

COMPILER ISSUES RELATED TO ICONIC  
VISUAL LANGUAGES

by

FANG ZHANG, B.S.M.E.

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty  
of Texas Tech University in  
Partial Fulfillment of  
the Requirements for  
the Degree of

MASTER OF SCIENCE

Approved

Accepted

---

Dean of the Graduate School

December, 1997

## ACKNOWLEDGMENTS

3  
2  
I would like to thank my advisor, Dr. Donald J. Bagert, thank for his patience, encouragement, and valuable advice. Appreciation is also expressed to Dr. Gopal Lakhani for his guidance and serving on the graduate committee. This work was partially supported by the Texas Higher Education Coordinating Board *Advance Research Program*, under grant 003644-084 to Texas Tech University. I would like to express my gratitude for the financial support.

I thank the following people in my research group for their help and friendship: Ms. Flora Namuswa, Ms. Yanwei Shi, Mr. Taoan Ge and Mr. Devon Peasley.

To my parents and my grandma, without their love, I could not have accomplished this.

## TABLE OF CONTENTS

ACKNOWLEDGMENTS .....	ii
ABSTRACT .....	vi
LIST OF TABLES .....	vii
LIST OF FIGURES .....	viii
CHAPTER	
I. INTRODUCTION .....	1
1.1 Overview of the Problem .....	1
1.2 Visual Programming Languages.....	2
II. LITERATURE SURVEY .....	7
2.1 Previous Research .....	7
2.2 Current Research .....	9
III. RESEARCH OBJECTIVES.....	11
3.1 Prototype of Iconic BNF .....	12
3.2 Visual Compiler .....	12
3.3 Visual Environment .....	13
IV. VISUAL TOKEN SYSTEM .....	14
4.1 Terminals .....	14
4.1.1 Icons .....	14
4.1.2 Connections .....	19
4.1.3 Other Types of Terminals .....	21

4.2 Non-terminals .....	21
4.3 Productions .....	23
V. THE PARSER AND SCANNING SOLUTION.....	31
5.1 The Parser.....	31
5.2 Go Through Scheme .....	33
5.3 An Example .....	36
VI. ERROR HANDLING .....	42
6.1 Error Types .....	42
6.2 Dealing With Errors in Different Levels .....	43
VI. INTERFACE AND DATA STRUCTURE .....	50
7.1 Overview .....	50
7.2 Classes.....	53
7.3 Functions.....	54
7.4 Variables.....	55
VII. CONCLUSION AND FUTURE RESEARCH .....	57
8.1 Conclusions.....	57
8.2 Future Research .....	58
REFERENCES .....	61
APPENDICES	
A. THEORIES AND ALGORITHMS ABOUT LR PARSER .....	63
B. THEORIES AND ALGORITHMS ABOUT DR PARSER .....	67

C. THE TRANSITION DIAGRAM AND SLR PARSING TABLE CONSTRUCTED IN THIS RESEARCH .....	69
---	----

## ABSTRACT

This research studies the visualization of traditional programming languages. The major problem is how to represent traditional one-dimensional language in a two-dimensional paradigm. The focus of this research is compiler issues, traditional compiler theories and algorithms, together with new concepts that are applied in the research. A multi-paradigm programming language environment was created to experiment the theoretical solution. The current target language is C++, but the environment configuration can be applied to other traditional programming languages. In this particular environment, specific attention has been given to the user interface, data structures, parsing, and error handling. This research uses a new approach, and provides some possible solutions in the Visual Programming Language field.

## LIST OF TABLES

5.1	Database .....	37
6.1	Relations Between Connections and Icons .....	45
C.1	The SLR Parsing Table Based on the Canonical Collection and the Transition Diagram.....	76

## LIST OF FIGURES

4.1	Icon .....	15
4.2	Predicate .....	16
4.3	Statement .....	16
4.4	Test .....	17
4.5	Case .....	18
4.6	Connector .....	18
4.7	Start .....	19
4.8	End .....	19
4.9	Arrows.....	20
4.10	Corners.....	20
4.11	Bend.....	21
4.12	Nonterminals .....	22
4.13	Production 1 .....	23
4.14	Production 2 .....	24
4.15	Production 3 .....	24
4.16	Production 4 .....	25
4.17	Production 5 .....	25
4.18	Production 6 .....	26
4.19	Production 7 .....	27
4.20	Production 8 .....	27



4.21	Production 9 .....	28
4.22	Production 10 .....	29
4.23	Production 11 .....	30
5.1	Two Different Translation Models .....	33
5.2	Program Flows .....	34
5.3	Program Diagram .....	36
5.4	The Go-through Process .....	38
5.5	The Intermediate Code .....	40
5.6	Parse Tree .....	40
6.1	Working Space .....	44
6.2	Part of the Toolbar .....	45
7.1	Structure of the IceC Environment .....	52
7.2	Class Definition Interfaces .....	53
7.3	Function Definition Interfaces .....	54
7.4	Parameter Definition Interface .....	55
7.5	Variable Definition Interfaces .....	56
C.1	The Transition Diagram Based on the Canonical Collection.....	74

# CHAPTER I

## INTRODUCTION

### 1.1 Overview of the problem

In this era of computer, a new carrier of information (electronic media) is replacing the old one (paper). Pictures and written words are not essentially different in this new media; all information, no matter whether it is in picture or in words, is stored in the same manner (binary digits). A person, through the use of a keyboard or mouse, can copy a picture, and it is guaranteed to be exactly the same copy. During the evolution of computers, people have been rediscovering the power of pictorial expression. In old-fashioned operating systems such as DOS and UNIX, commands needed to be typed in line by line, similar to traditional communication methods. When the user is working in front of a computer in such cases, he/she is explicitly aware that he/she is using a written language. This state of affairs, which dominated computer science for over a quarter century, started to come to an end in 1975 with the publication of David Canfield Smith's landmark dissertation *Pygmalion: A Creative Programming Environment* [1, 2]. Smith's research heralded a new era in programming, and made it possible to utilize the two-dimensional CRT screen as more than a mere back drop for linear text string wrap-around. To meet the challenge of this new era, many facilities have become popular, such as icons, pointing devices, word processors, query facilities, and application generators. UNIX has X-windows, and DOS has Microsoft Windows. This new environment opens a whole new world for end users.

Microsoft Windows has become very popular to common users all over the world. Microsoft Windows and other similar applications built these systems and created new rules. Such an environment is like international pictorial language, which has its own standards and dialects, and is becoming the mother tongue of more and more users.

Obviously, the increased popularity of computer is strongly related to this dramatic change in the user interface. Why is this? The human brain is divided into two hemispheres; linguistic ability is dependent primarily on the left hemisphere, while the perception of melodies and nonverbal visual patterns is largely a function of the right hemisphere. The new technology of visualization in the computer field explores the capability of the right half of the brain, and therefore made computer more acceptable for common people. The majority of potential computer users today are not computer professionals, and just need to use computer as tools to get their work done. A visual environment hides all the complicated details that ordinary users can't understand and don't need, while displaying pictures which they are familiar with or can understand easily. So it has gradually become a major weapon in increasing the popularity of computers.

## 1.2 Visual Programming Language

The term 'Visualization' in the computer field is actually an ambiguous concept. This thesis research looks visualization as its relates to programming languages. This section will present some terminology related to previous research in visual programming languages.

Visual programming is the use of visual expressions (such as graphics, drawings, or icons) in the process of programming; Visualization is the use of visual representations (such as graphics, images, or animation sequences) to illustrate programs, data, the structure of a complex system, or the dynamic behavior of a complex system. A visual language is the systematic use of visual expressions to convey meaning; a language supporting visualization is also a visual language although languages such as Microsoft Visual C++ and Visual Basic only use visualization in the user interface. Visual programming systems support both visual programming and visualization.

Visual programming languages can appear in many different forms: charts and diagrams, iconic or pictorial languages and table or form-based languages. The most popular of these is the iconic language. In an iconic paradigm, a visual language models an icon system, and a program consists of a spatial arrangement of pictorial symbols (elementary icons). The spatial arrangement is the two-dimensional counterpart of standard sequentialization in the construction of programs in traditional programming languages. This research studied the visualization of traditional programming languages, the research approach is iconic language. Although 'Visual Programming Languages' is a frequently used term in this research area, people sometimes confuse it with 'Visual Programming Environment' because of the popularity of Visual Basic and Visual C++. So to be more accurate, the subject of this research will be called iconic programming languages.

As mentioned above, research in this field is classified into two areas. One is Visual Programming Languages (VPL), which are qualified by programming languages using visual expressions such as diagrams, free-hand sketches, icons, or even graphical manipulations. The other is Visual Programming Environments (VPE), which provides visual method of working with a programming language, whether the language is itself visual or textual. The terminology 'visual programming', 'visual programming system' and 'visual language' are frequently used in VPL, while VPE emphasizes 'visualization' and 'visual languages' more.

The earliest work in this field concentrated on VPL, experimenting with visual approaches to traditional programming languages. These systems were exciting and intuitive when demonstrated with "toy" programs, but ran into difficulties when extended to programs of realistic size. These problems soon led to disenchantment with visual programming, causing many to believe that it was an academic exercise inherently unsuited to "real" work.

So researchers began to concentrate on VPEs. A lot of significant work has been done in this field. Successful commercial VPEs include Microsoft's Visual Basic, Visual C++ and ParcPlace Systems' VisualWorks. At the same time, other researchers are working on a different method called the domain-specific approach, in which the visual expressions and terminology are used to reflect the needs, problem-solving diagrams, and vocabulary specific to that domain. There is also a lot of interesting work done in this area.

The original challenge--how to devise VPLs with enough power and generality to address an ever-expanding variety of programming problems--is still an active research subject. This research focused on this direction and tried to find a possible solution on the difficulties. This research used traditional languages, more specifically, C++, while future work will apply this research to other languages such as Ada95 and Java. In the context of this thesis, the 'small-grain' solution means visualization down to the smallest grammar element--identifiers or terminals and the relations between them, while the 'large-grain' solution means the visualization higher than this level. One of the lessons learned from early 'small-grain' visual programming environments, such as PICT [3], is that it is not a good idea to try to visualize everything. Some things, like mathematical formulas and numerical algorithms, are handled better with text. So 'large-grain' will be a reasonable direction for the solution. When the term 'large-grain' is used, the picture of a program is like a flowchart, where just program flow is seen, without any details of expression.

This research proposed an iconic BNF which generalizes the 'large-grain' expressions of traditional languages. (BNF stands for Backus-Naur Form, which are used to defined context-free grammars). The efficiency and extendibility of this system was then tested by using it to generate a compiler which is used in a visual programming environment. This research is part of the STRIDES group research project [4], which is a variation and extension of the previous research product BACCII++ ( © and <sup>TM</sup> 1992-1995 Ben A. Calloni ) [5]. Studying the interaction of the

visual compiler and the 'small-grain' expression solution which was prototyped in BACCII++ is also a subject of this thesis.

A brief synopsis of the coming chapters is given in the following. Chapter II introduces some related previous research; Positional Grammar and DR parser are emphasized. In Chapter III, the major objectives of this research are presented. Chapter IV is about the iconic token system: terminal, nonterminal and productions; the theory and algorithm used to construct the iconic compiler, and the difference between the iconic compilers and traditional compilers are presented in Chapter V. Chapter VI covers the error handling during the compilation process in the environment (IceC) constructed in this research. Chapter VII introduces IceC in two aspects: user interface and data structure. Finally, Chapter VIII gives conclusions and future research directions.

## CHAPTER II

### LITERATURE SURVEY

#### 2.1 Previous Research

Most recent articles concern domain-specific problems, but few focus on programming languages. Even in a pure domain-specific problem, the languages issues, like the logical expression which are in the 'large-grain' level, cannot be avoided.

Several researchers have tried to construct the token and translation system for visual programming; E. J. Golint [6], P. D. Vigna [7], J. Feder [8], F. Ferucci [9], R. Helm [10], K. Wittenburg [11], C. Crimi [12] are among them. However, most of these projects never advanced beyond the theoretical stage, without having built a mature system to realize their theories. However, these works demonstrate the possibility of constructing an iconic BNF system, without building such a system to cover all traditional languages. Some common themes could be found among their works. The main challenge of iconic BNF is how to scan and parse tokens in a two-dimensional paradigm instead of the traditional one-dimensional paradigm.

Khoros [13, 14] was an ideal model for this research to build upon. Khoros is a large, mature system which uses a data flow visual language [15]. Although the Khoros is now considered as a general-purpose visual language, and useful for many different applications, originally it was developed for image processing; this is why Khoros is considered to be a data flow visual language.



Gennaro Costagliola's prototype VLCC [16] presented some interesting possibilities. The focus of VLCC is more general, covering both domain-specific visual languages and VPL for traditional languages. Costagliola's visualization of traditional languages is done on a 'large-grain' level, and leaves the 'small-grain' detail to the user. The problem of traditional programming languages' visualization is more or less unsolved here, because it does not convince the reader that a user not familiar with LEX and YACC could handle grammar definition on the 'small-grain' level very well. But the theory they presented provides the methodology for a general approach to research on visual programming languages.

There are two major parts of Costagliola's model: Positional Grammars and DR Parsers [17, 18, 19]. Positional Grammars generate languages in the two-dimensional paradigm.

A Positional Grammar  $G$  is a 6-tuple  $(N, T, S, P, POS, SP)$ .  $N$ ,  $T$ ,  $S$  stand for Non-terminals, Terminals, and Starting Symbol, respectively, the same as for a traditional BNF.  $POS$  is a set of positions in the space or positional operators.  $SP$  is the starting position and it is always the first element of the right hand side of the productions whose left hand side is the starting symbol  $S$ . Each production in  $P$  has the following form:  $A = x_1 q_1 x_2 q_2 \dots x_m$  where each  $x_i$  is in  $N \cup T$  and each  $q$  is in  $POS$ . Each  $q_i$  gives the relative position of  $x_{i+1}$  with respect to  $x_i$ .

DR Parsers are used to drive the scanning of particular languages or pictures. They are a generalization of the LR parser (D stands for driven scanning of the input). The differences between an LR parser and a DR parser are mainly in the parsing table.

in the access to the input, and in the parsing program. Besides the "action" and "goto" columns, the DR parsing table contains an additional column called "position": a position for the next possible input is associated to each state of the push down automation. The input is no longer accessed sequentially, but randomly.

## 2.2 Current Research

Costagliola's theory is a major factor in this research, which is a part of the ongoing STRIDES project. STRIDES stands for Software Through Iconic Design. It uses visual programming languages, and can also be viewed as an iconic environment for traditional programming languages. Because the theories related to VLCC have not been used in a complex, commercial-oriented system, STRIDES provides an opportunity to further research in this area.

In STRIDES, a Positional Grammar is being used in a system prototype to see if it could be used as a theoretical base for a visual programming system, or more specifically, as a context-free language for two-dimensional visual programming environment, to see if it is a robust candidate for the iconic BNF of traditional programming languages in two-dimensional paradigm, and also to see what kind of additional algorithms and modification are needed for the theories during the implementation.

The key point is whether a DR parser is needed in the particular environment created for this research. The difficulty of a parser for iconic visual languages is the scanning of two-dimensional structure which has never been encountered in a

traditional parser. The difference between a DR parser and a LR parser is that a DR parser needs to use spatial relations to find out which is the next item, which means some additional information is needed in the parsing table. Considering the needed generality for the theory, the DR parser model makes sense. On the other hand, iconic programming languages is still a premature and active research field, and many different theories and systems are being created and tried. Unlike a traditional one-dimensional paradigm which has a uniform representation --either linear text or string, the significant diversity of environments makes scanners very unlikely to be similar, because they need to deal with the environment-specific data structure.

## CHAPTER III

### RESEARCH OBJECTIVES

The approach used in this research is to divide the function of DR parsers into two parts. The first part is an environment embedded scanner whose job is to translate two-dimensional representations into a one-dimensional format. The second part is a traditional LR parser which takes the text generated by the scanner and translates it into the target language. The object of both parts are 'large-grain' grammar items. The research objectives consist of three major sections:

1. Create a prototype iconic BNF for traditional languages on the 'large-grain' level.
2. Use this BNF to define a iconic programming language and then construct a iconic compiler (including a system-embedded scanner and a LR parser) for it.
3. Build an iconic environment for the iconic programming language (including data structure and user interface) which interfaces with the compiler.

It should be emphasized that when a traditional programming language is enhanced to become an iconic programming language, there are now both language issues and environment issues to consider. That is part of the difficulties of defining an iconic programming language. When an iconic programming language is used, it is

difficult to ignore the concrete environment where it exists. Somehow, the environment becomes an undivided part of this language.

### 3.1 Prototype of Iconic BNF

This research primarily concentrates on the prototype of an iconic BNF. As can be seen in the previous discussion, there are many different kinds of representations for visual languages. The difficulty, however, is to build a general representation which can generalize the visualization of any kind of traditional text-based language while also being practical for software development environment. This research also concentrates on the Positional Grammar which presented in [16], studying its potential possibility as a candidate of future iconic BNF and implementing it in a wider range than [16] did for traditional programming languages.

### 3.2 Visual Compiler

The term 'Visual Compiler' is actually not a very accurate name for this part of the whole system. This compiler is generated by LEX and YACC, and the resulting object will still be text-based, so basically it is not different from traditional compiler. The compiler can not accomplish the translation job by itself; however, the whole translation process needs the cooperation of two different elements: a visual representation interpreter and a LR parser. The interpreter translates elements into one-dimensional text representation, and then the LR parser takes the text input

generated by the translator and generates code in one specific traditional language (currently C++).

### 3.3 Visual Environment

Unlike in traditional programming languages, the environment plays an important role in an iconic programming language. Iconic programming language research, at least in the context of this thesis, is not about creating a new language or making a language itself more powerful, but is about expressing a traditional language in a new way in which the potential power of a tool could be exploited more easily. Especially in this research, the environment also takes part of compiler's responsibility besides building UI and data structure. The data structure is built by Microsoft ACCESS 2.0 and the UI and the visual representation interpreter are created in Microsoft Visual Basic 4.0.

## CHAPTER IV

### VISUAL TOKEN SYSTEM

#### 4.1 Terminal Symbols

Terminals are not just those icons constructing the diagram, it also includes connections and some textual tokens which are used to represent some special meaning in the course of scanning. However, the icons are the major part of the terminals, and the other parts are merely complementary of it. This is because a compromise has been made between the concrete environment and the theory. In other words, unlike in [17], this thesis does not treat positional information or connection as another dimension of grammar definition. They are together with icons to construct the token collection (terminal symbols). After the translation from two-dimensional visual representation to one-dimensional textual representation, the terminals of the grammar may not correspond to an existed icon or connection, but could instead represent a logical relationship between icons or connections. Therefore, the terminals are discussed in three categories below: icons, connections and other type of terminals.

##### 4.1.1 Icons

Icons are viewed as hot spots of the program flow; that is they are the places where the possibilities for branching in and out are provided. In program flow, icons and the relations between them are dynamic. An icon receives program flow from

the previous icon, and passes it on to the next icon. The attributes of an icon at the diagram level are the attaching points, which provide the mechanism that scanning algorithm work on. The basic model of an icon is shown as Fig 4.1. It is implemented here as a 480×480 image, with a body and a maximum of 4 attaching points. The shape of the body doesn't have any syntactical meaning; theoretically, there could be any type of shape, but in implementation, shapes frequently used in flowchart-like diagrams are preferred. The four attaching points are defined as North, East, South and West. Generally, North always receives flow, the South always passes flow, the East and the West can either receive and pass flow, depending to which icon they are associated. Some icons have corresponding dialog boxes, which have pop up windows showing up when user double-clicks on these icons. In the dialog boxes, user can fill in the 'small-grain' details of program.

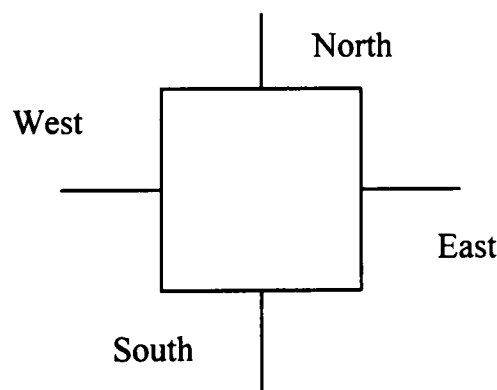


Fig 4.1 Icon

#### 4.1.1.1 Predicate

Fig. 4.2 is a Predicate icon. In the translation from visual representation to textual representation, a predicate could be translated into either IF or SWITCH, depending on how user defines this icon in its corresponding dialog box with



respect to it. The North receives flow while the East and West pass flow. The numbers on attaching points represents the priority in the scanning course or go-through scheme (see Chapter IV for details). The North will not allow fan-in and the West and East will not allow fan-out. In other words, the North can receive just one flow, while the East and West can pass just one flow in each case.

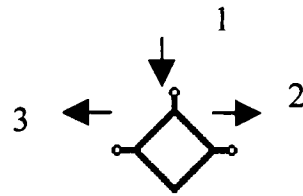


Fig. 4.2 Predicate

#### 4.1.1.2 Statement

Fig. 4.3 is a Statement icon. A statement icon will be translated by the interpreter as STAT. This icon has two attaching points: North and South, the North receives flow and the South passes flow. There is no fan-in for the North, nor fan-out for the South.

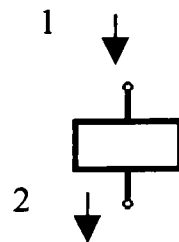


Fig. 4.3 Statement

#### 4.1.1.3 Test

Fig. 4.4 is a Test icon. A Test icon will be translated by the interpreter as TEST. In the dialog box with respect to this icon, user could define it as a loop (pretest or posttest). This icon has all four types of attaching points, and has no fan-in or fan-out for any of them. The North and the West receive flow while the South and the East pass flow.

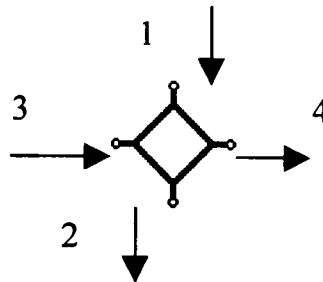


Fig. 4.4 Test

#### 4.1.1.4 Case

Fig. 4.5 is a Case icon. A Case icon will be translated by the interpreter as CASE. This icon has three attaching points: East, West and South; no fan-in or fan-out is allowed for any of them. In a diagram, the Case could appear in two connecting forms. The first form is receiving flow from the West and passing it to the South and East; The second is receiving flow from East and passing it to the South and West. It depends on how user defines the first one of the East and West, these two attaching points must in different situation, one catching flow and the other passing flow. Actually, there should be four situations for the Case, because for the one passing flow among the East and the West, it could be unconnected.

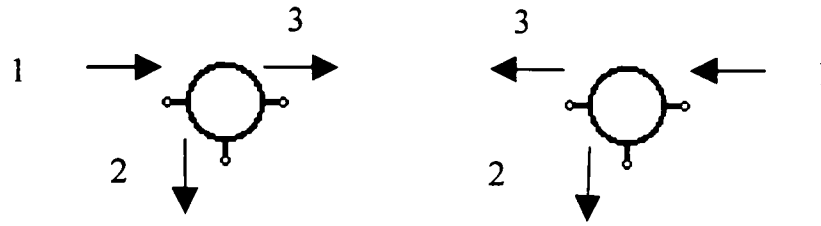


Fig. 4.5 Case

#### 4.1.1.5 Connector

Fig. 4.6 is a Connector icon. A Connector icon will be translated as CONET by the interpreter. It has two attaching points: the North and the South. No fan-out for the South, but fan-in is allowed for the North. It is the only icon allowing fan-in. Because it is a connector of program flow branches, its job is takes all branches back into the main stream. So for same reason , the Connector is just an icon functioning connection, and doesn't have any corresponding dialogue box.

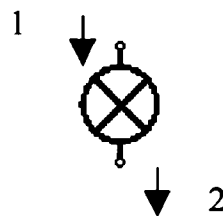


Fig. 4.6 Connector

#### 4.1.1.6 Start

Fig. 4.7 is a Start icon. A Start icon will be translated as START by interpreter. The Start just has one attaching point which passes flow from the South, no fan-out, no associated dialogue box, and indicates the starting point of a

function. A function only allows one Start icon to appear. at the beginning of the flowchart.



Fig. 4.7 Start

#### 4.1.1.7 End

Fig. 4.8 is an End icon. An End icon will be translated as END by the interpreter. The End just has one attaching point which catches flow from the North, no fan-in, no associated dialogue box, and indicates the end of a function. A function only allows one End icon to appear, at the end of the flowchart.



Fig. 4.8 End

### 4.1.2 Connections

#### 4.1.2.1 Arrow

The first kind of connection, arrow, as shown in Fig. 4.9, will be translated as A by the interpreter, including three different types: a, b and c. The A does not stand for any thing. It is only the textual notation of this kind of connection. The a is

from the East to the West; the b is from the West to the East; the c is from the South to the North.

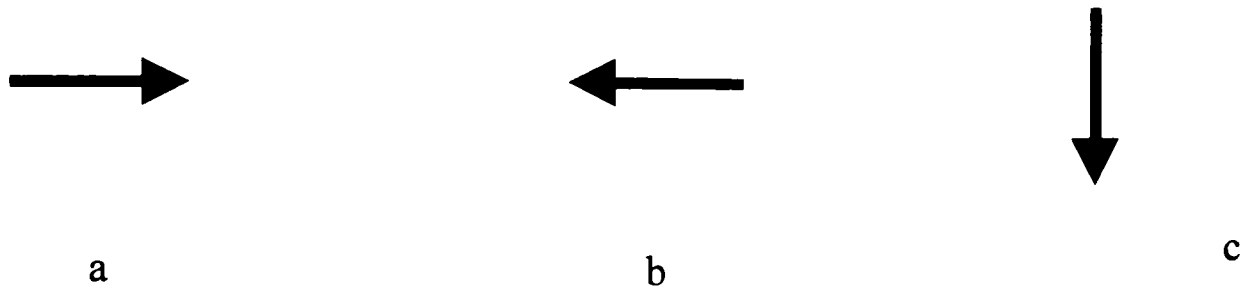


Fig. 4.9 Arrows

#### 4.1.2.2 Corner

The second kind of connection, corner, as shown in Fig. 4.10, will be translated as B by the interpreter, including two different types: a and b. The a is from the West to the North; the b is from the East to the North. The first and the second connection together will be parsed as non-terminal 'down' in the parsing tree because the orientation of these two kinds of connections is from up to down.



Fig 4.10 Corners

#### 4.1.2.3 Bend

The third kind of connection, bend, as shown in Fig. 4.11, will be translated as C by the interpreter. It is from the South to the West. This kind of connection is for all kinds of looping.

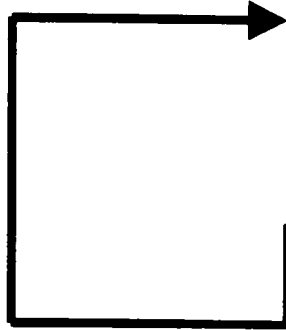


Fig. 4.11 Bend

#### 4.1.3 Other Types of Terminals

There are two other kinds of terminals: '\' and '|', they are translated as SLASH and DASH respectively. They are used to discriminate the difference between the IF\_THEN structure and the SWITCH structure, because they derive from same key icon -- Predicate. (Those icons with multi passes are key icons, they are the 'key' for the classification of the derived structure.)

#### 4.2 Nonterminals

There are eleven nonterminals. They are 'flowchart', 'compd', 'sent', 'loop', 'condition', 'single', 'multi', 'branch', 'casepart', 'onecase' and 'down', as shown in Fig. 4.12. The 'flowchart' is the start symbol of this grammar; all valid structures will use this symbol. In the notational convention of this thesis, all upper-case names are

terminals, while lower-case names are nonterminals. The nonterminals are represented as dash shapes except the 'down'. Some have connections on it, some don't. Basically those having connections are parts of a completed structure. (IF\_THEN, SWITCH, etc.). Dash attaching points on the same body are exclusive. Some nonterminals have more than one shade; it does not mean they are different non-terminals, it just means this symbol could appear in different shapes in a diagram, but these shapes will still be recognized as same item in the parsing process.

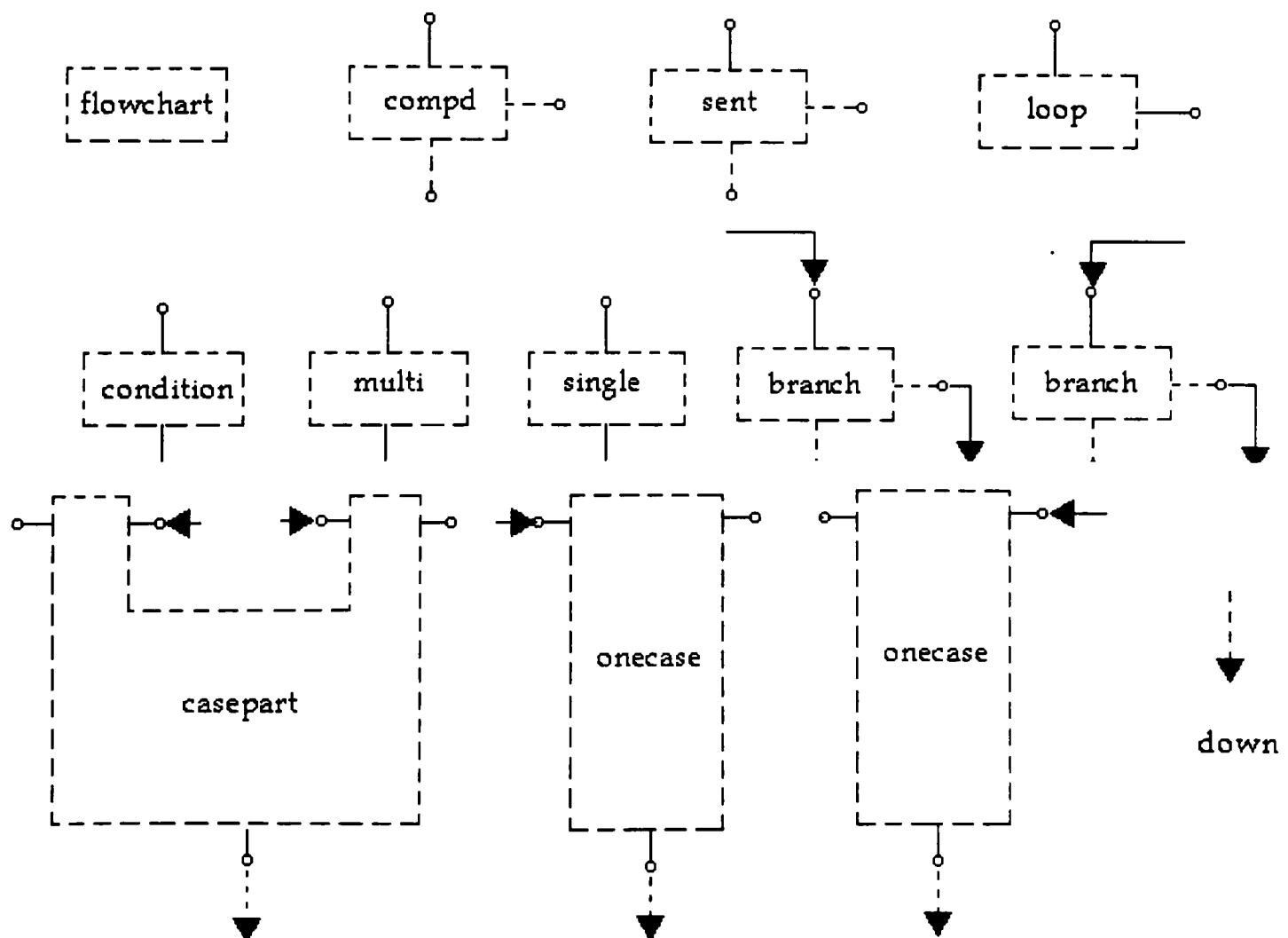


Fig. 4.12 Nonterminals

### 3.3 Productions

Since traditional LR parsers are used in this approach, the relation between the textual grammars and the diagrams are not simple one-to-one as in the DR parser. The picture showing with each production is more like an explanation of the rule.

1. flowchart = START A compd down END

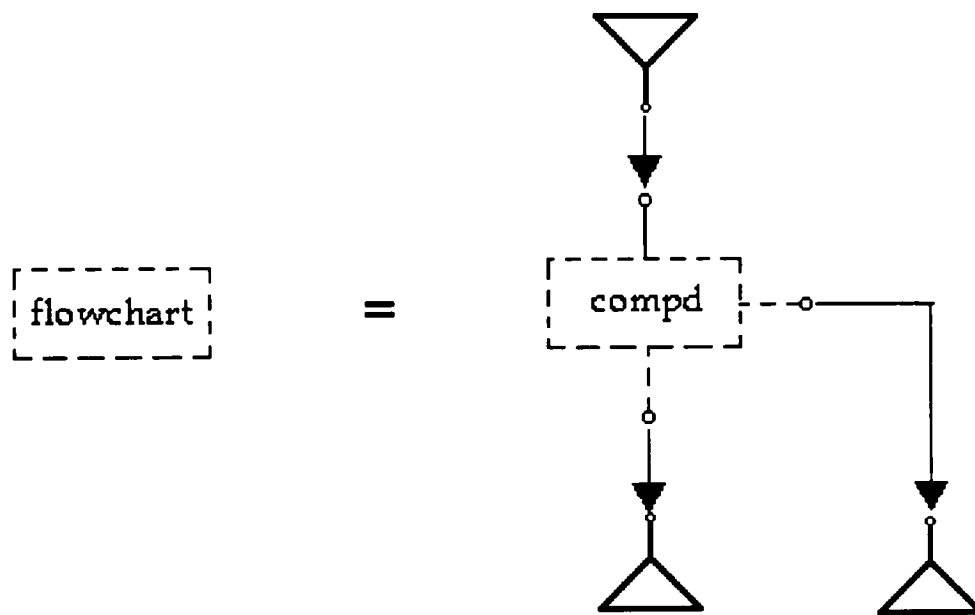


Fig. 4.13 Production 1

2. compd = compd down sent | sent



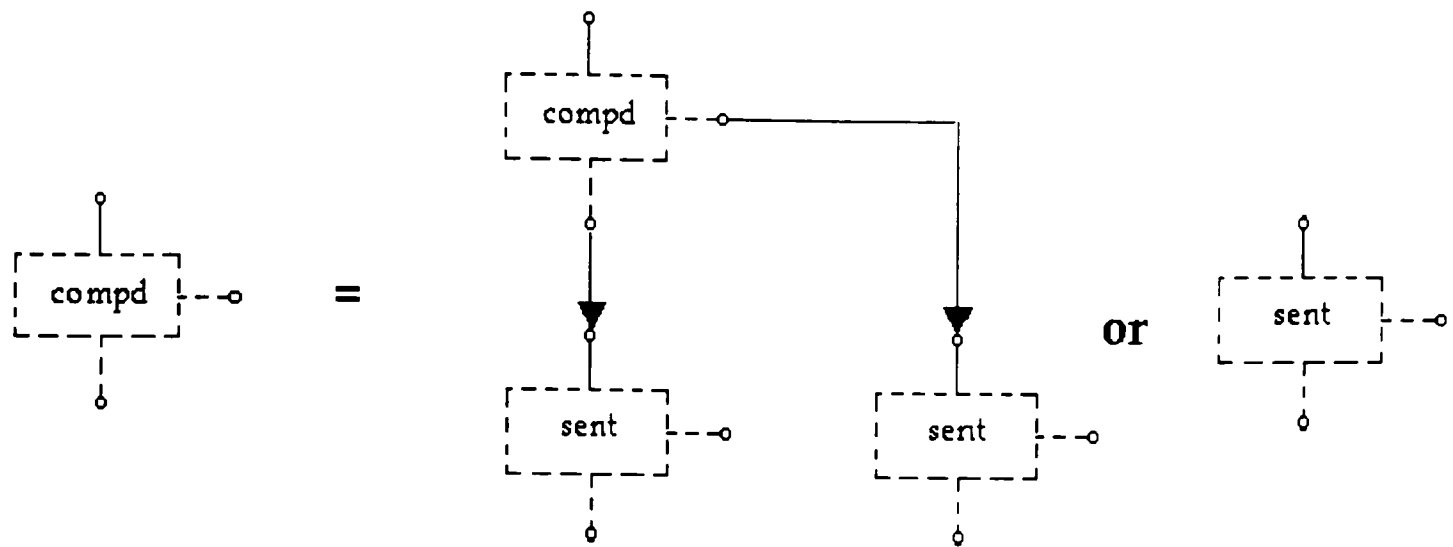


Fig. 4.14 Production 2

3. sent = condition | loop | STATE

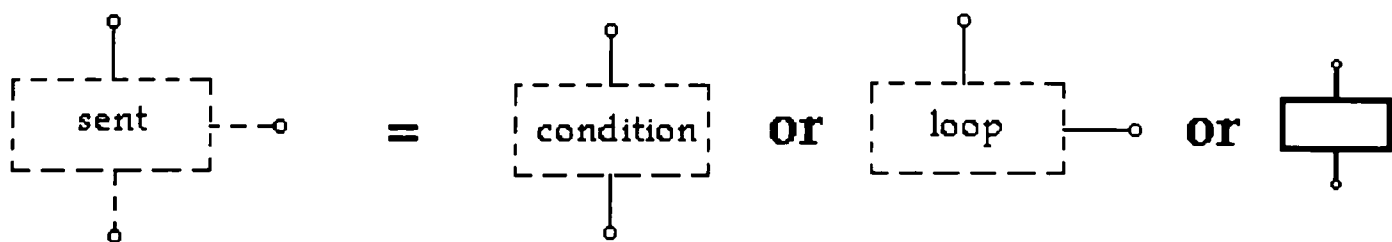


Fig. 4.15 Production 3

4. loop = TEST A compd C

In this production, theoretically from the East the program flow could go to the West of TEST. But practically it is impossible, because the connection C can not connect the East and the West.

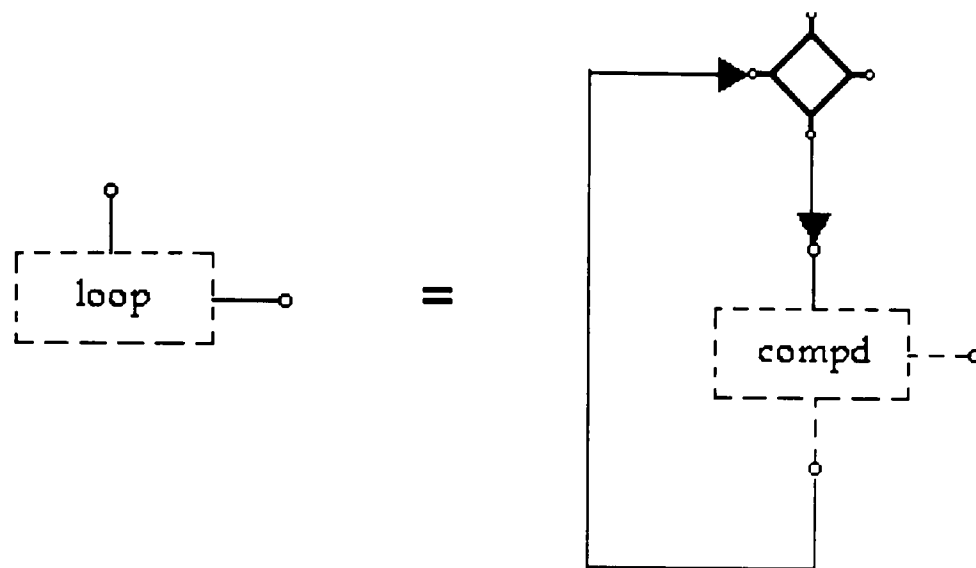


Fig. 4.16 Production 4

5. condition = single | multi

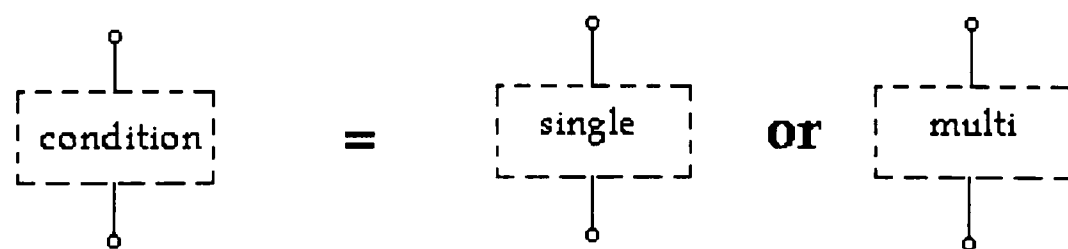


Fig. 4.17 Production 5

6. single = IF branch branch CONET

The horizontal dash line means that the CONET can catch all flows passed by the branches.

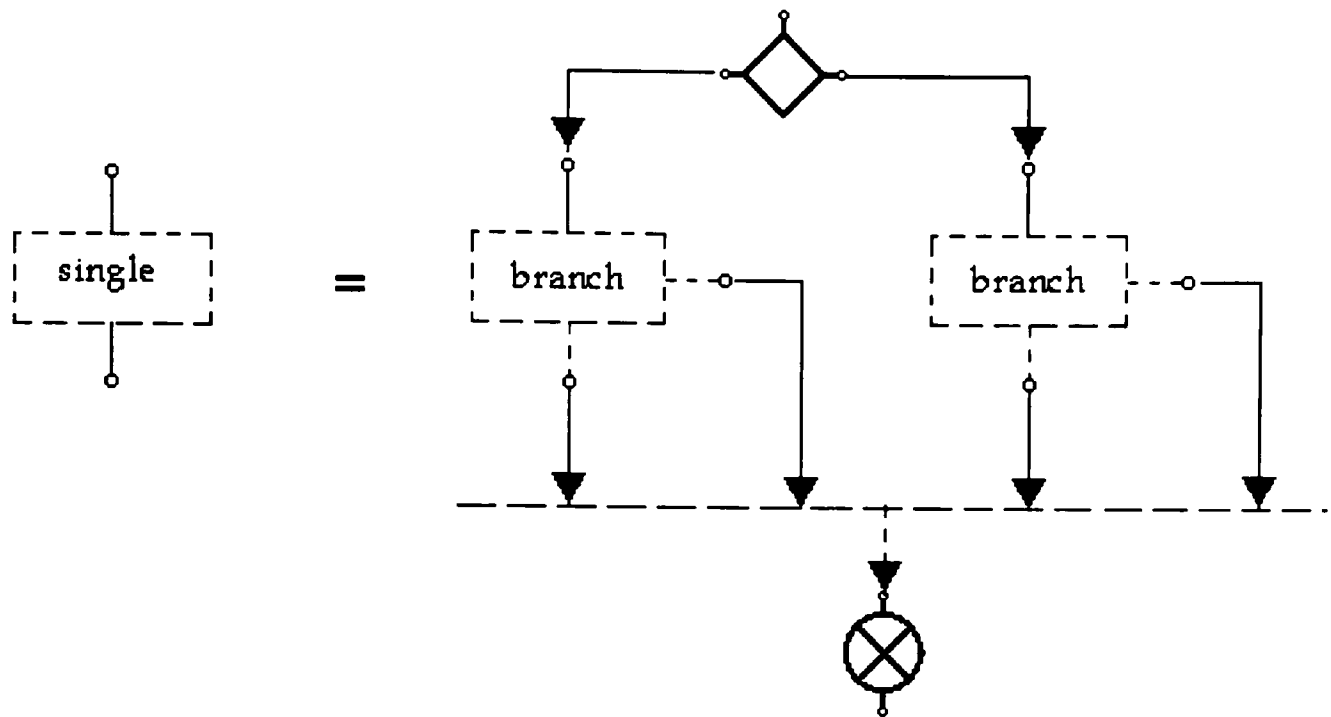


Fig. 4.18 Production 6

#### 7. branch = SLASH B compd down

The two sides of the production seems no big difference, the structures are very similar. But actually according to the definition, they are totally different structures. All the connections on the left side item -- 'branch' are part of this item, but for 'compd' on the right side, all the connections are separated part, they are together with 'compd' constructing the another item -- 'branch'. Although there are two diagram for this production, but they are recognized as same structure semantically.

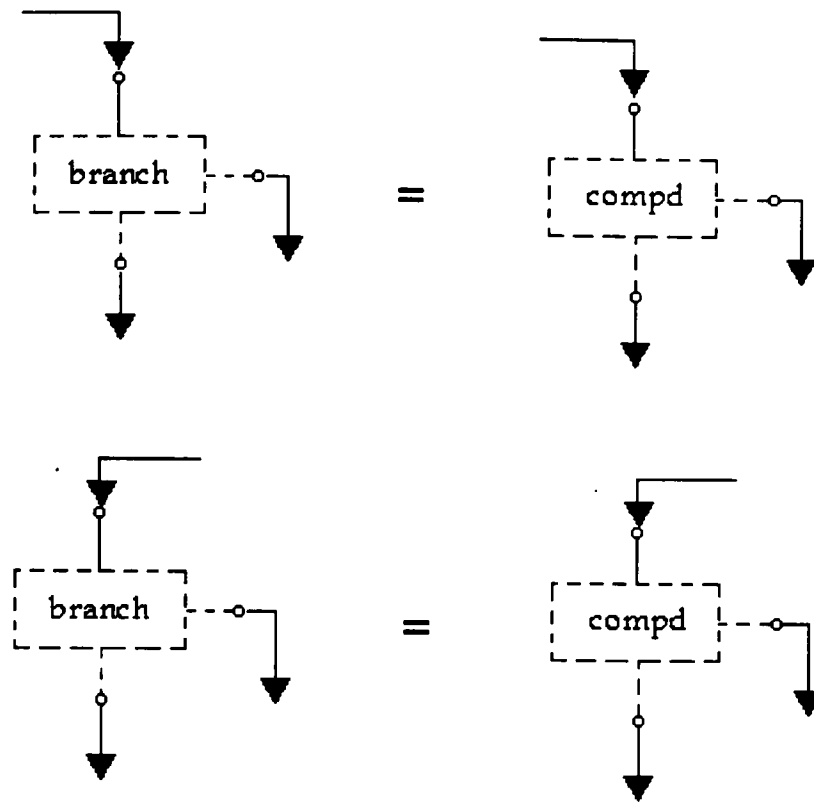


Fig. 4.19 Production 7

8. multi = SWITCH casepart CONET

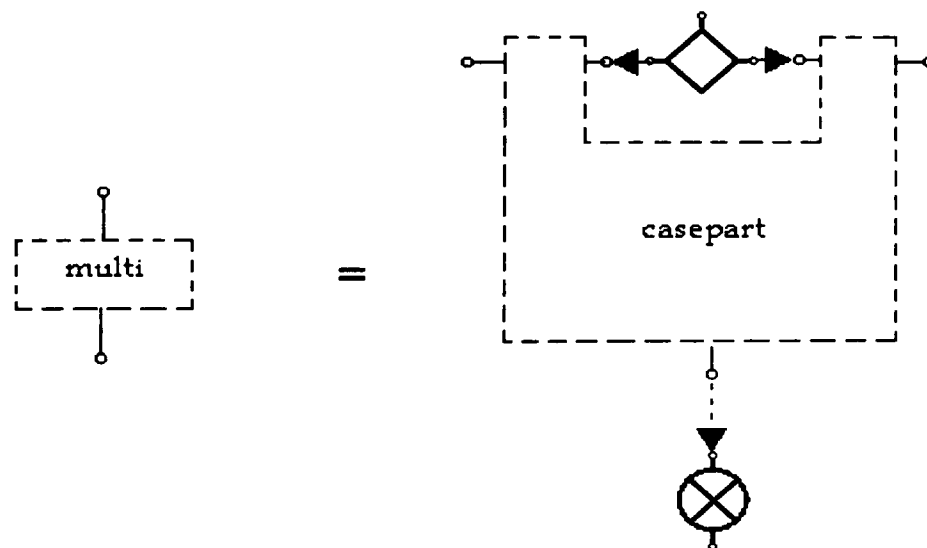


Fig. 4.20 Production 8

9.  $\text{casepart} = \text{onecase casepart} \mid \text{onecase onecase}$

According to this production, user at least need two cases, then he/she can construct a SWITCH-CASE structure.

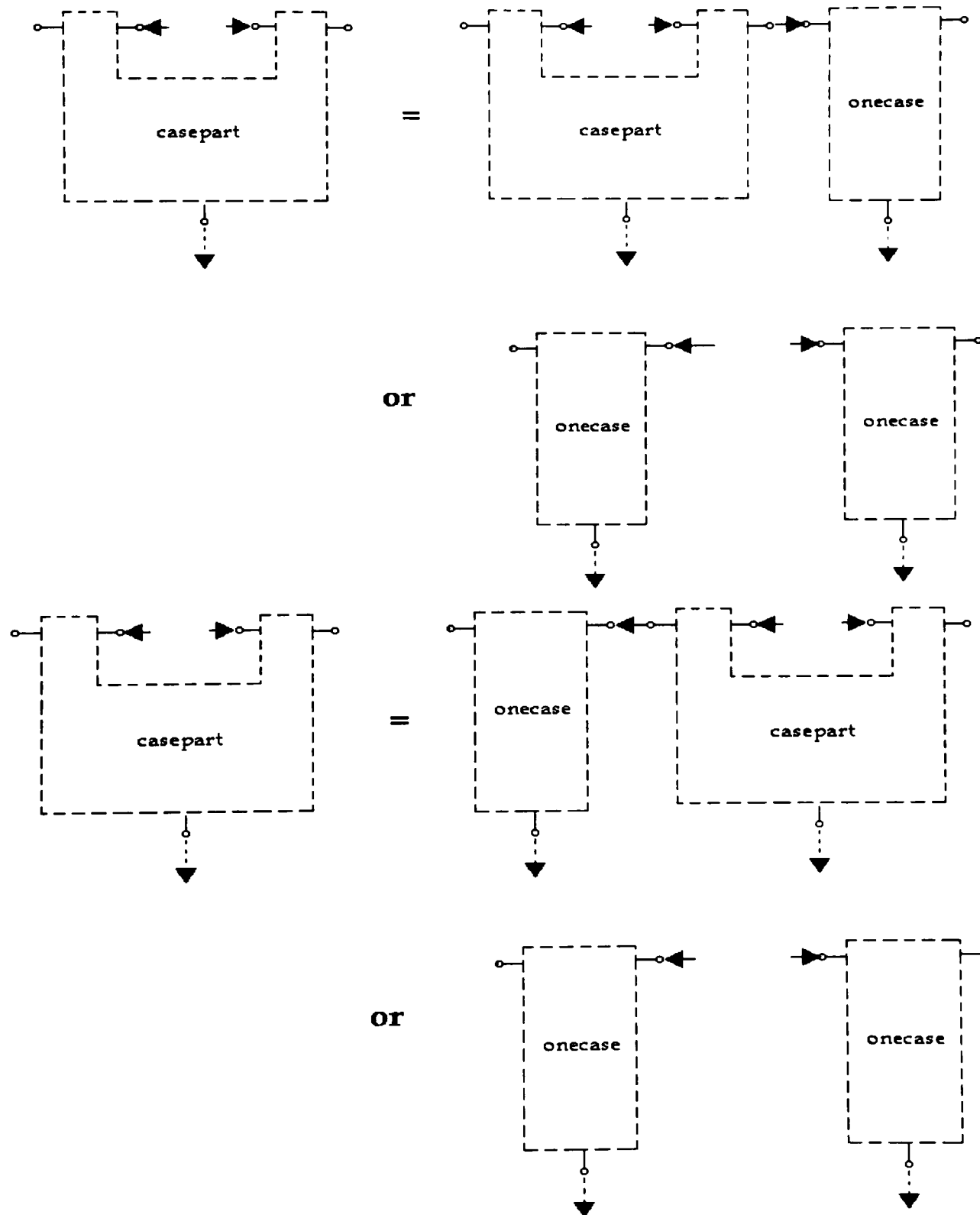


Fig. 4.21 Production 9

10. onecase = DASH A CASE A compd down

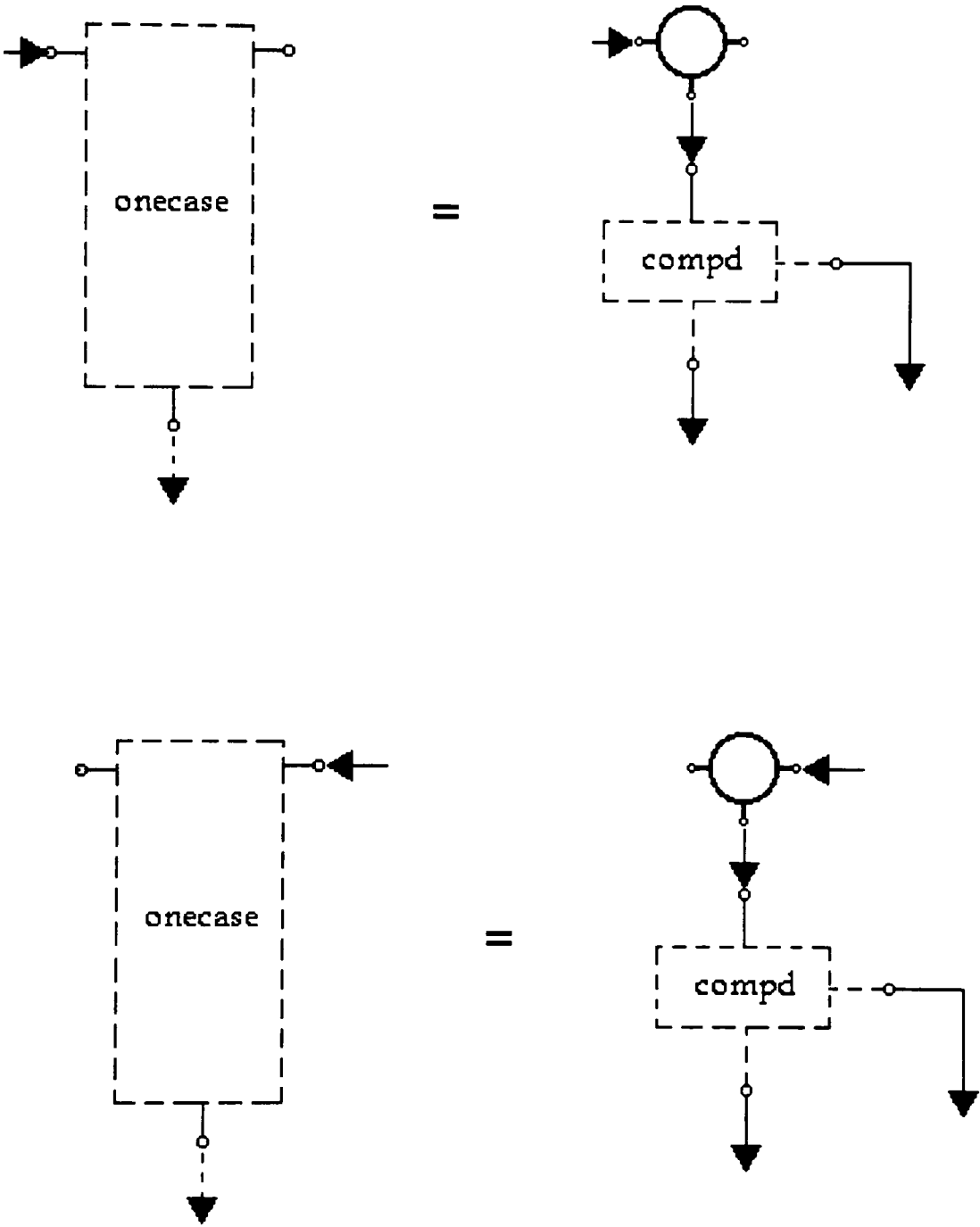


Fig. 4.22 Production 10

11. down = A | B

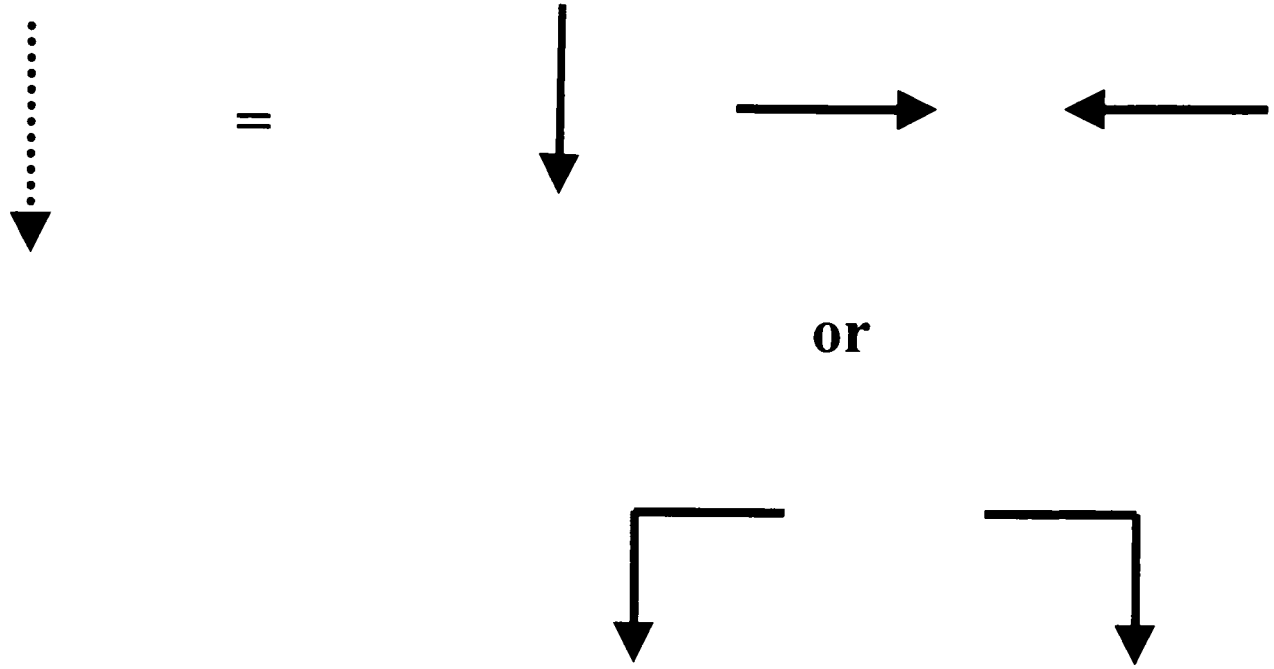


Fig. 4.23 Production 11

## CHAPTER V

### THE PARSER AND SCANNING SOLUTION

#### 5.1 The Parser

Scanning is the key issue for a two-dimensional visual programming language.. The job of the scanner, or lexical analyzer, is to read the input stream and divide it into tokens. In text-based programming languages, reading the input is trivial because the input stream is a character string, so the lexical analyzer just needs to get these characters one by one. Usually, as in a text-based programming language, the difficult part is how to recognize grammatically valid tokens that are constructed by these characters.

However, for a visual programming language, this is relatively easy. The smallest grammatical elements of the visual programming language are the icons, which are connected by lines. Obviously, icons that are both explicitly separated by spatial distance and related to some graphical object are much easier to be recognized than textual tokens, which have any number of characters and do not have an explicit delimiter. Conversely, the reading of the input stream is a major concern for a visual programming language scanner. In the one-dimensional environment, two directions of flow exist: forward and backward. The scanners for textual languages always go forward in the input stream. For two-dimensional languages, the possible directions are infinite. For the visual programming language defined for this research, there are only four valid directions for an icon: North, East, South and West. With a program



written in a VPL, when the scanner is at a particular icon. there is more than one possibility.

A DR parser solves this problem by modifying the parsing table. A traditional grammar  $G$  has four elements:  $N$ ,  $T$ ,  $S$  and  $P$  standing for nonterminals, terminals, start symbol and productions, respectively. But a positional grammar is a 6-tuple  $(N, T, S, P, POS, SP)$  [17]. Two extra elements, the  $POS$  is positional operators which is a set of positions in the space, and  $SP$  is the starting position. Both  $POS$  and  $SP$  give the scanner positional information; that is, which token should be next. The input is therefore different from the traditional input text, since the scanner accesses input items randomly using the position information. In this research, the approach was different; there are actually two scanners used. The first, a two-dimensional representation interpreter, is in charge of translating the two-dimensional structure into intermediate text. The second scanner will take the intermediate text and process, as is done for a scanner in a text-base language.

Fig. 5.1 shows the two different translation models. The one on the left is for the DR parser, while the one on right is used in this research. The input for DR parser is the data structure  $D_p$ , not a linear input. A function called 'Positional Query' uses the argument provided by the DR parsing table to arrange the scanning order of items in  $D_p$ . In the approach of this research, a traditional LR parser is used, its input is linear textual representation, which is generated from the data structure by using the 'Visual Representation Interpreter'. The mechanism of the interpreter is imitating the program flow to go through the visual representation. In this approach, the scanning process is

more environment-embedded. But the advantage is the parser could be constructed by using some compiler tool, such as LEX & YACC.

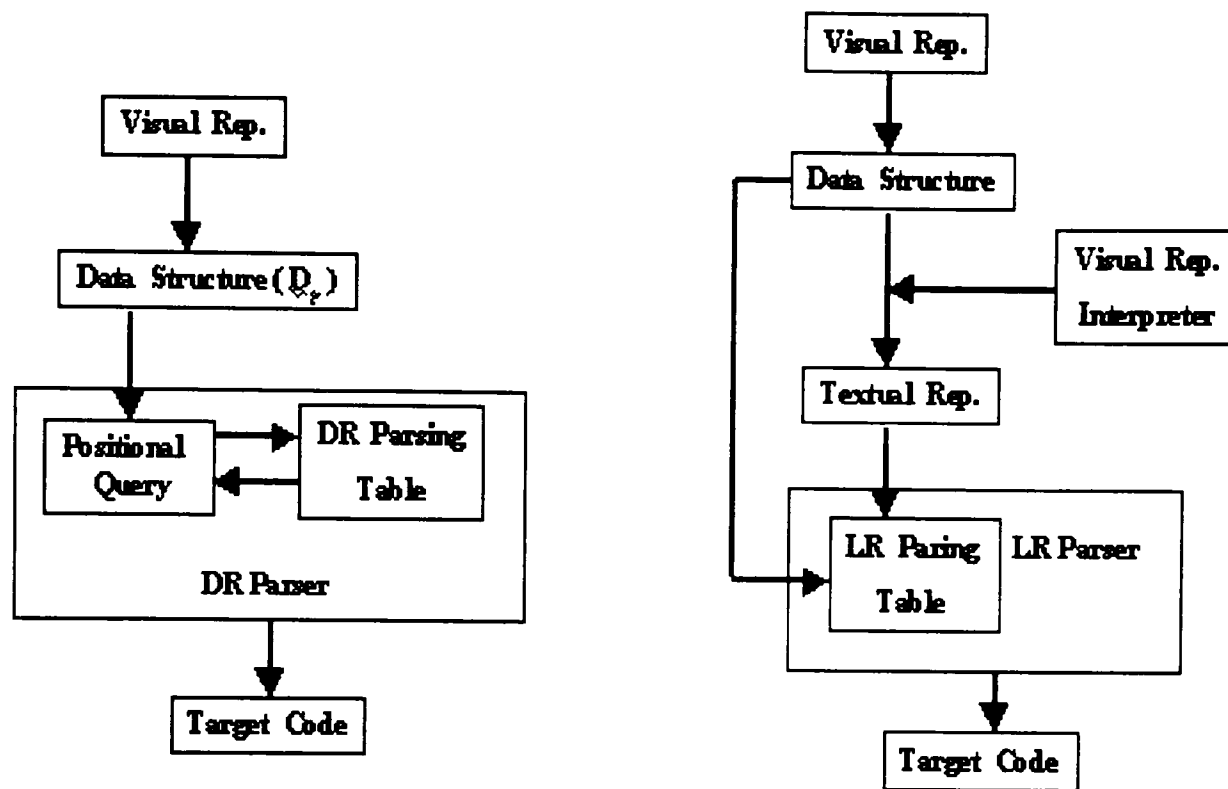


Fig. 5.1 Two Different Translation Models

## 5.2 The Go Through Scheme

Because parallel processing is not considered in this research, the program stream is one-dimensional, as it traverses through the two-dimensional structure. So the first scanner (two-dimensional representation interpreter) could imitate program flow to process a diagram. Each icon is like a stop for the stream; when it gets to a stop, the scanner has to decide what is the next direction to go in, and then go to the next stop.

For procedural languages, the program flow can be summarized into four patterns as shown in the following Fig. 5.2.

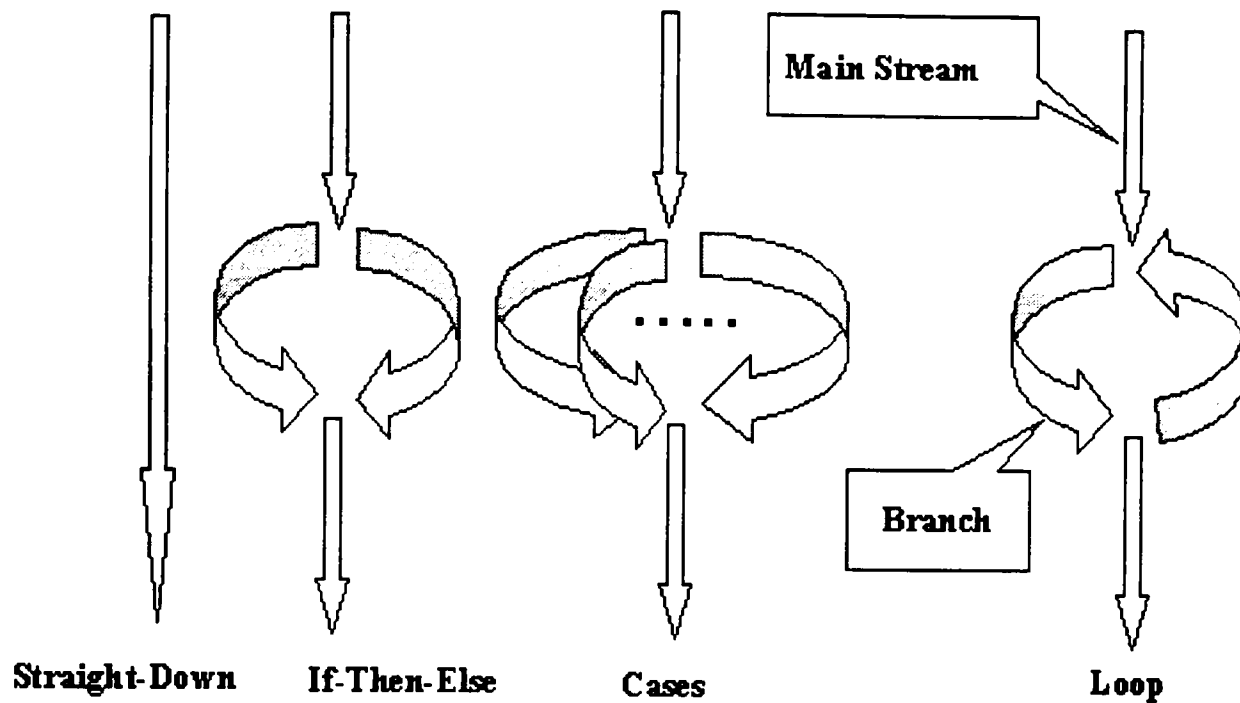


Fig. 5.2 Program Flows

For the Straight-Down pattern, there is not much difficulty in going through. The major concern are the branches derived from main stream. As is shown in Fig. 5.2, all branches start from same point, and finally come to another point. These points are called key points. For the visual programming language defined for this research, the key points are IF, SWITCH, TEST, CASE and CONET. These points can be divided into two groups according to the positions (branching out or branching in) they are at; CONET is the first group by itself, because it is the only icon at branching in position; and the rest construct the second group.

The Go-Through scheme for those icons outside of these two groups is rather simple. The STAT has one receiving point (the North) and one passing point (the

South), so the program stream goes through it from the North to the South; the START has one pass (the South), at the beginning of program stream; and the END has one catch (the North), at the end of program stream.

The icons in the second group require multiple passed. So the Go-Through scheme for them is about the priority of these pass. For IF and SWITCH (which use same icon), the priority is East over West; for TEST, the priority is South over East; for CASE, if it receives flow from the East and passes it to the South and the West, the priority is South over West, if it receives flow from the West and passes it to the South and the East, then the priority is South over East.

CONET needs to work with both the icons in the first group and a stack. When the icons in the first group is visited first time, they will be pushed in the stack, and when the last visit occurs (which in this research, will be the second time for all the key icons), they will be popped from the stack. Actually TEST does not need to work with CONET, since the program flow will automatically come back visit it second time. But for the other icons, which just have one receiving point, there is no explicit way for program flow to come back to make the second visit. Therefore, these icons need to work with CONET. When the program flow hits CONET, it checks the stack to see which one is on top, then goes back to the icon (usually during the second and last visit which means this item needs to be popped). When the IF or SWITCH is popped, the CONET with respect to it should be pushed into the stack. The reason for this is the IF or SWITCH received the main stream, and the CONET receives branches and continues the main stream. So when CONET is hit, and is also at the top of the

stack, it is then time to go on. The CONET will be popped from the stack and pass program flow to its South. If the CASE's second pass fails, in other words, there is no connection associated to the West or East, the program flow goes to the first icon on the top of the stack.

### 5.3 An Example

Fig. 5.3 contains a diagram with respect to a program, with Fig. 5.4 containing the database with the all necessary information for that diagram.

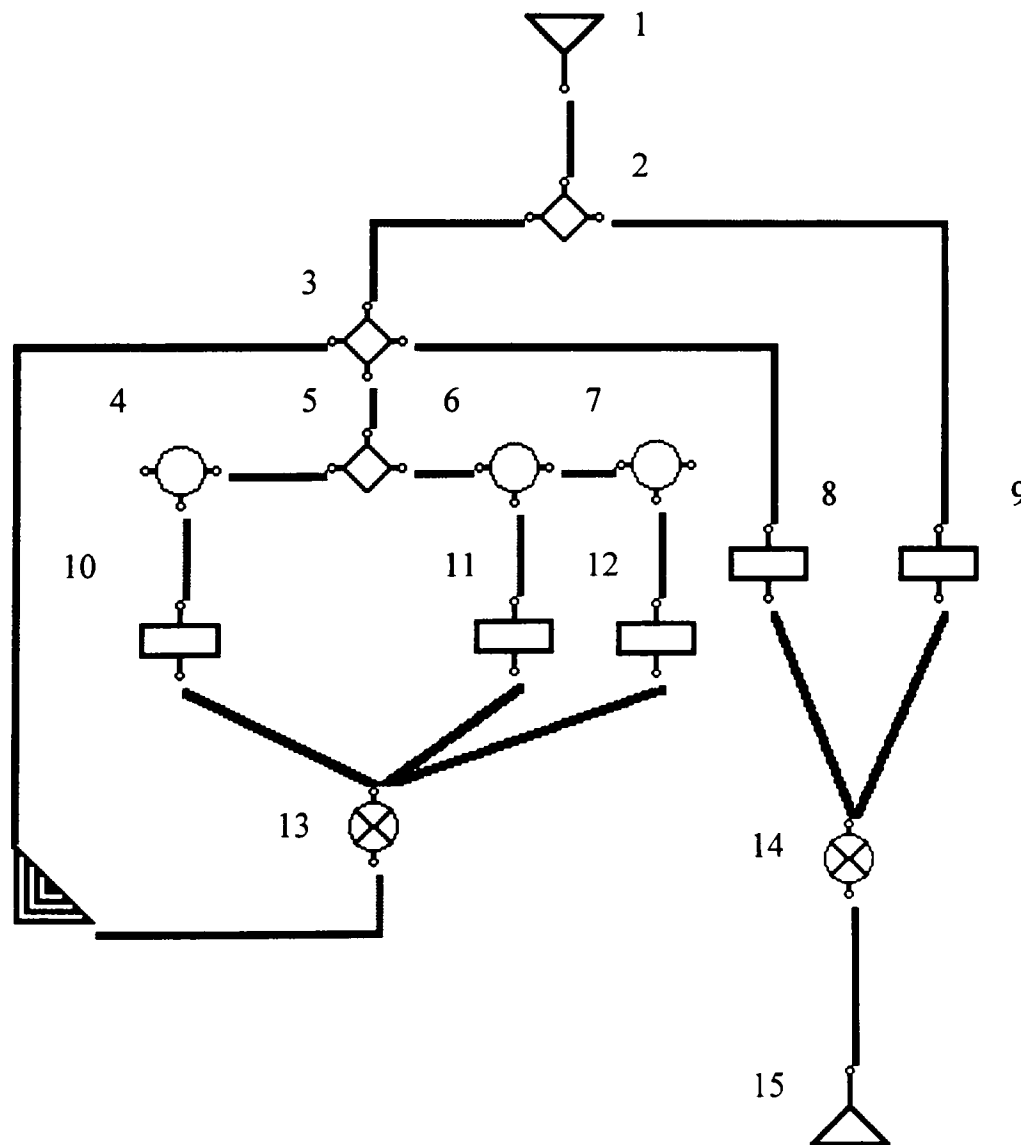


Fig. 5.3 Program Diagram

In this diagram, the program flow will be: 1--2--9--14--2--3--5--6--11--13--6--7--12--13--5--4--10--13--3--8--14--15. The Visual Representation Interpreter will traverse this route by using the information in the database shown in Table 5.1 and the Go-through scheme of each icon in the diagram. The whole process is shown in Fig. 5.4.

Table 5.1 Database

ID	TYPE	NORTH	EAST	SOUTH	WEST
1	START	X	X	pass 2	X
2	IF	catch 1	pass 9	X	pass 3
3	TEST	catch 2	pass 8	pass 5	catch 13
4	CASE	X	catch 5	pass 10	X
5	SWITCH	catch 3	pass 6	X	pass 4
6	CASE	X	pass 7	pass 11	catch 5
7	CASE	X	X	pass 12	catch 6
8	STATE	catch 3	X	pass 14	X
9	STATE	catch 2	X	pass 14	X
10	STATE	catch 4	X	pass 13	X
11	STATE	catch 6	X	pass 13	X
12	STATE	catch 7	X	pass 13	X
13	CONET	catch 10, 11, 12	X	pass 3	X
14	CONET	catch 8, 9	X	pass 15	X
15	END	catch 14	X	X	X

Step 1: Find the START icon in the database, which is the beginning point of the program flow. The stack is now empty. From the South, flow goes to 2.

Step 2: 2 is an IF icon, then push it onto the stack. Now the stack is (top 2 bottom). On the first visit, flow goes from the East to 9.

Step 3: 9 is a STATE. From the South of it, the flow goes to 14.

Step 4: 14 is a CONET. Then check the stack to see which token is next. Now 2 is on top of the stack, so the flow goes to 2.

Step 5: It is the second visit for 2, so 2 is popped up from the stack and 14 is pushed into the stack (top 14 bottom). The flow goes from the West to 3.

Step 6: 3 is a TEST, and it is the first visit. So 3 is pushed into the stack (top 3 | 14 bottom). The flow goes from the South to 5.

Step 7: 5 is a SWITCH, and it is the first visit. So 5 is pushed into the stack (top 5 | 3 | 14 bottom). The flow goes from the East to 6.

Step 8: 6 is a CASE, and it is the first visit. So 6 is pushed into the stack (top 6 | 5 | 3 | 14 bottom). The flow goes from the South to 11.

Step 9: 11 is a STATE. The flow goes from the South to 13.

Step 10: 13 is a CONET. Then check the stack to see which is the next. Now 6 is on the top of the stack. So the flow goes to 6.

Step 11: This is the second visit of 6, so 6 is popped up from the stack (top 5 | 3 | 14 bottom). The flow goes from the East to 7.

Step 12: 7 is a CASE, and it is the first visit. So 7 is pushed into the stack (top 7 | 5 | 3 | 14 bottom). The flow goes from the South to 12.

Step 13: 12 is a STATE. The flow goes from the South to 13.

Step 14: 13 is a CONET. Then check the stack to see which is the next. Now 7 is on the top of the stack. So the flow goes to 7.

Fig. 5.4 The Go-through Process

Step 15: It is the second visit for 7. So 7 is popped up from the stack (top 5 | 3 | 14 bottom). From the East of 7, the flow has nowhere to go. So check the stack, 5 is on the top of the stack. Then the flow goes to 5.

Step 16: It is the second visit of 5. So 5 is popped up from the stack. Because 5 is a SWITCH, the last CONET 13 is pushed into the stack ( top 13 | 3 | 14 bottom). The flow goes from the West to 4.

Step 17: 4 is a CASE, and it is the first visit. So 4 is pushed into the stack (top 4 | 13 | 3 | 14 bottom). The flow goes from the South to 10.

Step 18: 10 is a STATE. The flow goes from the South to 13.

Step 19: 13 is a CONET. So check the stack, 4 is on the top of the stack. Then the flow goes to 4.

Step 20: It is the second visit for 4, so 4 is popped up from the stack. (top 13 | 3 | 14 bottom). From the West of 4, the flow has nowhere to go. So check the stack , now 13 is on the top of the stack. So it is the time for 13 (a CONET) to do the passing. Pop 13 up from the stack (top 3 | 14 bottom). From the South of 13, the flow goes to 3.

Step 21: It is the second visit for 3. So 3 is popped up from the stack (top 14 bottom). The flow goes from the East to 8.

Step 22: 8 is a STATE. The flow goes from the South to 14.

Step 23: 14 is a CONET. Then check the stack, now 14 is on the top of the stack. So it is the time for 14 (a CONET) to do the passing. Pop 14 up from the stack (top bottom). From the South of 14, the flow goes to 15.

Step 24: 15 is a END. It indicates the end of the program or diagram, so the structure has been gone through.

Fig. 5.4 Continued

After the Go-through process, the intermediate code shown in Fig. 5.5 is generated. The intermediate code and the database of Table 5.1 will be the input for the compiler. The intermediate code contains the order information of icons; the database



contains detailed information of each icon, it is like a symbol table. Fig. 5.6 shows the parse tree for the intermediate code of this example.

1 A 2 / B 9 A / B 3 A 5 | 6 A 11 A | 7 A 12 A | 4 A 10 A 13 C B 8  
A 14 A 15

Fig. 5.5 The Intermediate Code

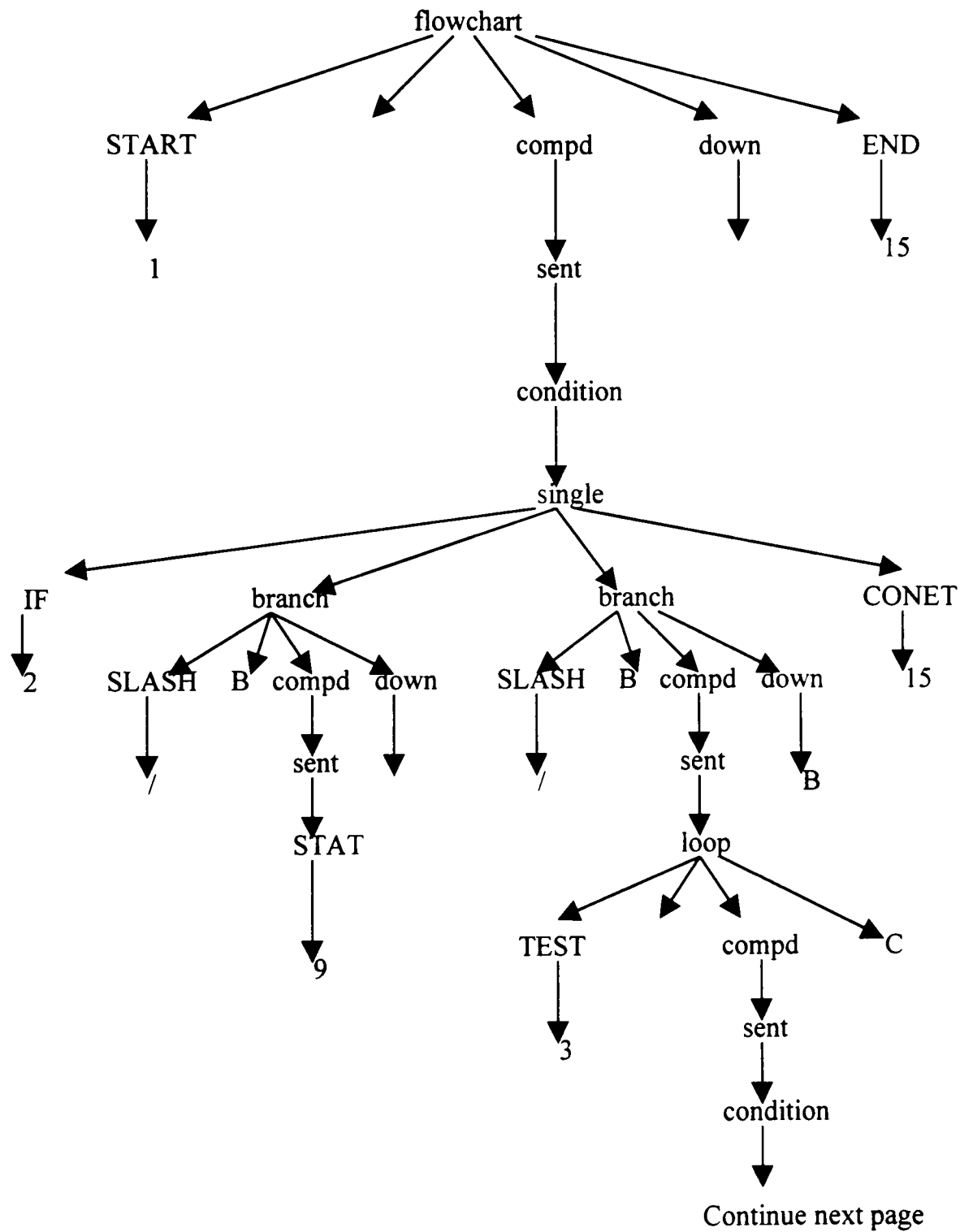


Fig. 5.6 Parse Tree

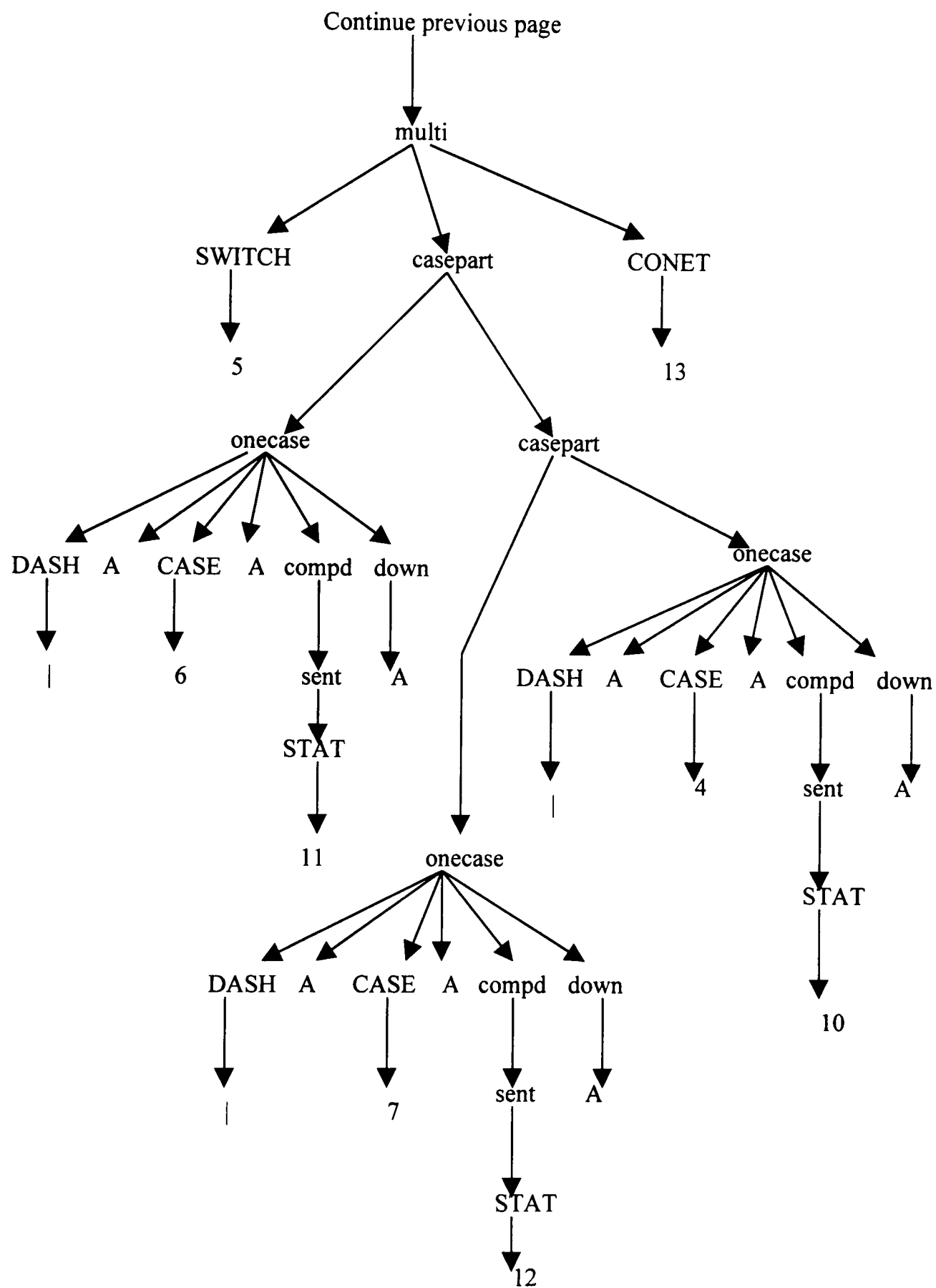


Fig. 5.6 Continued

## CHAPTER VI

### ERROR HANDLING

#### 6.1 Error Types

There are two kinds of errors if not including those errors within the dialog box, which are a small-grain level problem, which is not the concentration of this research. The first type of error concerns diagrams (the visual representation), which is a complete and valid structure according to the grammar. This is the major error that needs to be dealt with, because the environment constructed based on the theory is a free form environment. The user draws his/her own picture by creating icon objects and connecting them to different kinds of connections. The approach is different from BACCII++ and STRIDES. While this free form environment gives more flexibility, it also allows the user to make more errors. However, a careful design could avoid most of errors from happening (more details will be given in next section). The second type of error is within the structure; that is, after the diagram is checked correct, there still could be errors. This type of error is rather limited compared to the first one, however, since every icon object, when is first created, is empty, and does not have any content related to it. Of course, some icons, like START, END and CONET, never need related content, because their major function is to help connect part of the diagram. For the other icons, before the diagram could be translated into intermediate code, the dialog box related to each icon must be filled out. Actually, the compiler will need two parts of input to translate the diagram into target language code. Besides the intermediate

representation of the diagram, a data base storing information related to all icons is also needed. Also, for this language, the predicate plays different roles in different situations (it could be IF or SWITCH), which could lead to errors by misconnection. All the errors mentioned above are dealt with in different levels with respect to their different types.

## 6.2 Dealing with Errors in Different Levels

First of all, at design time of the environment, all decisions should be made by considering possibility of errors. For those events that might create errors, the environment is simply designed to not allow them happen.

The environment created in this research is IceC, which stands for an iconic environment for C++. In IceC, the activities consist of creating icon objects and connecting. Some icons, like START and END, just allowed once in a diagram or program. So if one of these icons already exists, when the user tries to create a new instance of this icon, the environment would not allow this to happen. Then the connections. Each connection must connect two icon objects, if user fails to connect them, the connection would not be created. Each connection is a part of program flow, it has its direction. When the user wants to create a connection object, he/she must follow three steps. First, click the icon for the connection. Second, click the icon object which passes flow. Third, click the icon which receives flow. The connection object is then created. The Failure of anyone of these three steps causes the creation action to fail.

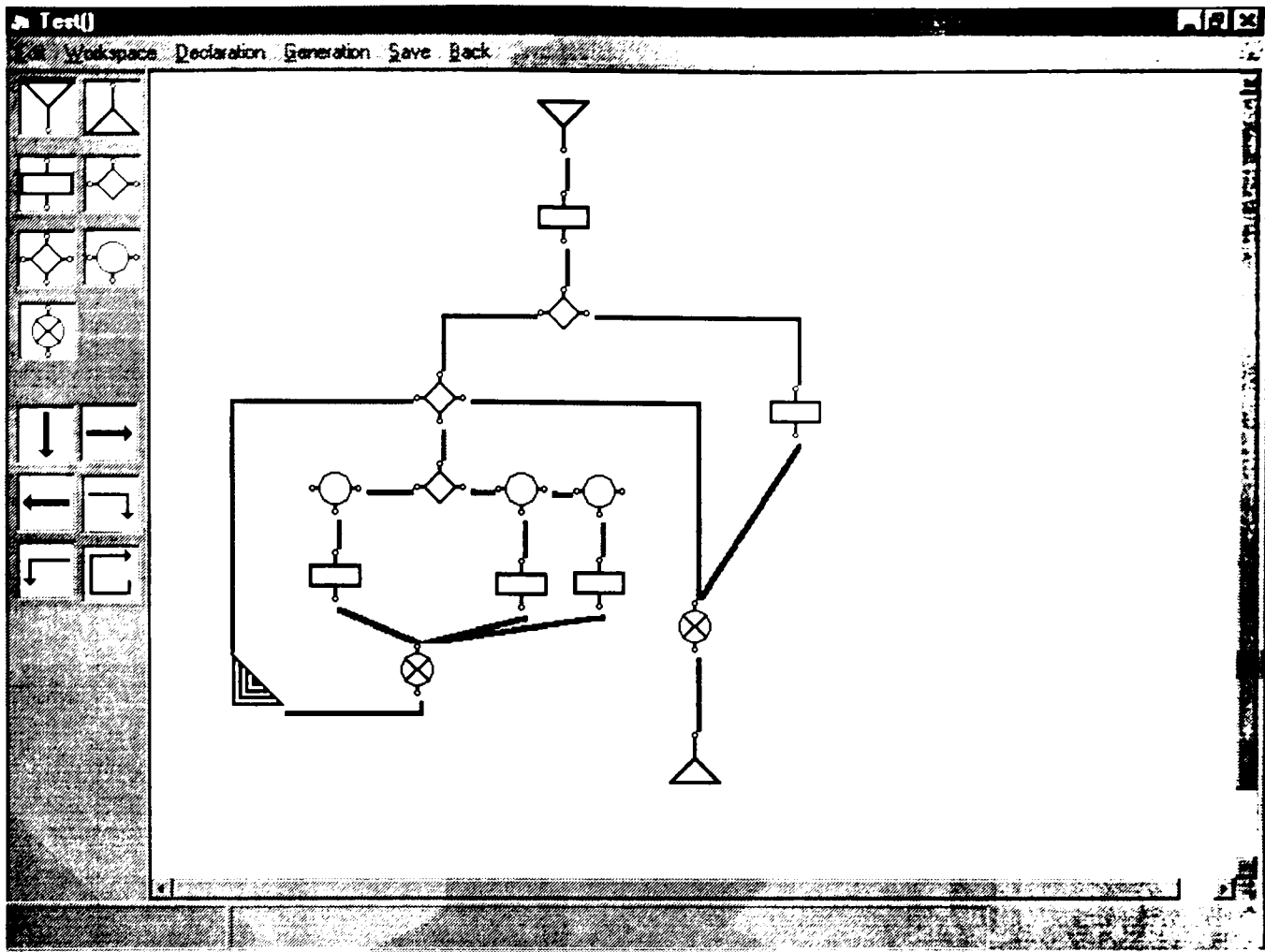


Fig. 6.1. Working Space

The working space of the environment is shown as Fig. 6.1.. On the toolbar, second section as shown in Fig. 6.2., from up to down, the first row, from left to right, the connections are labeled as 1, 2; the second row, from left to right are 3, 4; the third row, from left to right are 5 and 6. As seen in this figure, 1 is from this icon's South to next icon's North; 2 is from this icon's East to next icon's West; 3 is from this icon's West to next icon's East; 4 is from this icon's East to next icon's North; 5 is from this icon's West to next icon's North; 6 is from this icon's South to next icon's West. Each icon has its own possible connection.

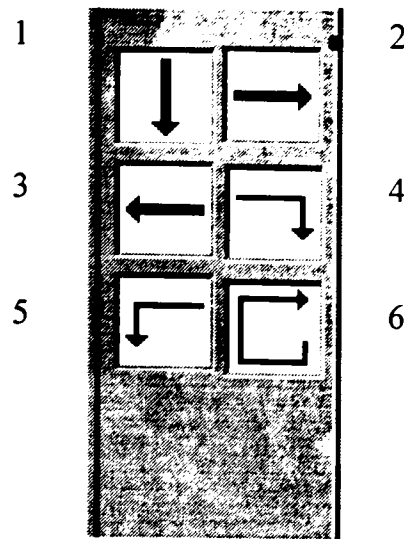


Fig. 6.2. Part of the Toolbar

Table 6.1 shows the relations between icons and connections. Each connection has its direction, it starts at a point and ends at a point. Both the starting point and ending point must be on an icon.

Table 6.1 Relations Between Connections and Icons

Connection	starting point	ending point
1	START, STAT, TEST, CASE, CONET	END, STAT, COND, TEST, CONET
2	COND, CASE	CASE
3	COND, CASE	CASE
4	COND, TEST	END, STAT, COND, TEST, CONET
5	COND	END, STAT, COND, TEST, CONET
6	STAT, CONET	TEST

After the user is finished creating a diagram and requests it, the interpreter will first check if all icons are connected. Each icon object's attaching points should be occupied by some kind of connection (except the CASE). For a CASE object, among

the East and the West, only one of them is connected with the connection 2 or 3's ending point, then it will be counted as a valid one. This is the first round of checking.

The second round checks to see if all the icon objects that have a related dialog box have been defined. If so, all information entered through dialog boxes will be transferred into a database file which will be part of the input for the compiler. If there is no dialog box information, an error message will be given, and the related icon objects will be highlighted.

The third round goes through the entire diagram. If the structure can not be gone through, there must be some error. How can the interpreter tell if a diagram has been gone through or not? First, all icons have their official visit time, usually one or two times, and each connection only could be passed through once. So, any violations in the go through process indicates an error. Second, if the END icon is hit but there are still some icons that have not been finished, there must be errors. Third, if the flow comes to the END icon, but the stack is not empty, something must be wrong. If none of above situations happen, then the intermediate code will generated. Intermediate code is about the order of all icons in the diagram according to the route that program flow has gone through. It is the another part of the input to the compiler.

The final round, the intermediate code and the database file is fed into the compiler. The compiler is constructed by using LEX and YACC, based on the grammar defined before. Theoretically, the major function of the compiler is not for error checking in the diagram level, because after all the checking mentioned above, the possibility of error should be very low. If it is not true, then the problem should be

solved in the previous stage, not this stage. The major function of the compiler is to generate the target code and detect deeper level errors. But that does not mean the compiler would not detect the errors in diagram, it just has not been designed to do so.

The compiler just detects two types of errors, both related to the predicate. As shown before, the predicate could be defined as either IF or SWITCH. However the structures with respect to these two key icons are different. So, the user might mix up these two kinds of structures. The rule is that the structure is decided by the key icons. That is, if the key icon is IF, then the structure is supposed to be an IF\_THEN structure. Then, the error will be detected in the rest of the structure, not in the key icon. The second kind of error concerns CASE. There is only one default case is allowed in the SWITCH structure. If user has defined more than one default case by mistake, it should be detected by the compiler. Fig. 6.3 is part of input file to LEX and YACC, showing how these are done.

```
single :    IF branch branch CONET
    {
        /*valid case, do the translation*/
    }
    |      IF onecase branch CONET
    {
        yyerror("Mixed up IfThen statement with Switch statement");
        YYABORT;
    }
    |      IF branch onecase CONET
    {

        yyerror("Mixed up IfThen statement with Switch statement");
        YYABORT;
    }
```

Fig. 6.3 Part of Input File for LEX & YACC



```

|      IF casepart branch CONET
{
  yyerror("Mixed up IfThen statement with Switch statement");
  YYABORT;
}
|      IF branch casepart CONET
{
  yyerror("Mixed up IfThen statement with Switch statement");
  YYABORT;
}
|      IF casepart CONET
{
  yyerror("Mixed up IfThen statement with Switch statement");
  YYABORT;
}
;

multi :  SWITCH casepart CONET
{
  /*valid case, do the translation*/
}
|      SWITCH onecase branch CONET
{
  yyerror("Mixed up Switch statement with IfThen statement");*/
  YYABORT;
}
|      SWITCH branch onecase CONET
{
  yyerror("Mixed up Switch statement with IfThen statement");*/
  YYABORT;
}
|      SWITCH casepart branch CONET
{
  yyerror("Mixed up Switch statement with IfThen statement");*/
  YYABORT;
}

```

Fig. 6.3 Continued

```
|      SWITCH branch casepart CONET  
{  
  yyerror("Mixed up Switch statement with IfThen statement");*/  
  YYABORT;  
}  
;
```

Fig. 6.3 Continued

## CHAPTER VII

### INTERFACE AND DATA STRUCTURE

#### 7.1 Overview

The compiler issues discussed previously are not necessarily for one particular programming language. The focus of this research is the program flow in a main program or a function. When the user interface is discussed in this chapter, it is primarily used for declaration.

The user interface connects the user and the program database. Basically, the content of the database could be classified in three sections: variables, functions and user defined data types. In IceC, the user defined data types mostly refers to a class. When the user accesses the database, there always is a module level with respect to that access. When the environment is at the global level, this module is 'Global'. If the user choose the class 'ThisClass', then the module is 'ThisClass'. For a particular function, since the same function name could appear in different places, both the class name and the function name are used to determine the proper module. This is done by giving each of them a unique number.

There are three types of user's actions: Add, Delete and Modify. In an Add action, a new record is added into the database. However before adding, the environment must check to see if there is any conflict with already-existing names. For instance, the user can not define two variables with the same name in the same module. In a Delete action, if the item is also a module, such as a class or function,

all other records in this module should be deleted at the same time. In a Modify action, the database stays at one particular record; the user can modify any fields of the record as long as there is no conflict to other records, but the user can't delete it. So the database is the same, but it will appear differently on the screen

Fig. 7.1 shows the structure of the software. Rectangle stands for interface; Oval stands for component of database. There are four groups of interface: Class Definition, Function Definition, Function Body and Variable Definition. The Function Body provides environment for coding, the others are for declaration. Corresponding to the four groups of interface, there are also four groups of database components. The Icons and Connections stores the visual program information, while the others stores declaration data. The arrows can be divided into three groups: the user's access to interface (labeled by number 1-7), interface's access to database (labeled by lowercase letter a-e), the relations among different database components (labeled by uppercase letter A-D). The user can enter a particular interface in different ways. For instance, the user can get to the Variable Definition interface by 3, 2-5, 1-4 or 2-6-4 four routes. The route 3 is a global level access; 2-5 is a class data member level access; 1-4 is a global function variable level access; 2-6-4 is a class member function variable level access. Basically, each type of interface has its corresponding component in database. The accesses are bi-directional, through interface, the user can either get information from the database or modify data. Particularly, through the access b, the environment for function body could either display current function's target code generated previously or put the target code

generated this time into the database. The third group of arrows are within the database, they relates to the control of consistency for database. The control of consistency is one-directional, from high level to low level. The levels from high to low are Class, Function, Icons and Connections, Variable. For instance, if the user deletes a variable, the database does not need to check tables related to class or function to keep the consistency; but if the user deletes a class, besides updates tables related to class, database still needs to check all other components to get rid of all functions and variables within the class.

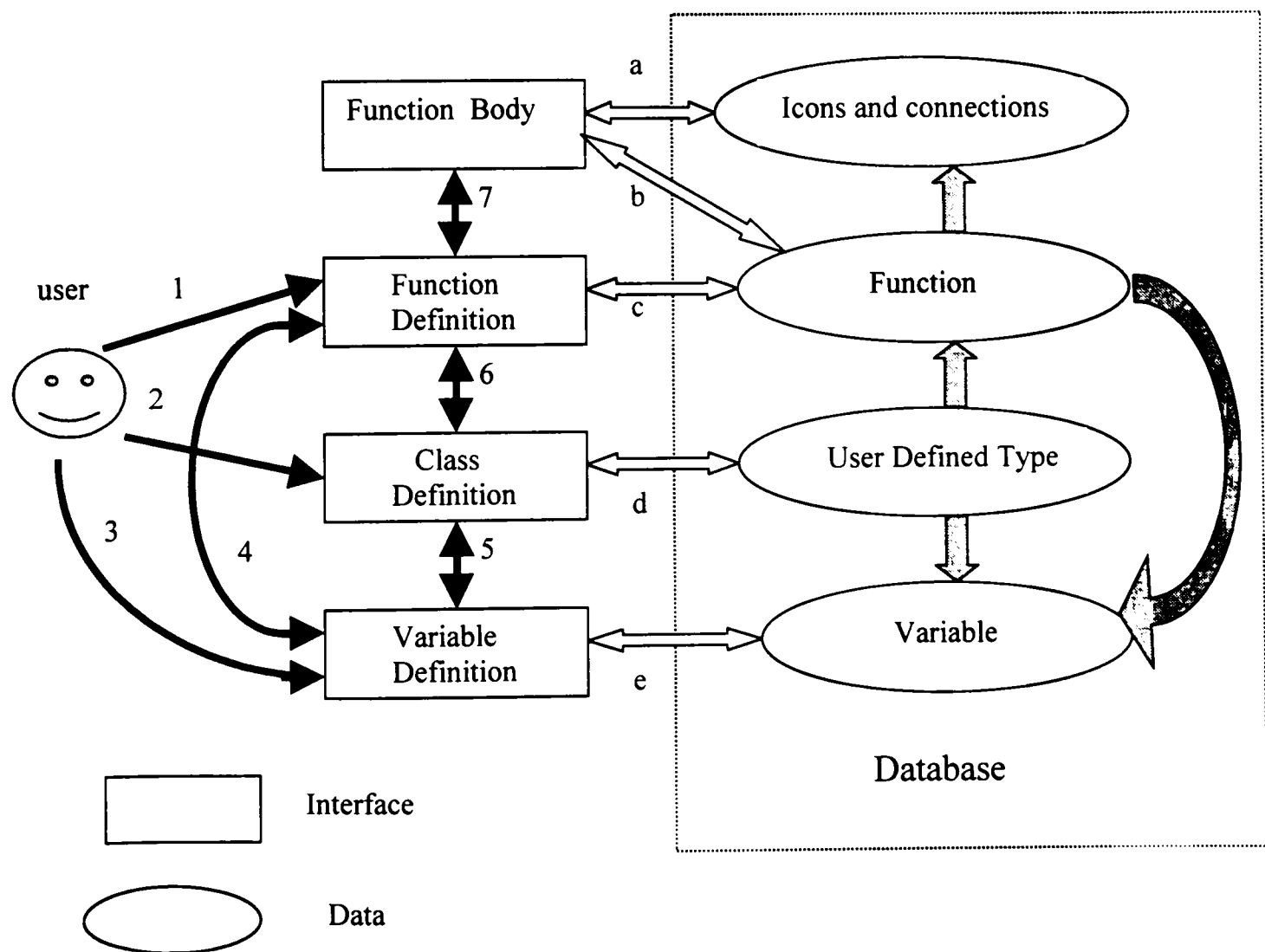


Fig. 7.1 Structure of the IceC Environment

## 7.2 Classes

A class definition includes objects, and relations between these objects. Of course, the data and function members are also part of the class definition; however, they will be discussed in the following sections. Fig. 7.2 has the work space for a class definition; and it is also the main screen for the software environment.

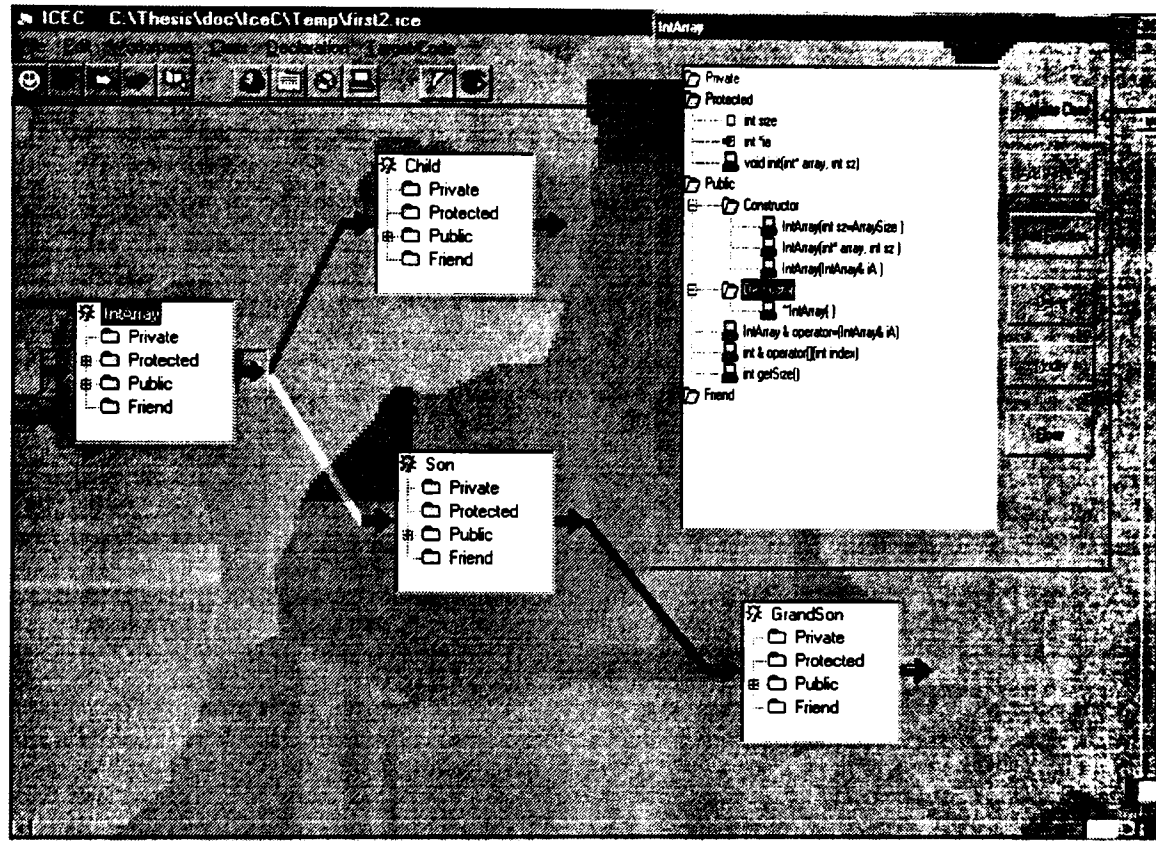


Fig. 7.2 Class Definition Interfaces

The first section of buttons on the toolbar, from left to right, are Class, Public, Protected, Private, and Details of Class. The Class button is for creating a class object. The Public, Protected and Private buttons are used for defining inheritance relations between classes. Visually, they are shown as connecting lines in Fig. 7.2. Green is for Public, Yellow is for Protected and Red is for Private (the color of connection imitates transportation lights). When the user wants to modify

the information inside of one class, he/she needs to highlight the class object and either click the Details of Class button or double click the class object; the window on the top right in Fig. 7.2 then appears.

### 7.3 Functions

The user can define a function at either the global level or the class level. As is shown in Fig. 7.3, when the user first opens the function declaration window at the global level, there is already an automatically generated main function there. The function declaration at the class level is shown in Fig. 7.3 as the previous picture in the top right window; when the user clicks on the modify button, the function modification window pops up as shown. The window caption of the window displays the module information of the function. In this window, user can modify the function name, parameter list, and return value type. For the details of the function, click on the Function Body button; then the interface shown in Fig. 6.1 pops up.

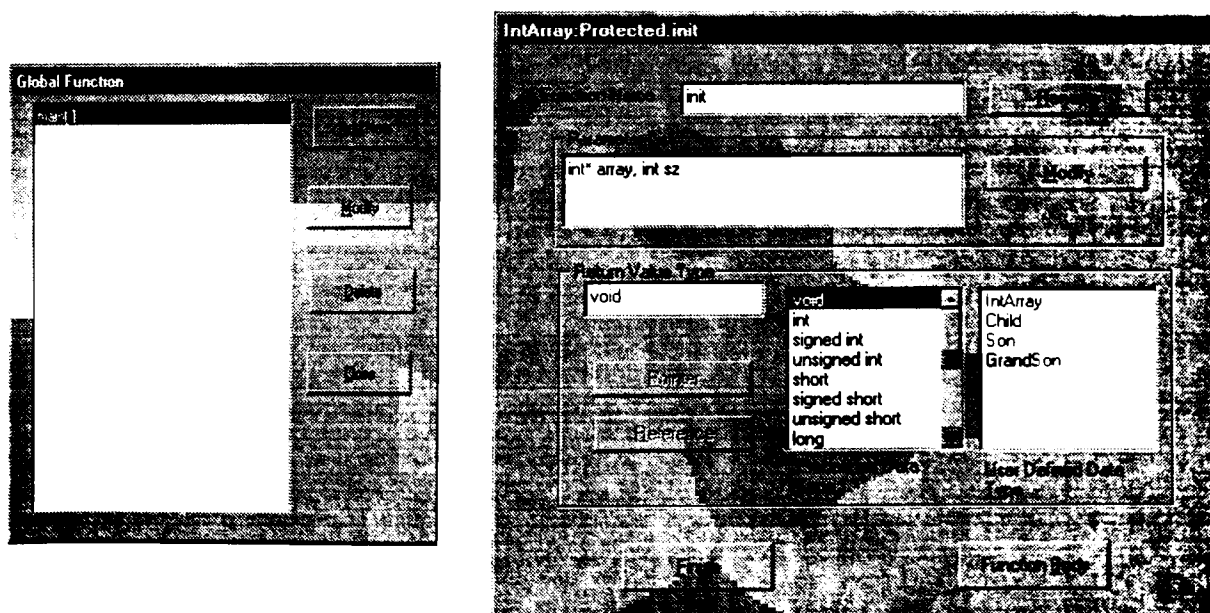


Fig. 7.3 Function Definition Interfaces

## 7.4 Variables

Variable declarations are done in three levels: global level, class level and function level. Basically the windows for them all with the same appearance, the only difference that the caption of the window would show the current module, and of course, the content is different. Also, for the function parameters declaration, the window is different, as shown in Fig. 7.4. The only reason for the difference is that the parameters need additional information for the order of them. So in this window, user can change the order of the parameters if desired. However, clicking on the modify or the new button on either window will pop up the modification window as shown in Fig. 7.5. When the user clicks on a different data type, the window for that particular data type will show up in either the New or Add mode. In the Modify mode, the data type is firm, i.e., user can not change it, but can still change any other information.

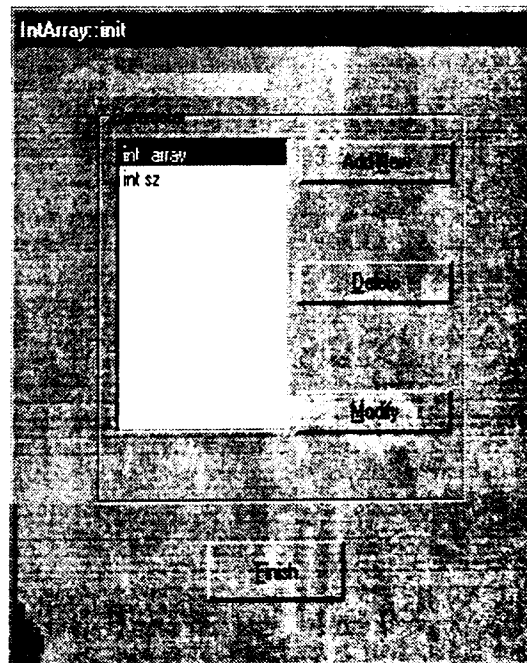


Fig. 7.4 Parameter Definition Interface



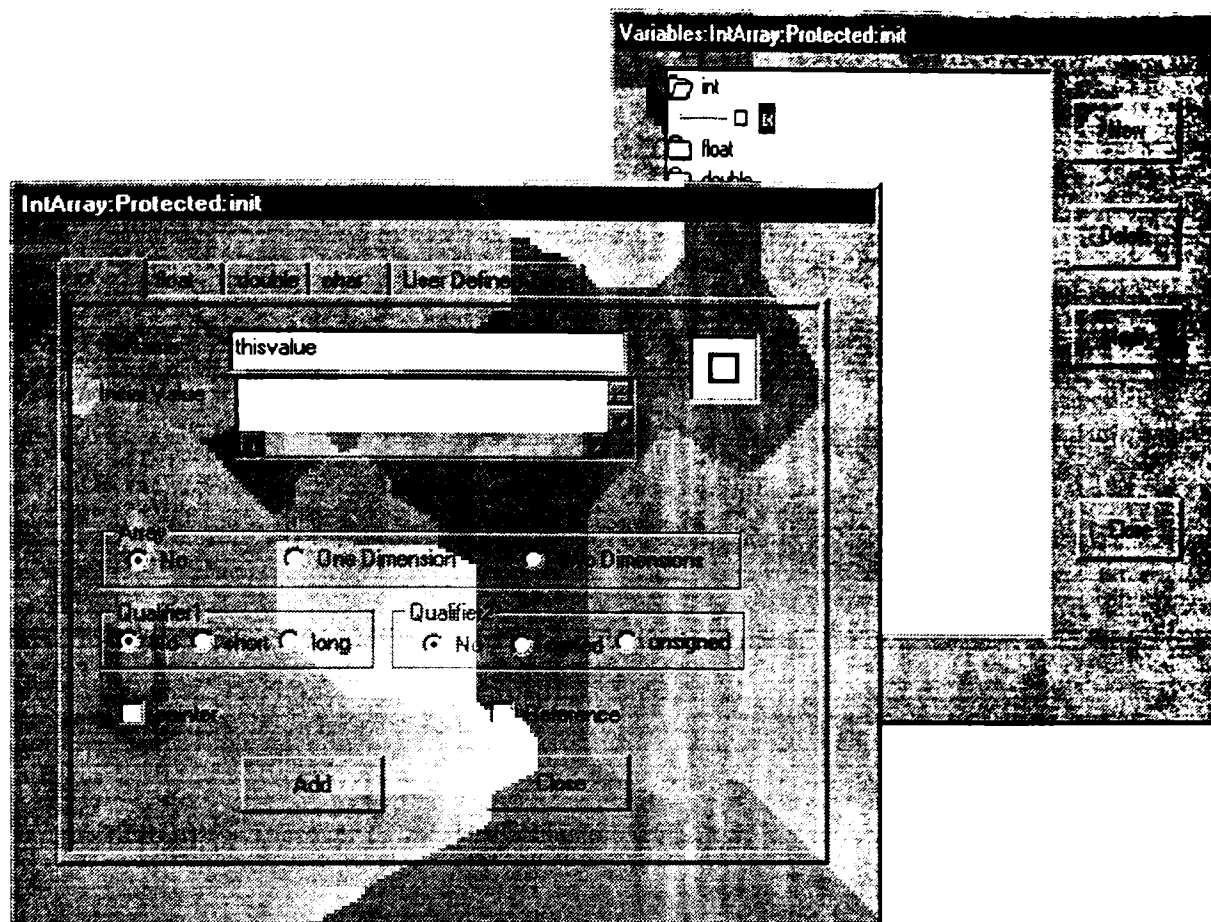


Fig 7.5 Variable Definition Interfaces

## CHAPTER VIII

### CONCLUSIONS AND FUTURE RESEARCH

#### 8.1 Conclusion

The purpose of this research was to study the visual representation of traditional programming languages and problems related to their implementation. An iconic BNF compared to the traditional BNF was created. Because of the similarity of traditional programming languages at procedural level, this iconic BNF can generate the visualization of any traditional programming languages at this level theoretically.

This research implemented the iconic BNF for C++. An software, IceC (An Iconic Environment for C++), was created. Icons play a central role in this iconic visual language. Icons are major part of the token system, they construct the main body of program or function. Icons' connection attributes (attaching points: North, East, South and West) provide the possibility of two-dimensional structure. The dialog boxes related to icons let the user enter more detailed information of program. In one words, all programming activities relate to icons.

Icons are also very important for realizing the two-dimensional scanning. Compared to traditional text-based languages, the difficulty for the compilation of iconic visual languages is how to scan two-dimensional structure. In this research, a visual representation was used. The mechanism of this visual representation interpreter is imitating program control flow. For each icon, it is a stop of program flow, it receives flow from previous icon and passes flow to next icon according to its go-through

scheme. Different types of icons have different go-through schemes. The icons that can receive or pass flow on multiple directions are 'key' icons, because they can generate branches of program flow. Their go-through scheme are most complicated, and also most important for the visual representation interpreter. A group of icons can construct a program by connecting their connection attributes. Then the interpreter can go through all of them according to their go-through schemes. In the go-through process, the interpreter transfers the two-dimensional structure into one-dimensional text, the intermediate code. The intermediate code and a data structure, which contains the content of dialog boxes related to each icon, are input for the compiler. The compiler is a traditional LR parser, generated by using LEX & YACC with the textual representation of the iconic visual language's productions.

Although the system was implemented for C++, by making small modification on the dialog boxes and some translation function, it could be used for other traditional programming languages.

This thesis proposed a new approach in solving the compilation problem for iconic visual languages; applied some basic theories of iconic visual languages in the area of traditional programming languages' visualization. Some VPE issues were also studied in the implementation.

## 8.2 Future Research

This research proposed a prototype iconic BNF for traditional programming languages, and implemented it for one particular language: C++. The research provided

the methodology and some algorithm on traditional programming languages' visualization. However, because it is a prototype, more refining works needs to be done. Since the limitation of time, so much compromise has been made that the flexibility of the free-form style presented in this research is not clear enough. So, more research efforts should be done to explore the advantage of the free-form style in the future.

The iconic BNF should be applied to other traditional programming languages. The environment should be modified to generate multiple target languages, and the issues about how different target languages work together in the same environment needs to be discussed.

The visualization approach presented in this thesis focus on the procedural level. For some computer languages, such as the object-oriented programming languages, the control flow doses not play a central role as it does in those procedural programming languages. For an object-oriented programming language, the relations of objects are also very important. So, the visualization on aspects besides the procedural flow deserves further discussion.

A comparison between the parser designed for this research and the DR parser is a potential topic. The DR parser has not been fully implemented with respect to a traditional language. The primary difficulty is how to construct the scanner in a way which can implement the scanning theory proposed by Positional Grammar. The approach of this research was to divide the scanning into two steps, a method which is more environment specific, but easier to implement.

The environment created in this research allows for free-form construction of diagrams, unlike BACCII++ and STRIDES projects which embed the parse tree into the actual programming. The comparison between these two approaches, the advantages and disadvantages, will enhance the research efforts in this area.

## REFERENCES

- [1] D. C. Smith, (1975). "Pygmalion: A Creative Programming Environment," Ph.D. dissertation, Dept. of Computer Science Stanford University (tech. Report STAN-CS-75-499)
- [2] D. C. Smith, (1990). "Pygmalion: A Computer Program to Model and Stimulate Creative Thought," *Visual Programming Environments: Paradigms and Systems*, IEEE Computer Society Press Tutorial, pp. 216-251.
- [3] E. P. Clinert, and S. L. Tanimoto, (1984). "Pict: An Interactive Graphical Programming Environment," *IEEE Computer* November pp.7-24.
- [4] F. Zhang, and B. A. Calloni, (1997). "Requirements Definition Document on STRIDES," Texas Tech University, November 4, 1996.
- [5] B. A. Calloni, (1992). "BACCII: An Iconic, Syntax-directed Windows System for Teaching Procedural Programming," M.S. Thesis, Dept. of Computer Science, Texas Tech University.
- [6] E. J. Golint, and S. P. Reiss, (1989). "The Specification of Visual Language Syntax," *IEEE Workshop on Visual Language*, IEEE Press pp105-110.
- [7] P. D. Vigna, and C. Ghezzi, (1978). "Context-free Graph Grammars," *Information and Control*, vol.37, pp. 207-233.
- [8] J. Feder, (1971). "Plex Languages," *Information Science*, vol.3, pp.225-241.
- [9] F. Ferrucci, G. Pacini, and G. Tortora, (1991). "Efficient Parsing of Multidimensional Structures," *Proc. IEEE Workshop on Visual Languages*, Kobe Japan, Oct. pp. 105-110.
- [10] R. Helm, K. Marriott and M. Odersky, (1991). "Building Visual Language Parsers," *Human Factors in Computing Systems: CHI '91 Conference Proceedings*, Amsterdam: Addison-Wesley, pp.145-158.
- [11] K. Wittenburg, and L. Weitzman, (1990). "Visual Grammars and Incremental Parsing for Interface Languages," in *Procs. Of the IEEE Workshop on Visual Languages*, Skokie, Illinois, pp.111-118.
- [12] C. Crimi, A. Guercio, G. Nota, G. Pacini, G. Tortora, and M. Tucci, (1991). "Relation Grammars and their Application to Multi-dimensional Language." *Journal of Visual Languages and Computing*, Academic Press, 2 pp. 333-346.

- [13] J. R. Rasure, and C. S. Williams, (1991). "An Integrated Data Flow Visual Language and Software Development Environment," *Journal of Visual Languages and Computing* , 2 pp. 217-246.
- [14] C. S. Williams, and J. R. Rasure, (1990). "A Visual Language for Image Processing ," *IEEE Workshop on Visual Languages*, Skokie, Illinois, 4-6 October, pp. 86-91.
- [15] A. L. Davis, and R. M. Keller, (1982). "Data Flow Program Graphs," *IEEE Computer* 15(2) pp. 26-41.
- [16] G. Costagliola, (1995). "VLCC: An Visual Languages' Compiler's Compiler," *IEEE Computer* March, pp. 56-66.
- [17] G. Costagliola, and S. K. Chang, (1990). "DR PARSERS: A Generalization of LR Parsers," *Proc. of IEEE Workshop on Visual Languages*, Skokie, Illinois, 4-6 October, pp.174-180.
- [18] G. Costagliola, A. D. Lucia, and S. Orefice, (1994). "Twoards Efficient Parsing of Diagrammatic Languages," *Proc. of International Workshop on Advanced Visual Interface*, Bari, Italy.
- [19] G. Costagliola, S. orefice, G. Polese, M. Tucci, and G. Tortora, (1993). "Automationg Parser Generation for Pictorial Languages," *Proc. of IEEE Symposium on Visual Languages* pp. 306-313.
- [20] A. V. Aho, R. Sethi, and J. D. Ullman, (1986). *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison Wesley.

## APPENDIX A

### THEORY AND ALGORITHM ABOUT LR PARSER [20]

#### A.1 FOLLOW

Define Follow(A), for non-terminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form  $S \Rightarrow \alpha A a \beta$  for some  $\alpha$  and  $\beta$ . Note that there may, at some time during the derivation, have been symbols between A and a, but if so, they derived  $\epsilon$  and disappeared. If A can be the rightmost symbol in some sentential form, then  $\$$  is in FOLLOW(A). To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place  $\$$  in FOLLOW(S), where S is the start symbol and  $\$$  is the input right endmarker.
2. If there is a production  $A \rightarrow \alpha B \beta$ , then everything in FIRST( $\beta$ ) except for  $\epsilon$  is placed in FOLLOW(B).
3. If there is a production  $A \rightarrow \alpha B$ , or a production  $A \rightarrow \alpha B \beta$  where FIRST( $\beta$ ) contains  $\epsilon$  (i.e.,  $\beta \Rightarrow \epsilon$ ), then everything in FOLLOW(A) is in FOLLOW(B).



## A.2 The Closure Operation

If  $I$  is a set of items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of items constructed from  $I$  by the two rules:

1. Initially, every item in  $I$  is added to  $\text{closure}(I)$ .
2. If  $A \rightarrow \alpha \bullet B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production, then add the item  $B \rightarrow \bullet \gamma$  to  $I$ , if it is not already there. We apply this rule until no more new items can be added to  $\text{closure}(I)$ .

## A.3 The Goto Operation

The second useful function is  $\text{goto}(I, X)$  where  $I$  is a set of items and  $X$  is a grammar symbol.  $\text{Goto}(I, X)$  is defined to be the closure of the set of all items  $[A \rightarrow \alpha X \bullet \beta]$  such that  $[A \rightarrow \alpha \bullet X\beta]$  is in  $I$ . Intuitively, if  $I$  is the set of items that are valid for some viable prefix  $\gamma$ , then  $\text{goto}(I, X)$  is the set of items that are valid for the viable prefix  $\gamma X$ .

## A.4 Algorithm to Construct Canonical LR(0) Collection

If  $G$  is a grammar with start symbol  $S$ , then  $G'$ , the augmented grammar for  $G$ , is  $G$  with a new start symbol  $S'$  and production  $S' \rightarrow S$ . The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by  $S' \rightarrow S$ .

**procedure** items( $G'$ );

**begin**

$C := \{\text{closure}(\{[S' \rightarrow \bullet S]\})\};$

**repeat**

**for** each set of items  $I$  in  $C$  and each grammar symbol  $X$  such  
that  $\text{goto}(I, X)$  is not empty and not in  $C$  **do** add  $\text{goto}(I, X)$  to  $C$

**until** no more sets of items can be added to  $C$

**end**

#### A.5 Algorithm 4.8. Constructing an SLR parsing table

Input. An augmented grammar  $G'$

Output. The SLR parsing table functions *action* and *goto* for  $G'$

Method.

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of LR(0) items for  $G'$ .
2. State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follows:
  - a. If  $[A \rightarrow \alpha \bullet a\beta]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift  $j$ ."  
Here  $a$  must be a terminal.
  - b. If  $[A \rightarrow \alpha \bullet]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{FOLLOW}(A)$ ; here  $A$  may not be  $S'$ .
  - c. If  $[S' \rightarrow S \bullet]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to "accept".

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state  $I$  are constructed for all non-terminals  $A$  using the rule: If  $\text{goto}[I, A] = j$
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow \bullet S]$ .

The parsing table consisting of the parsing action and goto functions determined by Algorithm 4.8 is called the SLR(1) table for  $G$ .

## APPENDIX B

### THEORY AND ALGORITHM ABOUT DR PARSER[17]

#### B.1 POSFOLLOW

In the following, we will always refer to  $POS=\{SP, HOR, VER, ANY\}$ , meaning with this that the next symbol is to be searched in the highest position in the next column on the right (HOR) or in the leftmost position in the next row below (VERO), or in any position (ANY). This last operator is used when we want to make sure that there are no symbols to scan in any position.

POSFOLLOW is defined similarly to FOLLOW, but it considers only the positional operators. It can be defined as follows:

1. Place ANY in POSFOLLOW(S), where S is the starting symbol;
2. If there is a production " $A := x B \text{ pos } y$ " then pos is placed in POSFOLLOW(B) where  $B \in N$ ;
3. If there is a production " $A := x B$ ", then everything in POSFOLLOW(A) is also in POSFOLLOW(B) where  $A, B \in N$ .

#### B.2 SDR Parsing Table Algorithm

Input. An augmented grammar  $G'$ ;

Output. The SDR parsing table functions action, goto and position for  $G'$ .

Method.

1. Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of sets of DR(0) items for  $G'$ .
2. State  $i$  is constructed from  $I_i$ . The parsing actions and positions for state  $i$  are determined as follows:
  - a. Each time  $A := [x \bullet p \ y]$  is in  $I_i$ , add  $p$  to position  $[I_i]$ .  $x$  and  $y$  are in  $\{N \cup T \cup \text{POS}\}$
  - b. If  $[A := x \bullet p \ a \ y]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[I_i, a]$  to "shift  $j$ " (" $s_j$ "). Here ' $a$ ' must be a terminal. Note that  $p$  is ignored.
  - c. If  $[A := x \bullet ]$  is in  $I_i$ , then set  $\text{action}[i, a]$  to "reduce  $A := x$ " for all ' $a$ ' in  $\text{FOLLOW}(A)$  and add  $\text{POSFOLLOW}(A)$  to  $\text{position}[I_i]$ . Here  $A$  may not be  $S'$ .
  - d. If  $[S' := SP \ S \ .]$  is in  $I_i$ , then set  $\text{action}[i, \$]$  to "accept" and add ANY to  $\text{position}[i]$ .

If any conflicting actions or positions are generated by the above rules, we say the grammar is not SDR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state  $I$  are constructed for all non-terminals  $A$  using the rule: If  $\text{goto}(I_i, A) = I_j$ , then  $\text{goto}[i, A] = j$ .
4. All entries not defined by rules (2) and (3) are made "error".
5. The initial state of the parser is the one constructed from the set of items containing  $[S' := SP \ S]$ .

A grammar having an SDR(1) parsing table is said to be SDR(1).

## APPENDIX C

### THE TRANSITION DIAGRAM AND SLR PARSING TABLE CONSTRUCTED IN THIS RESEARCH

#### C.1 The Augmented Grammar G'

flowchart' = flowchart  
flowchart = START A compd down END  
compd = compd down sent | sent  
sent = condition | loop | STATE  
loop = TEST A compd C  
condition = single | multi  
single = IF branch branch CONET  
branch = SLASH B compd down  
multi = SWITCH casepart CONET  
casepart = onecase casepart | onecase onecase  
onecase = DASH A CASE A compd down  
down = A | B

#### C.2 The Canonical LR(0) Collection for G'

I<sub>0</sub>:    flowchart' = . flowchart  
         flowchart = . START A compd down END

I<sub>1</sub>:    flowchart = flowchart .

I<sub>2</sub>:    flowchart = START . A compd down END

I<sub>3</sub>:    flowchart = START A . compd down END  
         compd = . compd down sent  
         compd = . sent  
         sent = . STATE  
         sent = . loop  
         sent = . condition  
         loop = . TEST A compd C  
         condition = . single  
         condition = . multi  
         single = . IF branch branch CONET  
         multi = . SWITCH casepart CONET

I<sub>4</sub>:    flowchart = START A compd . down END  
          compd = compd . down sent  
          down = . A  
          down = . B

I<sub>5</sub>:    compd = sent .

I<sub>6</sub>:    sent = STATE .

I<sub>7</sub>:    sent = loop .

I<sub>8</sub>:    sent = condition .

I<sub>9</sub>:    loop = TEST . A compd C

I<sub>10</sub>:   condition = single .

I<sub>11</sub>:   condition = multi .

I<sub>12</sub>:   single = IF . branch branch CONET  
          branch = . SLASH B compd down

I<sub>13</sub>:   multi = SWITCH . casepart CONET  
          casepart = . casepart onecase  
          casepart = . onecase onecase

I<sub>14</sub>:   flowchart = START A compd down . END  
          compd = compd down . sent

I<sub>15</sub>:   down = A .

I<sub>16</sub>:   down = B .

I<sub>17</sub>:   loop = TEST A . compd C  
          compd = . compd down sent  
          compd = . sent  
          sent = . STATE  
          sent = . loop  
          sent = . condition  
          loop = . TEST A compd C  
          condition = . single  
          condition = . multi  
          single = . IF branch branch CONET  
          multi = . SWITCH casepart CONET

I<sub>18</sub>:    single = IF branch . branch CONET  
           branch = . SLASH B compd down  
  
 I<sub>19</sub>:    branch = SLASH . B compd down  
  
 I<sub>20</sub>:    multi = SWITCH casepart . CONET  
           casepart = casepart . onecase  
           onecase = . DASH A CASE A compd down  
  
 I<sub>21</sub>:    casepart = onecase . onecase  
           onecase = . DASH A CASE A compd down  
  
 I<sub>22</sub>:    onecase = DASH . A CASE A compd down  
  
 I<sub>23</sub>:    flowchart = START A compd down END .  
  
 I<sub>24</sub>:    compd =compd down sent .  
  
 I<sub>25</sub>:    loop = TEST A compd . C  
           compd =compd . down sent  
           down = . A  
           down = . B  
  
 I<sub>26</sub>:    single = IF branch branch . CONET  
  
 I<sub>27</sub>:    branch = SLASH B . compd down  
           compd = . compd down sent  
           compd = . sent  
           sent = . STATE  
           sent = . loop  
           sent = . condition  
           loop = . TEST A compd C  
           condition = . single  
           condition = . multi  
           single = . IF branch branch CONET  
           multi = . SWITCH casepart CONET  
  
 I<sub>28</sub>:    multi = SWITCH casepart CONET.  
  
 I<sub>29</sub>:    casepart = casepart onecase .  
  
 I<sub>30</sub>:    casepart = onecase onecase .



I<sub>31</sub>:    onecase = DASH A . CASE A compd down

I<sub>32</sub>:    loop = TEST A compd C .

I<sub>33</sub>:    compd = compd down . sent  
          sent = . STATE  
          sent = . loop  
          sent = . condition  
          loop = . TEST A compd C  
          condition = . single  
          condition = . multi  
          single = . IF branch branch CONET  
          multi = . SWITCH casepart CONET

I<sub>34</sub>:    single = IF branch branch CONET

I<sub>35</sub>:    branch = SLASH B compd . down  
          compd = compd . down sent  
          down = . A  
          down = . B

I<sub>36</sub>:    onecase = DASH A CASE . A compd down

I<sub>37</sub>:    branch = SLASH B compd down .  
          compd = compd down . sent  
          sent = . STATE  
          sent = . loop  
          sent = . condition  
          loop = . TEST A compd C  
          condition = . single  
          condition = . multi  
          single = . IF branch branch CONET  
          multi = . SWITCH casepart CONET

I<sub>38</sub>:    onecase = DASH A CASE A . compd down  
          compd = . compd down sent  
          compd = . sent  
          sent = . STATE  
          sent = . loop  
          sent = . condition  
          loop = . TEST A compd C  
          condition = . single  
          condition = . multi  
          single = . IF branch branch CONET

multi = . SWITCH casepart CONET

I<sub>39</sub>: onecase = DASH A CASE A compd . down  
compd = compd . down sent  
down = . A  
down = . B

I<sub>40</sub>: onecase = DASH A CASE A compd down .  
compd = compd down . sent  
sent = . STATE  
sent = . loop  
sent = . condition  
loop = . TEST A compd C  
condition = . single  
condition = . multi  
single = . IF branch branch CONET  
multi = . SWITCH casepart CONET

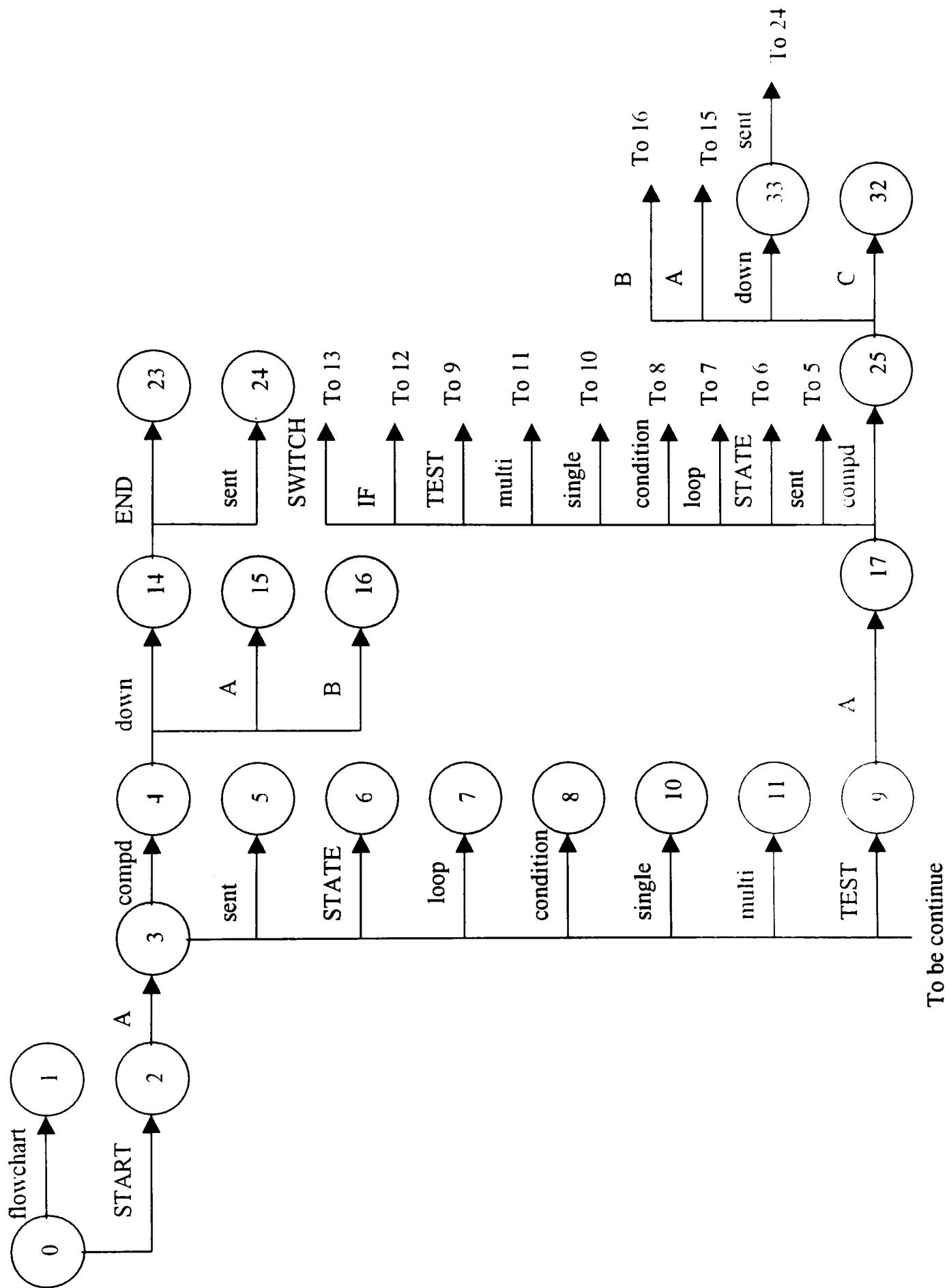


Fig. C.1. The Transition Diagram Based on the Canonical Collection

continue

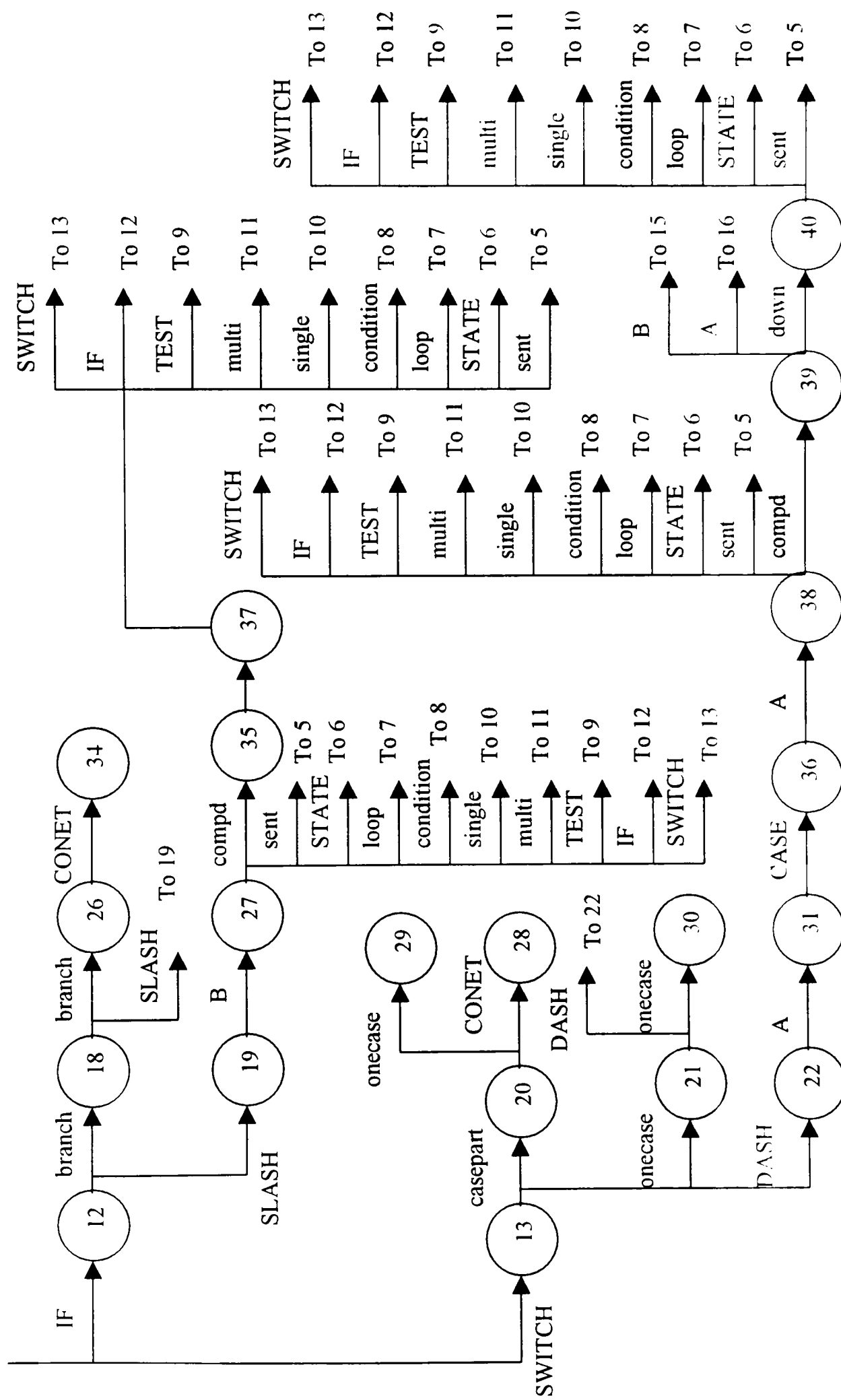


Fig. C.1 Continued

Table C.1 The SLR Parsing Table Based on the Canonical Collection and the Transition Diagram

	action														goto										
state	START	END	STATE	IF	SWITCH	TEST	ONET	CASE	A	B	C	SLASH	DASH	\$	flowchart	compd	sent	loop	condition	single	multi	branch	casepart	oncase	down
0															1										
1	s2													acc											
2									s3																
3			s6	s12	s13	s9										4	5	7	8	10	11				
4									s15 s16																14
5									r2	r2															
6									r3	r3															
7									r3	r3															
8									r3	r3*															
9									s17																
10									r5	r5*															
11									r5	r5*															
12												s19										18			
13													s22									20		21	
14																	24								
15		s23							r11	r11															
16			r11	r11	r11	r11	r11	r11																	
17				s6	s12	s13	s9									25	5	7	8	10	11				
18												s19										26			
19									s27																
20								s28																	29

Table C.1 Continued

action														goto											
state	START	END	STATE	IF	SWITCH	TEST	CONET	CASE	A	B	C	SLASH	DASH	\$	flowchart	compd	sent	loop	condition	single	multi	branch	casepart	onecase	down
21																									30
22									s31					s22											
23														r1											
24									r2	r2															
25									s15	s16	s32														32
26								s34																	
27			s6	s12	s13	s9										35	5	7	8	10	11				
28									r8	r8*															
29							r9																		
30							r9																		
31								s36																	
32									r4*	r4															
33																	24								
34									r6	r6*															
35									s15	s16															37
36									s38																
37			s6	s12	s13	s9	r7						r7				24	7	8	10	11				
38			s6	s12	s13	s9										39	5	7	8	10	11				
39										s15	s16														40
40			s6	s12	s13	s9	r10										24	7	8	10	11				

