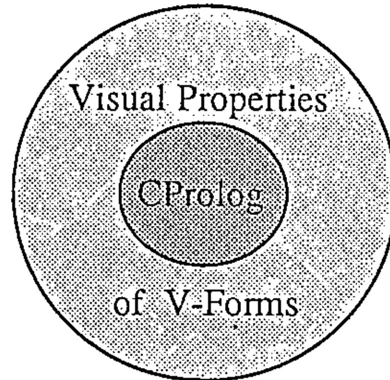


Fig. 5.3.1 The Structure of V-Prolog

The visual properties of V-Forms are explored by the system predicates added. These system predicates are further divided into two parts, namely, machine independent predicates and machine dependent predicates. All the other predicates not mentioned are left unchanged. For detail information about Prolog programming, please consult the CProlog manual and the text book by Clocksin.

5.3.1 The Machine-Independent System Predicates

5.3.1.1 Procedure Evaluation

The embedded procedures are evaluated by the system predicates: **sys$eval_atom_proc** and **sys$eval_fcn_proc** according to the following rules

Rule 1: **If** the embedded procedure is attached to an atom
then
it is performed AFTER that atom is displayed.

Rule 2: **If** the embedded procedure is attached to a non-atom
then
it is performed when that V-Form is selected and BEFORE the sub-VForms of it is displayed.

In order to avoid the delays during the procedure evaluation process of the atoms (which may be in a manner of:

display → evaluate procedure → display → evaluate procedure → ...

i.e., a V-Form is not visible until the contents and procedures of the previous V-Forms are displayed and evaluated), the embedded procedures are collected into a set and executed after the V-Forms of the whole screen are displayed.

5.3.1.2 Control Strategy

The control strategy predicates handle the selection and flow of V-Forms during interpretation. The system predicate **sys$fcn** controls the form control table and causes the sub-VForms of the selected V-Form to be displayed.

The predicate **sys$fired**, **sys$valid**, and **sys$position** checks the validation of a V-Form according to the input coordinate given by the user. The predicate **sys$goto** gives an unconditional control transfer to a list of V-Forms, while the predicate **sys$stop** halts the execution of the Prolog program.

5.3.1.3 Database Management

In Prolog, a collection of facts is called the database [6]. During the interpretation, V-Forms are divided into two categories: selectable or unselectable. The selectable V-Forms, which can be selected by the user, contains one or more sub-VForms, while the unselectable V-Forms are atoms. The information about the selectable V-Forms is asserted as a fact: **workset** in the database. The system predicate **sys$make_workset** is used, as the name stands, to make this workset.

Finally, the set of embedded procedure, as mentioned in section 5.3.1.1, are collected by the predicate: **sys$make_procedure_set**.

5.3.1.4 Window Manipulation

The system predicates of this category display the contents of V-Forms according to the following algorithm:

```
procedure sys$show;
   begin
      while there are V-Forms on the FCT to be displayed do
         begin
            case type of
               'VFORM':
                  begin
                     if there exists the alias then
                        display the alias of the V-Form
                     else display the name of the V-Form;
                  end;
               'TEXT':
                  begin
                     display the content using the system predicate:
                     sys$write_content;
                  end;
               'GRAPH', 'BIT-MAP', 'ANIMATION':
                  begin
                     invoke the graphical subsystem to draw by using
                        the predicate: system;
                  end;
               'RULE':
                  begin
                     interpret it directly
                  end;
            end; {case}
         end; {while}
   end;
```

5.3.2 The Machine-Dependent System Predicates

The machine-dependent system predicates can also be divided into four categories, namely, screen management, vedio control, system communication and miscellaneous. The machine factor considered here is the VT100 or VT100-compatible terminal.

5.3.2.1 Screen Management/Vedio Control

The system predicates of these two categories use ANSI standard escape sequence, which is a sequence of one or more ASCII graphic characters preceded by the ESC character, to control the screen and vedio display. These escape sequence applies to VT100 or VT100-compatible terminals.

The predicate **sys$cls** clears the whole screen. The predicate **sys$home** repositions the cursor to the left-upper corner of the screen. The predicate **sys$draw_margin** draws the scaled margin as shown in fig. 5.4.1. The predicates **sys$open_window** and **sys$set_cursor**, as their names stand, opens the window and sets the cursor to the required position, respectively.

The vedio control system predicates: **sys$flash_text**, **sys$normal_text**, **sys$inverse_text** set the vedio background to flash, normal, and inverse, respectively. While the predicates **sys$text_mode** and **sys$graph_mode** switch the character sets of the terminal.

5.3.2.2 System Communication

The predicate **sys$get_response** retracts user's input coordinates upon the selected V-Forms. If other input devices shch as touch panel, mouse, or pointing devices, are used, then a slight modification on this predicate is required to let the system work properly.

The predicate **system** provided by CProlog allows the invoking of the graphics subsystem and the extending of the APS system.

5.3.2.3 Miscellaneous

The system predicate **sys$init** initiates the screen and divides it into a scratchpad region, which is the region for V-Forms to be displayed, and an interactive region, which is used to communicate with the user. The predicate **sys$final** restores the screen and halts the execution of Prolog. Finally, the predicate **sys$execute** starts the execution of the mapped Prolog program.

5.4 Interpreting Environment

V-Prolog interpreter is used as the kernal of the interpreting environment. In order to let the internal structure interpretable by V-Prolog, a mapping algorithm is used to translate the internal structure into Prolog facts and rules. In this section, a mapping algorithm as well as an example of the courseware CAI_Course will be discussed and demonstrated to illustrate the mapping and execution processes.

5.4.1 Mapping Algorithm

The mapping algorithm, which is used to translate the self-synthesized internal structure into Prolog facts and rules, is a six-tuple $M_P = (V_I, V_P, P, T, L, S)$, where

1. V_I is a finite set of nodes and information (about the screen margin) in the internal structure,
2. V_P is a finite set of Prolog facts, rules, and questions,
3. P is a finite set of productions,
4. T is a finite set of strategies to support the generation of Prolog programs,
5. L is a finite set of production labels, and
6. S is the starting information, $S \in V_I$.

The productions of M_P are of the form

$$(r) \qquad A \rightarrow \beta \qquad T_S \qquad S(U) \qquad F(W)$$

as shown in table 5.4.1, where $A \rightarrow \beta$ is called the core, $A \in V_I, \beta \in (V_I \cup V_P)^*$, (r) is the label, $r \in L$. U is the success field and W is the failure field. $U, W \subset L$. T_S is the strategy being taken, $T_S \subset T$.

Table 5.4.1 The productions of the mapping algorithm M_P

(r)	Core	T_S	S(U)	F(W)
1	$S \rightarrow BCx$	$\{\emptyset\}$	{2, 3}	$\{\emptyset\}$
2	$B \rightarrow \alpha B$	$\{\emptyset\}$	{2, 3}	$\{\emptyset\}$
3	$B \rightarrow \beta$	{A1, A2}	{3}	{4, 5}
4	$C \rightarrow CC$	$\{\emptyset\}$	{4, 5}	$\{\emptyset\}$
5	$C \rightarrow \gamma$	{A1, A2, A3}	{5}	$\{\emptyset\}$

Notice that, in table 5.4.1, the notation α, β, γ, x, A_1, A_2, and A_3 represents:

1. $\alpha \in V_P$, and denotes the fact:

   /* 1.1 */ margin(Id, Row, Column, Width, Height).

2. $\beta \in V_P^*$, and denotes the following set of rules and facts:

   /* 2.1 */ vform(Name, Id, Alias, Row, Column, Width, Height, Indicator, Sub_VForm_list).

   /* 2.2 */ coordinate(Row, Column).

   /* 2.3 */ workset(VForm_id_list).

   /* 2.4, optional, depends on A_1 */

   procedure(Id) :- conditions & actions.

   /* 2.5 */ go :- sys$valid(Id), sys$fcn(Id).

3. $\gamma \in V_P^*$, and denotes the following set of rules and facts:

   /* 3.1 */ vform(Name, Id, Alias, Row, Column, Width, Height, Indicator, Sub_VForm_list).

   /* 3.2, optional, depends on A_1 */

   procedure(Id) :- conditions & actions.

   /* 3.3,optional, depends on A_3*/

   content(Name:Id, Type, Attribute, The_contents).

   /* 3.4, optional, depends on A_3 */

   go :- sys$valid(Id), sys$fcn(Id).

4. $x \in V_P$, and denotes the question:

```
/* 4.1 */   ?- sys$execute.
```

5. $A_1 \in T$, and denotes the strategy:

```
if the V-Form has embedded procedure then
   begin
      assert the predicate: procedure;
   end;
```

6. $A_2 \in T$, and denotes the strategy:

```
begin
      search the sub-VForms that has a direct link to the
         V-Form encountered;
      append the searched sub-VForms' names and ids
         into a list;
      sub_Vform_list  ← the appended list;
end;
```

7. $A_3 \in T$, and denotes the strategy:

```
if this V-Form is an atom then
   begin
      assert the predicate: content:
   end
else
   begin
      assert the rule: go;
   end;
```

The mapping algorithm operates as follows:

Step 1: Production (1) is applied first,

Step 2: For production (i), $i \in L$,

if the internal structure α contains the message Λ,
$\Lambda \in V_I$
then
the production (r) $\Lambda \rightarrow \beta \quad T_S$ is applied according to the strategy T_S and the next production is selected from the success go-to field U.
else
the production (r) is not used (i.e. α is not changed), and the next production is selected form the failure go-to field W.

Step 3: **If** the applicable go-to field contains $\varnothing$ **then**
the mapping halts
else go to step 2.

In this algorithm, the information kept in SN, FCN, MN, and AN are directly mapped to the predicates: vform, vform, content, and procedure, respectively. The relationship between these four nodes are mapped into Prolog program according to the productions as shown in table 5.4.1.

5.4.2 An Example

Consider the courseware CAI_Course of fig. 4.5.1 with part of the internal structure shown in fig. 3.5.4. Applying the mapping algorithm to the internal structure, we have the Prolog program (only part of it is shown):

```
margin( 5, 1, 1, 77, 18).
margin( 9, 1, 1, 77, 18).
margin( 13, 1, 1, 77, 18).
margin( 14, 1, 1, 77, 18).
margin( 21, 1, 1, 77, 18).
margin( 25, 1, 1, 77, 18).
margin( 27, 1, 1, 77, 18).

vform('Cai',0,_,0,0,0,0,1,['Title':1,'Snoopy':2,'Head':3,'No':4,'Yes
':5]).
coordinate(0,0).
workset([0]).
procedure_set([]).
go :- sys$valid(0), sys$fcn(0).

vform('Title', 1,0, 1,20,40, 1,1,[]).
content('Title':1,text,normal,'        Linear Programming~~').

vform('Snoopy', 2,0, 5, 5,20,10,1,[]).
content('Snoopy':2,graph,normal,'snoopy.dat~~').

vform('Head', 3,0, 5,35,40,10,1,[]).
content('Head':3,text,normal,'~   The   following   topic  is
the~~Graphical Solution of Linear Programs~~i').
content('Head':3,text,normal,'n two variables.~~~    Do you want to
try ?~~' ).

vform('No', 4,'   No',18, 5,10, 1,1,[]).
procedure( 4) :- stop.
go :- sys$valid( 4),sys$fcn( 4).

vform('Yes', 5,'  Yes',18,65,10,1,1,
['Prob':6,'Ex':7,'Exit':8,'Ready':9]).
go :- sys$valid( 5),sys$fcn( 5).
.
.
.

vform('Desc', 17,0, 1,40,35,15,1,[]).
procedure( 17) :- value(X,Y),cursor(5,49),write(X),cursor(5,63),
write(Y).
procedure( 17) :- retract(value(X,Y)),cursor(9,56), Z is 2*X+Y,
write(Z) .
content('Desc':17,text,normal,'      >>> F(x,y) = 2x + y <<<~~At x =
  , y =~~Then,~~   F(x,y) =~~so~ ').
content('Desc':17,text,normal,'  270 at vertix (120,30)~~is
MAXIMUM.~~Another t ry ?~~').
.
.
.
```

```
vform('Good', 32,0, 2,20,35, 2,1,[]).
content('Good':32,text,inverse,'          Good !~        You got it
Right.~ ~').

vform('Snoopy', 33,0, 8,10,30,10,1,[]).
content('Snoop':33,graph,normal,'snoopy2.dat~~').

vform('Retry', 34,' Retry', 8,55,10, 1,1,
['Title':1,'Snoopy':2,'Head':3, 'No':4, 'Yes':5]).
go :- sys$valid( 34),sys$fcn( 34).

vform('ByeBye', 35,' ByeBye',17,55,10, 1,1,[]).
procedure( 35) :- stop.
go :- sys$valid( 35),sys$fcn( 35).

?- sys$execute.
```

The facts "margin(id,row,col,width,height)" direct the windowing during execution. At the beginning, the facts: coordinate(0,0), workset([0]), and procedure_set([]) give an unconditional display of the screen shown in fig. 4.5.1(a) and then an unification is initiated by the system predicate: **sys$fcn** in the rule **go**. If the V-Form "No" in fig. 4.5.1(a) is selected which in this stage is the supporting of two integer values by the user, then the embedded procedure in this V-Form (in this case, the fact: procedure(4)) is executed to exit from this application. On the other hand, if "Yes" is selected, then the sub-VForms within VFORM:Yes are displayed (by a pointer to FCT (F2) as in fig. 3.5.4). The user can proceed in this way to "run" the application visually. The resulting execution flow is shown in fig. 4.5.4. Accordingly, users can program or run their applications in visual.

The mapping is invoked by using the command EXECUTE in fig. 4.5.2. which is a combination of mapping and running. On the other hand, the command RUN in fig. 4.5.2 runs a mapped program. Fig. 5.4.1 shows a typical screen during execution of the CAI courseware.

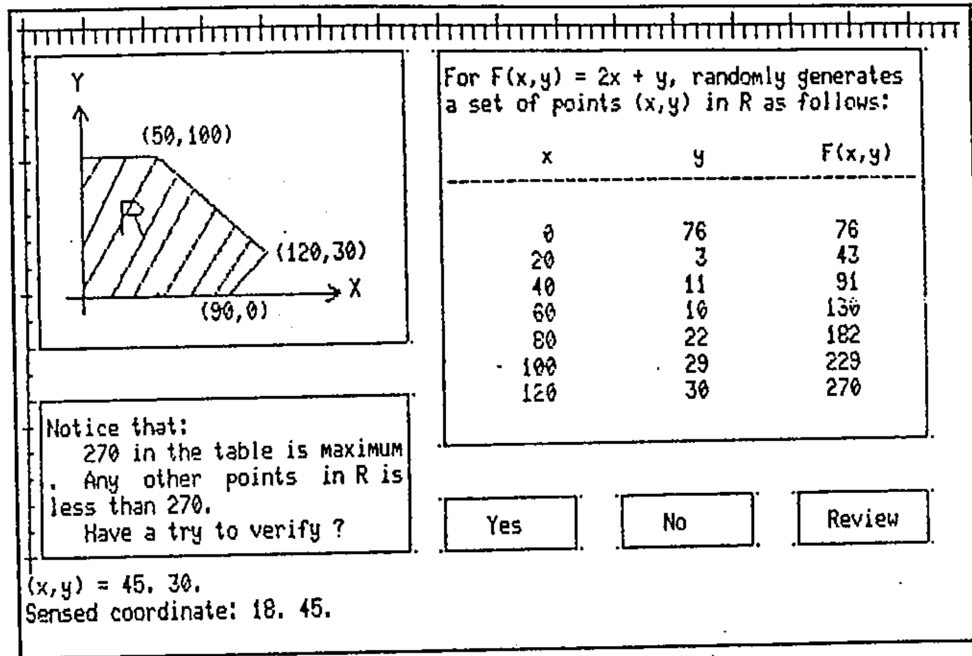


Fig. 5.4.1 A Typical Screen During Execution

CHAPTER 6

CONCLUDING REMARKS

In this report, a theoretical sound approach to visual program synthesizing system as a footstone toward automatic programming for Non-Programmer Professionals has been presented. The visual program-synthesizing system contains a V-Form Definition Language, a prototype graphics utility, and a V-Prolog interpreter. The V-Form Definition Language offers the NPP users a programming environment to define objects of an application program in a manner of what-they-sketch-is-what-they-get and automatically synthesizes the sketched objects into a consistent internal structure. The prototype graphics utility is designed to provide the entry of graphical objects. While the V-Prolog interpreter provides the users a visual environment to run their application programs.

The interface between the V-Form Definition Language and the V-Prolog is a mapping algorithm which transforms the consistent internal structure of the sketched objects by VDL into an executable Prolog program which can be interpreted by V-Prolog. VDL, V-Prolog, and the mapping algorithm are all based on a V-Form model in which the contents and the structure of the objects sketched by the user are explicitly incorporated. Accordingly, the users can sketch freely the objects whose

contents may include text, static graphics, dynamic graphics, and rules.

The Automatic Program-Synthesizing system provides the user a feasible environment to program their applications and run the application programs visually. Users, with or without any programming language background can focus on describing their specialized knowledge through the processing of VDL to get the desired applications. This will release the users from doing ardious and tedious works. Also, because of its sound theoretical basis, the APS system may be easily extended to deal with more knowledgable objects.

In formal language aspect, the APS system introduces a visual programming concept which is used to get the desired application programs in a manner of what-they-sketch-is-what-they-get operations. Through these operations, the behavior of the visual grammar G_V of a visual language environment can be simulated. In other words, the visual programming itself is an interactive process which allows the users to describe the inference algorithm by themself. Here, the inference algorithm is a process of describing the positive samples as designed by the user. In this way, based on the visual grammar which is a context-free programmed grammar, the context-sensitive applications can be synthesized by the APS system.

Although, at the present stage, APS system is only an experimental system in the study of visual programming techniques, it already reveals its capabilities in flexibility, extensibility, and adaptability for application software development. However, there is still a lot to do in the future research. This may include expressing more knowledgable objects and

their relations, integrating the APS system into personal computers or even workstations, supporting more application packages, and so forth. But first, it is suggested a better understanding on visual perception such that more knowledgable representations for wider objects can be devised in a more general manner.

APPENDIX A

INSTALLATION OF APS SYSTEM

A. 1 START UP

To install the APS system, first of all, the following files must be copied from the distribution directory:

APS.COM	APSLIB.SYS	APSMENU.PAS	BOOT.COM
CAI.DIA	CPROLOG.EXE	DRAW.PAS	DRAWMENU.PAS
EDIT.PAS	EDTCMD.SYS	ERRFILE.SYS	HEAD.PAS
INIT.SYS	LIB.PAS	LIBDATA.SYS	MAPPING.PAS
PLOT.PAS	PROTECT.SYS	VDL.DEF	VDL.PAS
VDLCMD.SYS	VMSALL.SYS		

The files with extension .SYS are system files which are needed during the installation process. The source programs are written in VAX-11 Pascal under VAX/VMS. The file CPROLOG.EXE is the CProlog interpreter available from DEC. All the above files take about 627 blocks of disk quota (512 byte/block).

A.2 INSTALLATION

Assume all necessary files are copied to your working directory. In the installation procedure, a temporary space of about 700 blocks is needed. You can give the following command to start installation:

$@BOOT

and you will see:

```
>>>>>        Install APS system, V1.0        <<<<<

Assuming that you've copied all the files from the
distribution directory, this procedure will procede to
install the V-PROLOG interpreter and set up the APS
system and define the APS command accordingly.


First, I'll set up the PRO$LIBRARY logical name to point
to the subdirectory "LIBRARY.DIR" of the current
directory, and I'll define the PROLOG command
accordingly.

(ultimately PRO$LIBRARY should point to a central
directory accessible to all users, then the files
"VPROLOG.EXE" and "STARTUP." should be moved there, and
the PROLOG command should be redefined appropriately)

Done!

now then, I'll go through the "Interpreter Bootstrap"
procedure

CProlog version 1.5

This software should not be used for commercial purposes
and should not be distributed outside of Digital
(per Digital's agreement with the Univ. of Edinburgh).

This software is still being field tested.

Contact Michael Poe or Roger Nasr for more information.

[ Bootstrapping session. Initializing from file init.sys ]
| ?- vmsall.sys consulted 15532 bytes 13.316689 sec.
yes
| ?-
yes
apslib.sys consulted 9804 bytes 7.483345 sec.

yes
| ?- protect.sys consulted 372 bytes 10.300003 sec.

yes
| ?- libdata.sys consulted 408 bytes 0.350028 sec.

yes
| ?-
[ closing all files ]
```

```
| ?-
[ Prolog execution halted ]

The V-PROLOG interpreter should be all set now! (with
PRO$LIBRARY defined as the subdirectory "LIBRARY.DIR"
of the current directory)

please let me know if you have any problems with the
V-PROLOG installation procedure.

Second, I'll set up the APS$LIBRARY logical name to
point to the subdirectory "LIBRARY.DIR" of the current
directory, and I'll define the APS command accordingly.

(ultimately APS$LIBRARY should point to a central
directory accessible to all users, then all the files
in the subdirectory "LIBRARY.DIR" should be moved there,
and the APS command should be redefined appropriately)


The APS system should be all set now!

Please let me know if you have any problems with the
bootstraping procedure, or with this new version of APS
system that you just installed...

The APS.MAN manual, available from APS$SOURCE:, is
still not the latest. You will be notified when the new
manual is available.

This software is still being field tested. For more
information about APS and this bootstraping procedure,
please contact:

     Mr. Ming-Chin Lu
     Department of Electrical Engineering
     National Taiwan University
     Taipei, Taiwan, 10764
     R.O.C

Notice that: If you want to use the APS system, after
this bootstraping procedure and re-LOGIN, you must run
the command procedure named "APS$LIBRARY:INSTALL.COM".
Of course, you can put it in the "LOGIN.COM" file like
this:

  "$ @(your aps$library directory)install"
```

This installation procedure takes about 8 minutes (with CPU time about 4 minutes and 34 seconed). And finally, a disk quota of 455 blocks is taken. You can start developing your application programs by giving the

following command:

$APS

APPENDIX B

AN EXAMPLE ON INTERACTIVE SEQUENCE

```
VDL> Begin Cai_Course
VDL> op text at 1 20 with 40 1 as Title
VDL> op graph at 5 5 with 20 10 as Snoopy
VDL> op text at 5 35 with 40 10 as Head
VDL> op vform at 18 5 with 10 1 as No
VDL> op vform at 18 65 with 10 1 as Yes
VDL> set total F1
VDL> set procedure No
_Procedure: stop
VDL> set procedure Yes
_Procedure: assert(error(0))
VDL> ext Yes
VDL>
VDL> op text at 1 1 with 45 7 as Prob
VDL> op text at 11 1 with 45 6 as Ex
VDL> op vform at 1 65 with 10 1 as Exit
VDL> op vform at 16 65 with 10 1 as Ready
VDL> set total F2
VDL> set procedure Exit
_Procedure: stop
VDL> ext Ready
VDL> op graph at 1 1 with 30 10 as Fig1
VDL> op text at 14 1 with 30 5 as Desc
VDL> op text at 1 35 with 40 14 as Table
VDL> op vform at 18 35 with 10 1 as Yes
VDL> op vform at 18 50 with 10 1 as No
VDL> op vform at 18 65 with 10 1 as Review
VDL> set total F3
VDL> set next Review goto F2
VDL> ext Yes
VDL>
VDL> op graph at 1 1 with 30 10 as Fig2
VDL> op text at 1 40 with 35 15 as Desc
VDL> op vform at 15 1 with 10 1 as Yes
VDL> op vform at 15 21 with 10 1 as No
VDL> set total F4
VDL> set next No goto F5
VDL> set next Yes goto F3
VDL> ret
VDL>
VDL> ext No
VDL> op text at 5 5 with 65 4 as Query
VDL> op vform at 15 5 with 10 1 as Yes
```

```
VDL> op vform at 15 60 with 10  1 as No
VDL> set total F5
VDL> set next No goto F3
VDL> ext Yes
VDL>
VDL> op text at 2  2 with 40  10 as Quiz
VDL> op graph at 2  48 with 30  10 as Fig
VDL> op vform at 16  2 with 12  1 as 1
VDL> op vform at 16  18 with 12  1 as 2
VDL> op vform at 16  34 with 12  1 as 3
VDL> op vform at 16   50 with 12  1 as 4
VDL> op vform at 16  66 with 12  1 as 5
VDL> set total F6
VDL> ext 1
VDL>
VDL> op text at 5  10 with 60  10 as Desc
VDL> op vform at 18  60 with 10  1 as Ready
VDL> set total F7
VDL> set next Ready goto F6
VDL> ret
VDL>
VDL> copy 1 to  2  4  5
VDL> ext 3
VDL> op text at 2  20 with 35  2 as Good
VDL> op graph at 8  10 with 30  10 as Snoopy
VDL> op vform at 8  55 with 10  1 as Retry
VDL> op vform at 17  55 with 10  1 as ByeBye
VDL> set total F8
VDL> set procedure ByeBye
_Procedure: stop
VDL> set next Retry goto F1
VDL> home
VDL>
VDL> ed title
VDL$ED> fill
       Linear Programming
^Z
VDL$ED> ex
VDL> ed no
VDL$ED> al
   No
VDL$ED> ex
VDL> ed yes
VDL$ED> al
  Yes
VDL$ED> ex
VDL> ed head
VDL$ED> fill

      The    following    topic    is    the

Graphical Solution of Linear Programs
```

```
in two variables.


   Do you want to try it ?
^Z
VDL$ED> ex
VDL> ed snoopy
VDL$ED> fill
snoopy.dat
^Z
VDL$ED> ex
VDL> ext yes
VDL>
VDL> ed prob
VDL$ED> fill
Problem Statement:

   Given  a  close region R enclosed by a

set of lines, how to find the maximum value

of a function F(x,y) ?
^Z
VDL$ED> ex
VDL> ed ex
VDL$ED> fill
Example:

   Suppose x >= 0, y >= 0, y <= 100,
         x + y <= 150, x - y <= 90.

   Find the maximum value of F(x,y) = 2x+y
^Z
VDL$ED> ex
VDL> ed exit
VDL$ED> al
  Exit
VDL$ED> ex
VDL> ed ready
VDL$ED> al
  Ready
VDL$ED> ex
VDL> ext ready
VDL>
VDL> ed yes
VDL$ED> al
  Yes
VDL$ED> action
_Action: cursor(22,1),write('(x,y) = '),read(X),read(Y),assert(value(X,Y))
VDL$ED> ex
VDL> ed no
```

```
VDL$ED> al
  No
VDL$ED> ex
VDL> ed review
VDL$ED> al
 Review
VDL$ED> ex
VDL> ed desc
VDL$ED> fill
Notice that:
  270 in the table is maximum
. Any other points in R is
less than 270.
  Have a try to verify ?
^Z
VDL$ED> ex
VDL> ed table
VDL$ED> fill
For F(x,y) = 2x + y, randomly generates
a set of points (x,y) in R as follows:
```

x	y	F(x,y)
0	76	76
20	3	43
40	11	91
60	10	130
80	22	182
100	29	229
120	30	270

```
^Z
VDL$ED> ex
VDL> ext yes
VDL>
VDL> ed yes
VDL$ED> al
 Yes
VDL$ED> ex
VDL> ed no
VDL$ED> al
 No
VDL$ED> ex
VDL> ed desc
VDL$ED> fill
     >>> F(x,y) = 2x + y <<<

At x =    ,   y =

Then,

  F(x,y) =
```

```
so
   270 at vertix (120,30)

is MAXIMUM.

Another try ?
^Z
VDL$ED> action
_Action: value(X,Y),cursor(5,49),write(X),cursor(5,63),write(Y)
VDL$ED> action
_The action to take is: value(X,Y),cursor(5,49),write(X),cursor(5,63),write(Y)
_Do you want to A)dd, C)hange, or unchange(<CR>) ? A
_Action: retract(value(X,Y)),cursor(9,56), Z is 2*X+Y, write(Z)
VDL$ED> ex
VDL> ret
VDL>
VDL> ext no
VDL>
VDL> ed yes
VDL$ED> al
  Yes
VDL$ED> ex
VDL> ed no
VDL$ED> al
  No
VDL$ED> ex
VDL> ed query
VDL$ED> fill
         In the following, we will give you a quiz.


           A R E   Y O U   R E A D Y ?
^Z
VDL$ED> ex
VDL> ext yes
VDL>
VDL> ed 1
VDL$ED> al
(1) (0,0)
VDL$ED> ex
VDL> ed 2
VDL$ED> al
(2) (0,60)
VDL$ED> ex
VDL> ed 3
VDL$ED> al
(3) (70,45)
VDL$ED> ex
VDL> ed 4
VDL$ED> al
(4) (30,90)
```

```
VDL$ED> ex
VDL> ed 5
VDL$ED> al
(5) (120,0)
VDL$ED> ex
VDL> ed quiz
VDL$ED> fill
Answer the following question:

Suppose x >= 0, y >= 0,
      x + y <= 120, x - y >= 60.

For ax + by = F(x,y),
   a,b are any real numbers.

Which of the following can NEVER be the
maximum?
^Z
VDL$ED> ex
VDL> ext 1
VDL>
VDL> ed desc
VDL$ED> fill

   SORRY, it is a wrong answer.


   Because what you chose  is a point WITHIN R, and


   Any points inside region R CAN NOT be the maximum.
^Z
VDL$ED> ex
VDL> ed ready
VDL$ED> al
 Ready
VDL$ED> action
_Action: retract(error(X)), Y is X+1, assert(error(Y)), !, Y > 1,  sys$goto(0)
VDL$ED> ex
VDL> ret
VDL>
VDL> ext 3
VDL>
VDL> ed retry
VDL$ED> al
 Retry
VDL$ED> ex
VDL> ed byebye
VDL$ED> al
 ByeBye
VDL$ED> ex
VDL> ed good
```

```
VDL$ED> fill
        Good !
     You got it Right.
^Z
VDL$ED> set  reverse
VDL$ED> ex
VDL> ho
VDL>
VDL> ext yes
VDL>
VDL> ext ready
VDL>
VDL> ed fig1
VDL$ED> fill
Fig1.dat
^Z
VDL$ED> ex
VDL> ext yes
VDL>
VDL> ed fig2
VDL$ED> fill
Fig2.dat
^Z
VDL$ED> ex
VDL> ret
VDL>
VDL> ext no
VDL>
VDL> ext yes
VDL>
VDL> ed fig
VDL$ED> fill
Fig3.dat

^Z
VDL$ED> ex
VDL> ext 3
VDL>
VDL> ed snoopy
VDL$ED> fill
snoopy2.dat
^Z
VDL$ED> ex
VDL> exit
```

REFERENCES

[1] Anderson, J. R., Boyle, C. F. and Yost, G., "The Geometry Tutor," IJCAI, LosAngles, U.S.A.,pp. 1-7 (1985).

[2] Campbell, J. A., ed., *Implementations of Prolog*, Ellis Horwood Series in Artificial Intelligence (1984).

[3] Chang, S. K. and Charisse, O., "The Interpretation and Construction of Icons for Man-Machine Interaction in an Image Information System," Proceedings of IEEE Workshop on Language for Automation, New Orleans (Nov. 1984).

[4] Chen, J. C., "A Knowledge-Based Environment on Virtual Workstation," M. S. Thesis, Graduate Institute of NTUEE, Taipei, Taiwan, R. O. C. (June 1985).

[5] Cheng, K. Y., Hsu, C. C., Lin, I. P., Lu, M. C., and Hwu, M. S., "VIPS: A Visual Programming Synthesizer," Second IEEE Computer Society Workshop on Visual Language, Dallas, Texas, U. S. A. (June 1986).

[6] Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, 2nd ed., Springer-Verlag, Berlin, 1984.

[7] Ellis, C. A. and Nutt, G. J., "Office Information Systems and Computer Science," Computing Surveys, Vol. 12, No. 1, pp. 27-60 (March 1980).

[8] Fu, K. S. and Booth, T. L., "Grammatical Inference: Introduction and Survey -- Part I," IEEE Trans. on System, Man, and Cybernetics, Vol. SMC-5, No. 1, pp. 95-111 (1975).

[9] Fu, K. S., "An Introduction to Formal Language," Chap. 2. in *Syntatic Pattern Recognition and Application*, Prentice-Hall, Englewood Cliffs, N. J., 1982.

[10] Gray, M. D., *Logic, Algebra and Database*, Halsted Press, 1984.

[11] Harrison, M. A., *Introduction to Formal Language Theory,* Addison-Wesley (1978).

[12] Hopcroft, J. E. and Ullman, J. D., *Introduction to Automata Theory, Language, and Computation*, Addison-Wesley, 1979.

[13] Hwu, M. S., "A Manipulation Language for Visual Programmings," M. S. Thesis, Graduate Institute of NTUCSIE, Taipei, Taiwan, R. O. C. (June 1986).

[14] Jacob, Robert J. K., "A State Transition Diagram Language for Visual Programming," IEEE Computer, pp. 51-59 (1985).

[15] Kearsley, G., "Authoring Systems in Computer Based Education," Comm. of the ACM, Vol. 25, No. 7, pp. 429-437 (1982).

[16] King, K. J. and Maryanski, F. J., "Information Management trends in Office Automation," Proceedings of the IEEE, Vol. 71, No. 4, pp. 519-528 (1983).

[17] Kitagawa, H., Gotoh, M., Misaki, S., and Azuma, M., "Form Document Management System SPECDOQ - Its Architecture and Implementation," Second ACM-SIGOA Conference on Office Information System, Vol. 5, No. 1-2, pp. 132-142 (June 1984).

[18] Kowalski, R. A., "Predicate Logic as Programming Language," Proc. of IFIP 74, Stockholm (1974).

[19] Moriconi, M. and Hare, D. F., "Visualizing Program Designs Through PegSys," IEEE Computer, Vol. 18, No. 8, pp. 72-85 (1985).

[20] Pereira, F., ed., *CProlog User's Manual,* Version 1.1 (Dec. 1982).

[21] Reiser, B. J., Anderson, J. R. and Farrel, R. G., "Dynamic Student Modelling in an Intelligent Tutor for LISP Programming," IJCAI, LosAngles, U. S. A., pp. 8-14 (1985).