INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Semantic Analysis and SIPL AST Translator Generation in the GIPSY

Ai Hua Wu

A Thesis

in the

Department of Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

December 2002

© Ai Hua Wu, 2002

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-77724-3

Canada

Abstract

Semantic Analysis and SIPL AST Translator Generation in the GIPSY

Ai Hua Wu

The GIPSY system is the implementation of Intensional Programming, which is a new programming paradigm based on intensional logic. Semantic Analysis is the intermediate between the parsers of the system and the run-time system. The objective of this work is to generate an intermediate representation to be fed to the eduction engine. Because of the specific character of the Lucid language, we consider the elimination of the functions in the original abstract syntactic tree before semantic analysis. Then, we can build a dictionary in which includes each identifier with its attributes. At the same time, we can build a bridge between SIPL and GIPL. A SIPL AST Translator Generation is developed, which can make semantic analysis only focus on GIPL, and will enable an easy implementation of other variations of the Lucid language or other languages of intensional nature. Finaly, we successfully integrate the semantic analyzer, SIPL and GIPL parsers, SIPL AST translator generation in the overall GIPSY system.

Acknowledgements

I would like to express my sincere gratitude to my supervisor – Dr. Joey Paquet, for accepting me as his student, introducing me into this field and providing me financial support. I am also thankful to Dr. Peter Grogono, for giving me much valuable advice. Without their contribution, this thesis would not have been possible. Thanks also to all GIPSY project teammates for their generous help and efforts, and overall collaboration. Specially, I would like to thank my dearest parents, my sisters and my brother. Their love has always been a great encouragement for me.

Table of Contents

1. Introduction 1
- 1.1 BACKGROUND 1
- 1.2 LIMITATIONS OF PREVIOUS WORK 2
- 1.3 ARCHITECTURE OF THE GIPSY SYSTEM 3
  - 1.3.1 General Intensional Programming Compiler (GIPC) 4
  - 1.3.2 Run-time Interactive Programming Environment (RIPE) 6
  - 1.3.3 General Eduction Engine (GEE) 8
- 1.4 CONTRIBUTIONS 10
- 1.5 STRUCTURE OF THE THESIS 11

2. Semantic Analysis 12
- 2.1 BUILDING A DICTIONARY 12
  - 2.1.1 General introduction 12
  - 2.1.2 Main data structure 15
  - 2.1.3 Algorithm 16
  - 2.1.4 Error reporting and error recovery 20
- 2.2 TYPE CHECKING 21
  - 2.2.1 General Introduction 21
  - 2.2.2 Type checking rules 22
  - 2.2.3 Main data structure 22
  - 2.2.4 Algorithm 23
  - 2.2.5 Error reporting and error recovery 28
- 2.3 RANK ANALYSIS 29
  - 2.3.1 General introduction 30
  - 2.3.2 Rank analysis rules 31
  - 2.3.3 Algorithm 31
  - 2.3.4 Error reporting and error recovery 36
- 2.4 SEMANTIC ANALYSIS: SECOND PASS 37
- 2.5 SUMMARY 38

3. Function Elimination 39
- 3.1 INTRODUCTION 39
- 3.2 FUNCTION TABLE CONSTRUCTION 40
  - 3.2.1 Use general symbols to replace parameters in the original function .. 41
  - 3.2.2 Building the function table 44
- 3.3 FUNCTION ELIMINATION 45
  - 3.3.1 Using actual parameters to replace general symbols 46
  - 3.3.2 General function elimination 47
  - 3.3.3 Function elimination with new variable definitions 49
- 3.4 ERROR REPORTING AND ERROR RECOVERY 52

3.5 SUMMARY 53

4. SIPL AST Translator Generation 54

4.1 INTRODUCTION 54
4.2 THE STYLE DEFINITION FOR USER INPUT 57
4.3 THE SIPL TRANSLATOR GENERATION SYNTACTICAL SPECIFICATION 58
4.3.1 Lexical conventions 59
4.3.2 Productions of Grammar 59
4.4 BUILDING OPERATOR TRANSLATION TABLE 61
4.4.1 Table-Driven Parsing 61
4.4.2 Building a Forest for Operators 63
4.5 SPECIFIC OPERATOR TRANSLATION 66
4.6 ERROR REPORTING AND ERROR RECOVERY 68
4.7 SUMMARY 70

5. Result Evaluation 71

5.1 SEMANTIC ANALYSIS 72
5.2 FUNCTION ELIMINATION 77
5.3 SIPL AST TRANSLATOR 80
5.4 SUMMARY 84

6. Integration with other components in GIPSY 85

6.1 INTERGRATE WITH THE PARSER 85
6.2 RELATIONSHIP WITH THE RIPE 87
6.3 OUTPUT FOR THE GEE 87
6.4 SUMMARY 87

7. Future Work 88

7.1 PERMIT NESTED INPUT IN THE SIPL TRANSLATOR PART 88
7.2 RECURSION IN LUCID PROGRAMS 89
7.3 GRAPHIC INTERFACE DESIGN FOR GIPC 89

8. Bibliography 90

9. Appendix I: GIPL AST structures 91

10. Appendix II: Table used for Lexical in Auto-translator 92

11. Appendix III: Table used for Syntactic in Auto-translator 93

12. Appendix IV: First/Fellow sets in Auto-translator 94

List of Tables

TABLE 1-1 INDEXICAL LUCID PROGRAM FOR THE HAMMING PROBLEM 7

TABLE 2-1 AN EXAMPLE OF LUCID PROGRAM 13

TABLE 2-2 AN ILLEGAL EXAMPLE OF LUCID PROGRAM 14

TABLE 2-3 A LEGAL EXAMPLE OF LUCID PROGRAM 14

TABLE 2-4 THE RULES USED IN THE TYPE CHECKING 22

TABLE 2-5 THE SEMANTIC STACK 23

TABLE 2-6 THE SEMANTIC STACK AFTER BUILDING THE DICTIONARY 25

TABLE 2-7 THE DICTIONARY AFTER TYPE CHECKING ON THE SUB-TREE OF Y 27

TABLE 2-8 THE DICTIONARY AFTER TYPE CHECKING 28

TABLE 2-9 ERROR INFORMATION IN THE TYPE CHECKING 29

TABLE 2-10 A LUCID PROGRAM WITH DIMENSION DECLARATION 31

TABLE 2-11 THE RULES USED IN THE RANK ANALYSIS 31

TABLE 2-12 THE DICTIONARY AFTER ANALYZING "Z=3" 33

TABLE 2-13 THE DICTIONARY AFTER ANALYZING "X=Z+1" 34

TABLE 2-14 THE FINAL DICTIONARY 36

TABLE 3-1 THE MEANING OF FUNCTION ELIMINATION 39

TABLE 3-2 A LUCID PROGRAM WITH A FUNCTION 41

TABLE 3-3 THE BRANCH AST OF THE FUNCTION DEFINITION 42

TABLE 3-4 THE GENERALIZED FUNCTION DEFINITION BRANCH 44

TABLE 3-5 THE AST AFTER ELIMINATING FUNCTION 45

TABLE 3-6 THE SUBSTITUTE OF FUNCTION TWOAT( ) 47

TABLE 3-7 A SIMPLE LUCID PROGRAM WITH FUNCTION 47

TABLE 3-8 CHANGE PROCESS DURING GENERAL FUNCTION ELIMINATION 49

TABLE 3-9 CHANGE PROCESS DURING SPECIFIC FUNCTION ELIMINATION 52

TABLE 3-10 ERROR INFORMATION IN FUNCTION ELIMINATION 53

TABLE 4-1 A SIPL PROGRAM AND ITS TRANSLATION 56

TABLE 4-2 AN EXAMPLE OF THE INPUT FILE 58

TABLE 4-3 THE GRAMMAR OF THE SIPL TRANSLATOR GENERATION 60

TABLE 4-4 RULES TO INSERTTING OPERATION AND ITS NUMBER 62

TABLE 4-5 ERROR INFORMATION IN THE TRANSLATOR GENERATION 69

TABLE 5-1 THE LUCID PROGRAM AND ITS TEST RESULT OF TEST 1 73

TABLE 5-2 THE LUCID PROGRAM AND ITS TEST RESULT OF TEST 2 74

TABLE 5-3 THE LUCID PROGRAM AND ITS TEST RESULT OF TEST 3 75

TABLE 5-4 THE LUCID PROGRAM AND ITS TEST RESULT OF TEST 4 76

TABLE 5-5 THE SEMANTIC ANALYSIS PROCESS OF A LUCID PROGRAM WITH FUNCTIONS 79

TABLE 5-6 THE RESULT OF APPLYING THE SIPL AST TRANSLATOR 83

TABLE 7-1 THE TRANSLATION FILE WITH NESTING 88

List of Figures

FIGURE 1-2 THE HIGH-LEVEL ARCHITECTURE OF GIPSY 4

FIGURE 1-2 THE ARCHITECTURE OF THE COMPILER MODULE (GIPC).......... 5

FIGURE 1-3 DATAFLOW GRAPH FOR THE HAMMING PROBLEM 8

FIGURE 2-1 AN EXAMPLE OF LUCID PROGRAM AND ITS AST 12

FIGURE 2-2 A LUCID PROGRAM AND ITS AST 16

FIGURE 2-3 THE HASHTABLE AND DICTIONARY AFTER TRAVERSING THE SUB-TREE1 17

FIGURE 2-4 THE NEW AST AFTER TRAVERSING THE SUB-TREE2 18

FIGURE 2-5 THE HASHTABLE AND DICTIONARY AFTER TRAVERSING THE SUB-TREE2 18

FIGURE 2-6 THE RESULT OF THE AST AFTER TRAVERSING THE SUB-TREE3.......... 19

FIGURE 2-7 THE HASHTABLE AND DICTIONARY AFTER TRAVERSING THE SUB-TREE3 20

FIGURE 2-8 THE AST OF THE LONG EXPRESSION.......... 23

FIGURE 2-9 THE EXAMPLE USED TO DO THE TYPE CHECKING 24

FIGURE 2-10 THE NEW AST AFTER TYPE CHECKING ON THE SUB-TREE OF Y 27

FIGURE 2-11 THE FINAL AST AFTER TYPE CHECKING.......... 28

FIGURE 2-12 RANK[A] = {X, Y}.......... 30

FIGURE 2-13 RANK [A+3] = {X, Y} U Ø = {X, Y} 30

FIGURE 2-14 THE EXAMPLE USED TO DO THE RANK ANALYSIS 32

FIGURE 2-15 THE CORRESPONDING AST AFTER ANALYZING "Z=3" 33

FIGURE 2-16 THE CORRESPONDING AST AFTER ANALYZING "X=Z+1" 34

FIGURE 2-17 THE CORRESPONDING AST AFTER ANALYZING THE @ EXPRESSION. 35

FIGURE 2-18 THE FINAL AST 36

FIGURE 2-19 A TYPICAL LUCID PROGRAM THAT NEED THE SECOND SEMANTIC ANALYSIS 37

FIGURE 3-1 THE CHANGE PROCESS ILLUSTRATION 48

FIGURE 3-2 THE CHANGE PROCESS ILLUSTRATION 50

FIGURE4-1 THE ARCHITECTURE OF SIPL AST TRANSLATOR GENERATION 57

FIGURE 4-2 THE FOREST OF TRANSLATION OPERATORS 66

FIGURE 4-3 SLICE AFTER REPLACING THE GENERAL OPERAND WITH THE REAL OPERAND 68

FIGURE 6-1 THE RELATIONSHIP OF THE SEMANTIC ANALYZER AND OTHER COMPONENTS 85

List of Code Excerpts

EXCERPT 2-1 THE STRUCTURE OF ELEMENTS IN THE DICTIONARY 15

EXCERPT 3-1 THE METHOD TO REALIZE THE FUNCTION PARAMETERS GENERALIZED 43

EXCERPT 3-2 THE STRUCTURE OF ITEMS IN THE FUNCTION TABLE 44

EXCERPT 3-3 ELIMINATING THE BRANCH OF FUNCTION DEFINITION 45

EXCERPT 4-1 PROGRAM THAT EXPLAINS "~" IS AT THE BEGINNING OF A RULE 61

EXCERPT 4-2 PROGRAM THAT EXPLAINS "~" IS IN THE MIDDLE OF A RULE............ 62

EXCERPT 4-3 DATA STRUCTURE OF MANAGING OPERATORS AND THEIR AST........ 63

EXCERPT 4-4 THE CREATEHASH() METHOD 64

EXCERPT 4-5 THE DOCOND() METHOD 65

EXCERPT 4-6 PROGRAM TO REALIZE TRANSLATION 68

EXCERPT 6-1 MODIFICATION FOR INTEGRATING THE SEMANTIC ANALYZER WITH THE PARSER 86

1. Introduction

This thesis is mainly about semantic analyzer and the SIPL AST translator generation in the GIPSY (General Intensional Programming System). GIPSY is an implementation of Intensional Programming, which involves the programming of expressions placed in an inherent multidimensional context space [1]. GIPSY is also a very ambitious project and is directed by Dr. Joey Paquet and Dr. Peter Grogono in the Computer Science Department at Concordia University, Montreal, Canada. There are currently two PhD students and four Msc students working on the various parts of the project.

1.1 Background

In all scientific domains, it is common to do computer simulations that normally correspond to the operational version of a set of differential equations. These intrinsically multidimensional and intensional equations allow complex physical phenomena such as plasma physics to be represented very naturally [1].

Conventional programming languages (e.g. Fortran, C, Java, etc.) can be used to solve problems of a multidimensional nature, such as particle in-cell simulation of plasma using differential equations. However, the programming of such equations in these languages does not reflect their original simplicity. The situation gets worse when programming tensor equations. [1]

Some mathematical programming languages and environments such as Matlab and Maple allow the natural expression of differential equations. However, none of these systems enable the high-speed execution of the equation system because a parallel code generation facility is not provided. Moreover, none of mathematical programming languages and environments allows the natural expression of tensor equations. [1]

It has been proven in [1] that intensional programming can be used to build programs to solve such problems and to achieve the high-performance parallel computation of differential or tensor equations expressed in a natural manner. However, intensional programming is in its early stages of development, and recent history has proven that it is still an area that is extremely evolutionary and of extremely general application. To cope with these changing and general needs, we need a system whose architecture is extremely flexible and adaptable. [1]

The GIPSY project proposes the design of a general intensional programming system. Programmers today have a strong tendency to use mainstream solutions to solve their problems. Aside from its use of sequential threads, the GIPSY system is far from a mainstream solution. One of our main goals is to design a system that effectively demonstrates that intensional programming can be used as an effective solution to solve problems of intensional nature, and to efficiently develop parallel programs through code reuse. [1,5]

1.2 Limitations of Previous Work

Lucid is a multidimensional intensional programming language whose semantics is based on the possible world semantics of intensional logics. Many very different languages were part of the ancestors and offsprings of Lucid. For example, GLU (Granular Lucid), uses sequential threads as parallel computing units. Tensor Lucid, enables the expression of mathematical equations, up to tensor expressions. [1]

Intensional programming (in the sense of Lucid) has been successfully applied to topics as diverse as reactive programming, software configuraton [4], tensor programming [1] and distributed operating systems [3]. However, these projects have all been developed in isolation.

The GLU parallel/distributed programming environment, developed at Stanford Research Institute (SRI) in Menlo Park, was the first intensional programming system that enabled the compilation of Indexical Lucid programs, together with the use of sequential threads written in C. It has proven to be a usable and highly efficient solution for the parallelization of sequential programs. However, the GLU system suffered from a lack of flexibility and adaptability. It could not cope with the latest evolution of Lucid. For example, the GLU system does not enable dimensions and functions as first-class values, which is one of the key principles used in Tensor Lucid. [1]

Consequently, new tools for intensional programming are required. One possible option solution is GIPSY. The design and implentation of GIPSY is done towards generality, flexibility and efficiency.

1.3 Architecture of the GIPSY System

A key achievement is to define a clear and adaptive architecture in order to reach the flexibility and adaptability goals of the system. The system is composed of three main subsystems: general intensional programming language compiler (GIPC), general eduction engine (GEE) and run-time interactive programming environment (RIPE). All three components are designed in a modular manner to permit the eventual replacement of each component. This improves the overall efficiency of the system. The organization of the GIPSY system is depicted in Figure1-1 [2].

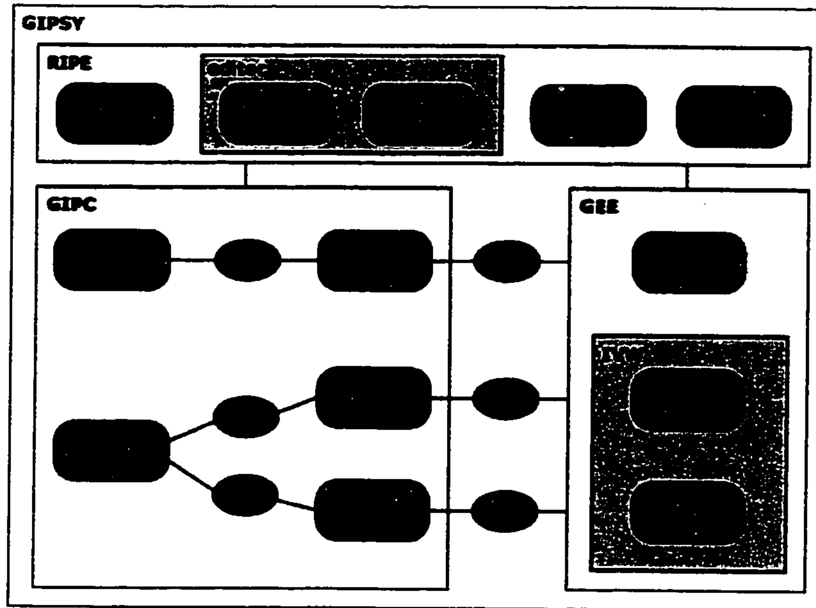


Figure 1-1 The high-level architecture of GIPSY

(**RIPE**: Run-time interactive programming environment)
(**GIPC**: General intensional programming language compiler)
(**GEE**: General eduction engine)
(**AST**: Abstract syntax tree)
(**IDP**: Intensional demand propagator)
(**IVW**: Intensional value warehouse)
(**ST**: Sequential thread)
(**CP**: Communication procedure)
(**DPR**: Demand propogation resources)

1.3.1 General Intensional Programming Compiler (GIPC)

GIPSY programs are compiled in a two-stage process. First, the intensional part of the GIPSY program is translated into Java; then, the resulting Java program is compiled in the standard way. The structure of the GIPC part can be seen in Figure1-2. The task of the thsis is to find the solutions to the parts with arrows.

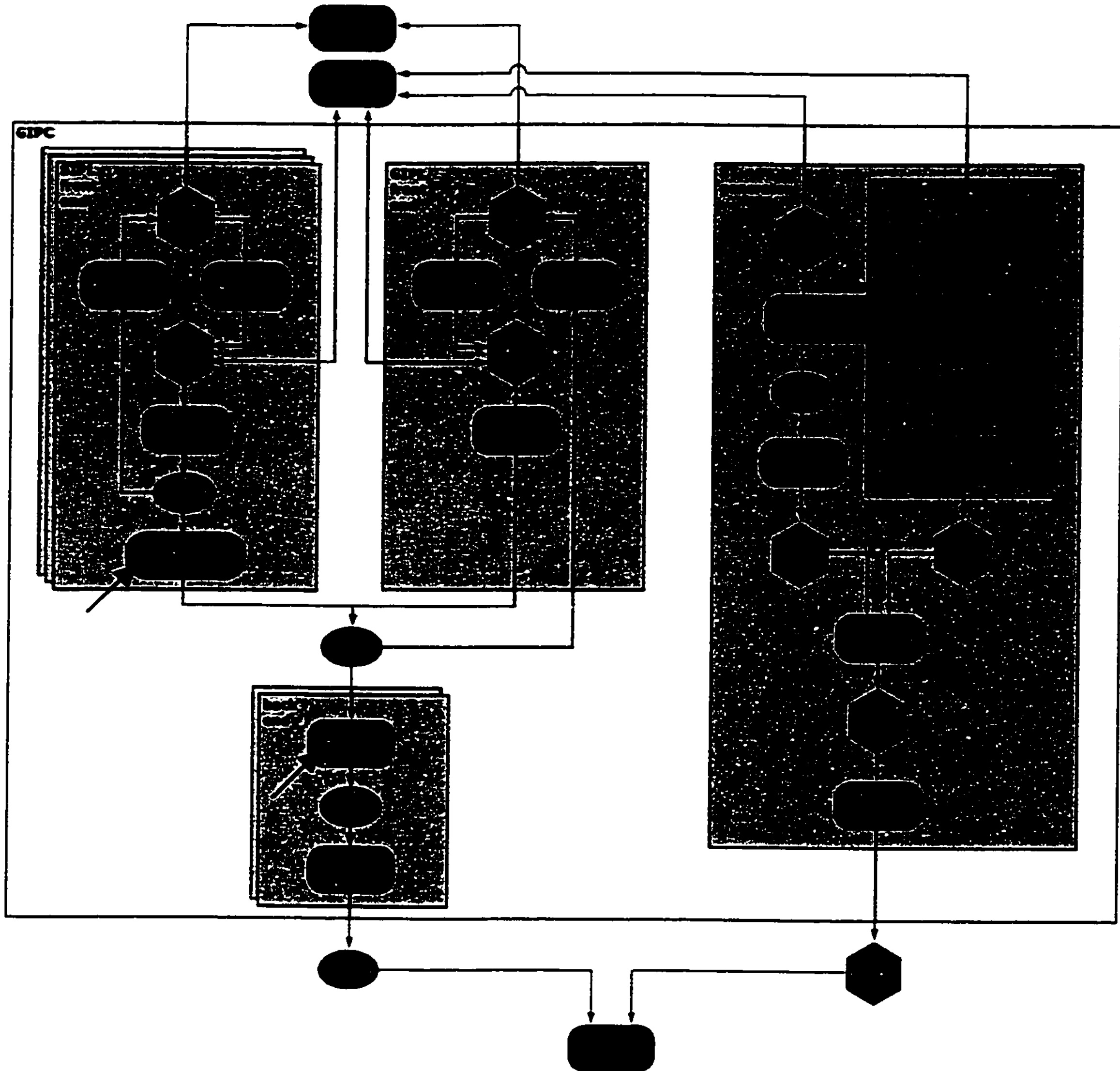


Figure 1-2 The architecture of the compiler module (GIPC)

It was proven in [1] that all specific intensional programming languages (SIPL) currently known share the semantics of the generic intensional programming language (GIPL) with only the @ and # operators. It means that the GIPL is the core language and all other SIPL can be translated into @ and # primitives. So, there are two kinds of front ends in the GIPC: a SIPL front end for each SIPL, and a single GIPL front end.

However, the semantic analyzer checks a tree of the program translated to GIPL primitives, no matter in what SIPL it was written, or if it was written in GIPL itself. So, there is SIPL-GIPL AST translator in all SIPL front ends.

This structure is interpreted at run-time by the GEE following the demand propagation mechanism. Data communication procedures used in a distributed evaluation of the program are also generated by the GIPC according to the data structure definitions written in the GIPSY program. This yields a set of intensional communication procedures (ICP). These are generated following a given communication layer definition such as provided by IPC, CORBA or the WOS.

The sequential functions defined in the right part of the GIPSY program are translated into sequential code using the second stage (Java) compiler syntax, yielding sequential threads (ST). Intensional function definitions, including higher order functions, are flattened using a well-know efficient technique. [1]

1.3.2 Run-time Interactive Programming Environment (RIPE)

The RIPE is a visual run-time programming environment enabling the visualization of a dataflow diagram corresponding to the Lucid parts of GIPSY programs. The user can interact with the RIPE at run-time in the following ways. [2]

- Dynamically inspect the IVW;
- Change the input/output channels of the program;
- Recompile sequential threads;
- Change the communication protocol;
- Change parts of GIPSY itself (e.g. garbage collector).

Because of the interactive nature of the RIPE, the GIPC is modularly designed to allow the individual on-the-fly compilation of either the DPR (by changing the Lucid code) ICP (by changing the communication protocol) or ST (by changing the sequential code). Such a modular design even allows sequential threads to be programs written in different languages (for now, we are concentrating on Java sequential threads).

A graphical formalism to visually represent Lucid programs as multidimensional dataflow graphs had been devised in [1]. For example, consider the Hamming problem that consists of generating the stream of all numbers of the form $2^i3^j5^k$ in increasing order and without repetition. The following Lucid program solving this problem can be translated into a dataflow diagram, as shown in Table1-1.

```
H
where
  H = 1 fby merge(merge(2*H,3*H),5*H);
  merge(x,y)= if (xx<=yy) then xx else yy
  where
    xx = x upon (xx<=yy);
    yy = y upon (yy<=xx);
  end;
end;
```

Table 1-1 Indexical Lucid program for the Hamming problem

Figure 1-3 represents the dataflow diagram defining the merge function. Such nested definitions will be implemented in the RIPE by allowing the user to expand or reduce sub-graphs, thus allowing the visualization of large scale Lucid definitions.

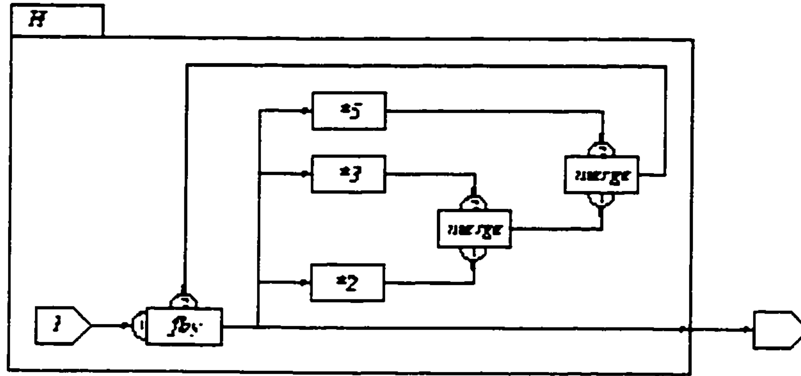


Figure 1-3 Dataflow graph for the Hamming problem

Using this visual technique, the RIPE will enable the graphic development of Lucid programs, translating the graphic version of the program into a textual version that can then be compiled into an operational version. However, the development of this facility for graphical programming poses many problems whose solutions have not yet settled. An extensive and general requirements analysis will be undertaken, as this interface will have to be suited to many different types of applications. There is also the possibility to have a kernel run-time interface on top of which we can plug-in different types of interfaces adapted to different applications. [2]

1.3.3 General Eduction Engine (GEE)

GIPSY uses a demand-driven model of computation, whose principle is that a computation takes effect only if there is an explicit demand for it. GIPSY uses eduction, which is demand-driven computation in conjunction with a value cache called a warehouse. Every demand can potentially generate a procedure call, which is either computed locally or remotely, thus eventually in parallel with other procedure calls. A value should be warehoused if it is cheaper to extract it from the warehouse than to recompute it. Every demand for an already-computed value is extracted from the warehouse rather than

computed anew. Eduction thus reduces the overhead induced by the procedure calls needed for the computation of demands. [1]

The GEE is composed of two main modules: the intensional demand propagator (IDP) and the intensional value warehouse (IVW). First, the demand propagation resources (DPR) are fed to the demand generator (IDG) by the compiler (GIPC). This data structure represents the data dependecies between all the variables in the Lucid part of the GIPSY program in input. This directs in what order all demands must be generated to compute values from this program. The demand generator recieves an initial demand, which in turn raises the need for other demands to be generated and computed. For all non-functional demands (i.e. demands not associated with the execution of a sequential thread (ST)), the IDG makes a request to the warehouse to see if this demand has already been computed. If so, the previously computed value is extracted from the warehouse. If not, the demand is propagated further, until the original demand is resolved and is put in the warehouse for further use.

Functional demands, (i.e. demands associated with the execution of a sequential thread), are sent to the demand dispatcher (IDD). The IDD takes care of sending the demand to one of the workers or resolves it locally (which normally means that a worker instance is running on the processor running the generator process). If the demands are sent to a remote worker, the communication procedures (ICP) generated by the compiler are used to communicate the demand to the worker. The IDD receives some information about the lifecycle and efficiency of all workers from the demand monitor (IDM), to help it make better decisions in dispatching functional demands.

The demand monitor, after some functional demands are sent to workers, gathers various information of each worker:

- Its status (is it still alive, not responding, or dead)
- Its network link performance
- Its response time statistics for all demands sent to it
- etc.

This information is accessed by the IDD to make better decisions about the load balancing of the workers, and thus achieving better overall run-time efficiency.

1.4 Contributions

This thesis is part of the development of the compiler (GIPC) component of the GIPSY system, as depicted in more detail in Figure 1-2 (on page 5).

More specifically, this thesis aims at:

- Semantic analysis based on GIPL (Generic Intensional Programming Language). GIPL is the common and basic set of all SIPL languages. So GIPL parser is the starting-point for semantic analysis.
    - Build a dictionary including each identifier fed to the GEE.
    - Type checking: decide the type of each identifier.
    - Rank analysis: analyze the dimensionality of each identifier.
- General function elimination in the Abstract Syntactic Tree (AST).
    - Build a function table with general symbols.
    - Generate a substitute of function call.
    - Eliminate functions in the AST.
- Develop a SIPL AST Translator Generation, which translates the AST of Indexical Lucid SIPL into the GIPL AST automatically and flexibly. Since the GIPSY back-end and run-time systems will only treat the GIPL AST, any SIPL AST constructed by a related generator has to be translated into a GIPL AST with corresponding translator. Hence, this Indexical Lucid SIPL AST translator is also acting as the template for the other new-defined SIPL AST translators.

    - Define the input mechanism for users in the translator generation.
    - Define the grammar of the translator generation.
    - Develop the translator.
    - Build a translation table for specific operators.
    - Translate specific operators in the SIPL AST.
- Explain how the components generated are to be integrated in the whole system.

1.5 Structure of the Thesis

Chapter 2 presents how to do semantic analysis from three aspects: building a dictionary, type checking and rank analysis. Chapter 3 presents function elimination. Chapter 4 presents how to generate a SIPL AST Translator Generation, as well as user input style definition and grammar definition for the translator generation. Chapter 5 presents the result of evaluation from each aspect introduced above. Chapter 6 presents how to integrate the semantic analyzer with the parser, and to integrate the semantic analysis, function elimination, and the SIPL AST translator into the GIPSY system. Chapter 7 presents the future work.

2. Semantic Analysis

GIPL is the core of language, even though there are many versions in Lucid. The semantic analyzer only analyzes the tree of the program translated to GIPL primitives, no matter in what SIPL it was written, or if it was written in GIPL itself. In this chapter, we do semantic analysis based on a GIPL AST. The SIPL AST translator generation will be discussed in chapter 4.

For the semantic analyzer, there are three basic parts: building a dictionary, type checking and rank analysis.

2.1 Building a Dictionary

2.1.1 General introduction

The parsing procedure generates an Abstract Syntactic Tree (AST). We use this tree to build a dictionary that includes the attributes of each identifier and can be used at run time.

In the Lucid program, there is no identifier declaration, the "**where**" clause implies that there is a definition. So we focus on the "**where**" node in the AST. Figure 2-1 gives an example of Lucid program and its corresponding AST.

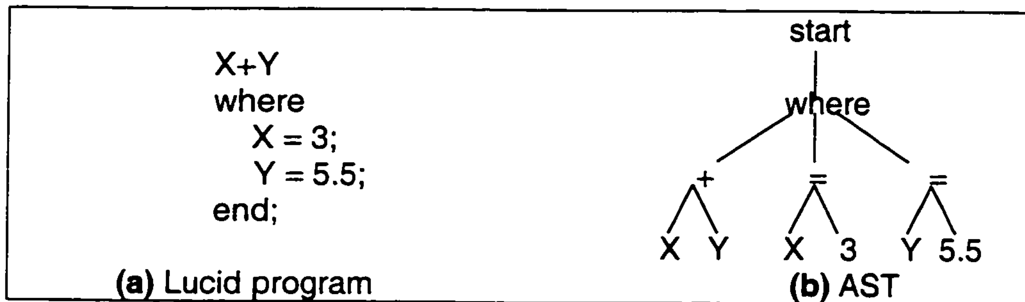


Figure 2-1 An example of Lucid program and its AST

We traverse the tree from the "**start**" node and encounter the "**where**" node. By convension, the first child of the "**where**" node is the expression before the "**where**" clause, and other children which begin with symbol "=" are the definition of identifiers enclosed directly in this "**where**" clause.

During the procedure of building a dictionary, we also need to do some simple semantic checking. In this step, the semantic checking only includes the following two kinds:

1. Check whether each identifier in the expression is defined.
In the Lucid program, each identifier used in an expression must be defined in the following "**where**" clause. There is an example in Table 2-1.

```
X + Y
    where
        X = 3;
    end
```

Table 2-1 An example of Lucid program

In the above program, there are two identifiers "**X**" and "**Y**" in the expression. However, in the following "**where**" clause, there is only one definition of identifier "**X**". In this condition, semantic analyzer will report the error message: "**Identifier Y is not defined**".

2. Check whether each identifier is in the correct scope.
Similar to regular programs, the Lucid programs also need each identifier to be used in the correct scope. This scope depends on different "**where**" clauses. There are examples in Table 2-2 and Table 2-3.

```
(1) X + Y
(2)     where
(3)         X = Y + 1;
(4)         where
(5)             Y = 3 ;
(6)         end ;
(4)     end;
```

Table 2-2 An illegal example of Lucid program

Here, the identifier "**Y**" in expression (1) is used again in definition (3), and it is defined in definition (5). This is illegal because identifier "**Y**" in the definition "**X=Y+1**" is out of scope. So in this condition, semantic analyzer will warn the error message: "**Identifier Y is overriding**".

```
(1)  X + Y
(2)  where
(3)      X = Z + 1      ┐
(4)      where          │
(5)          Z = 3;     ├        scope 1
(6)      end;           ┘
(7)      Y = Z + 5.0    ┐
(8)      where          │
(9)          Z = 3.3;   ├        scope 2
(10)     end;           ┘
(11)   end
```

Table 2-3 A legal example of Lucid program

Here, the identifier "Z" is used in definition (3) and definition (7) and is defined in definition (5) and definition (9). Because it belongs two different scopes, the program is legal.

2.1.2 Main data structure

We use the Java language for the implementation, and we adopt the Vector structure to build the dictionary. Each element in a Vector is a Class. The definition of the Class is shown in Excerpt 2-1:

```
Class Node{
        int  ID;                    // the unique code of each identifier;
        String  name;               // the name of each identifier;
        String  kind;               // is a variable or a dimension;
        int  type;                  // 0-int, 1-float, 2-string, 3-boolean;
        String  rank;               // the rank of the identifier;
        SimpleNode  entry;          // the entry address of each identifier;
        Item_in_Dic  previous; // point to the father table;
        Hashtable  ht;
}
```

Excerpt 2-1 The structure of elements in the dictionary

Because identifiers can use the same name in different scopes, we give a unique code (ID) to each identifier. The reason why we adopt the Vector structure to store the identifiers is because the position of each identifier in Vector is its ID. It will be very simple to find an identifier at run time.

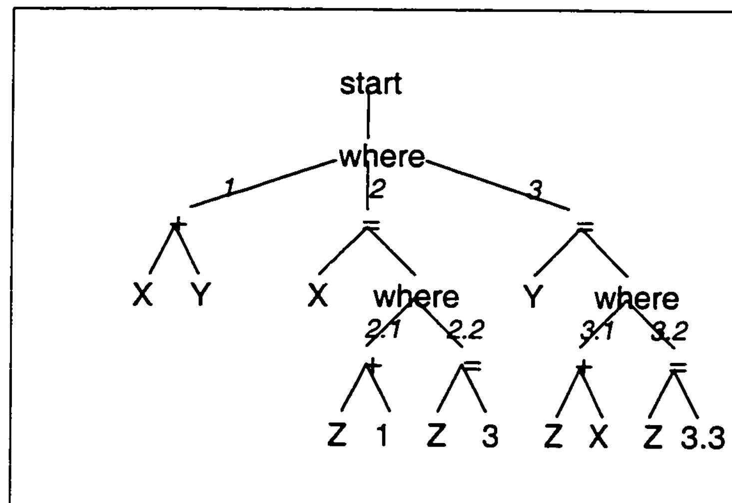
A Hashtable of each identifier can be used to distinguish the different scopes.

2.1.3 Algorithm

The main algorithm of this part is to traverse a tree by the top-down and from left to right. This is explained with an example in Figure 2-2. In the AST, we use numbers in italics to show the number of sub-trees.

```
(1) X + Y
(2) where
(3)        X = Z + 1
(4)       where
(5)            Z = 3;
(6)       end;
(7)       Y = X + Z;
(8)        where
(9)            Z = 3.3;
(10)       end;
(11) end
```

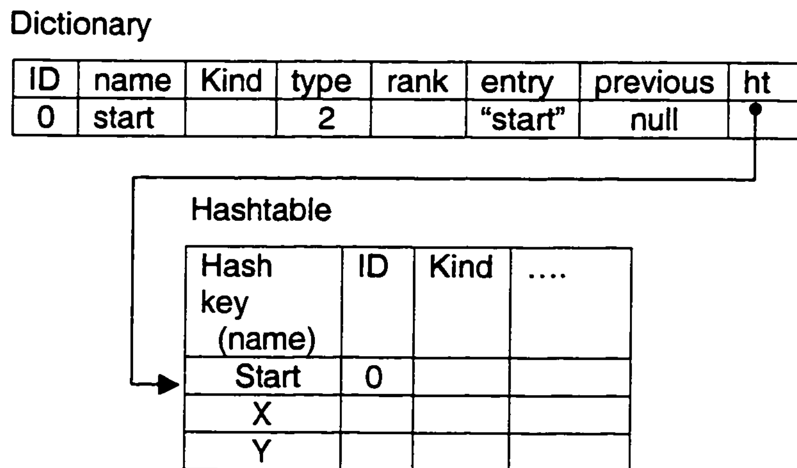
(a) A Lucid program



(b) The corresponding AST of the above program

Figure 2-2 A Lucid program and its AST

1. Begin from the "**start**" node, then move to the "**where**" node. Traverse the first child of the "**where**" node. The sub-tree1 is the expression before the "**where**" clause. We do not give the ID to each identifier now because the identifier in the sub-tree1 may not be unique. After traversing the sub-tree1, there is no change in the AST. The Hashtable and Dictionary are in Figure 2-3.



Dictionary

ID	name	Kind	type	rank	entry	previous	ht
0	start		2		"start"	null	

Hashtable

Hash key (name)	ID	Kind	....
Start	0		
X			
Y			

Figure 2-3 The Hashtable and Dictionary after traversing the sub-tree1

2. Then, traverse the second child of the "**where**" node. The sub-tree2 is the definition of identifier "**X**", and it includes another definition of identifier "**Z**". We will give **ID=1** to identifier "**X**" and rewrite its ID in the sub-tree1 and the other sub-trees of the same "**where**" clause. After, continue to traverse sub-tree2.1, sub-tree2.2. The operation of sub-tree2.1 and sub-tree2.2 is the same as sub-tree1 and sub-tree2. After traversing sub-tree2, there is a change in the original AST. The new AST is shown in Figure 2-4 and the Hashtable and Dictionary are shown in Figure 2-5.

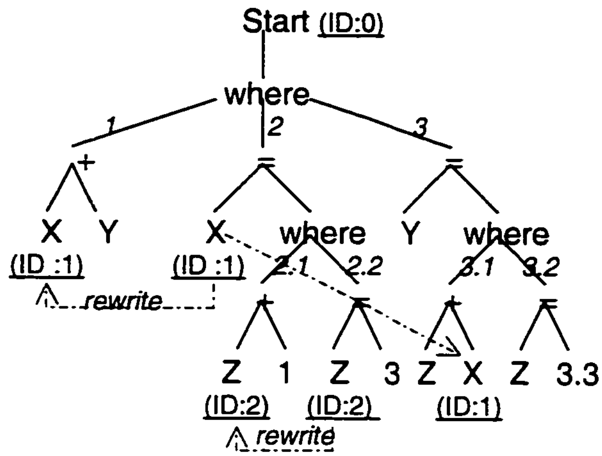


Figure 2-4 The new AST after traversing the sub-tree2

Dictionary

ID	name	Kind	type	Rank	entry	previous	Ht
0	Start		2		"start"	null	•
1	X	Identifier			"+"	null	•
2	Z	Identifier			"3"	X	Null

Hashtable

Hash key (name)	ID	kind	....
→ start	0		
X	1		
Y			

Hash key (name)	ID	kind	....
→ X	1		
Z	2		

Figure 2-5 The Hashtable and Dictionary after traversing the sub-tree2

3. Last, traverse the third child of the "**where**" node. The sub-tree3 is the definition of identifier "**Y**" and it also includes another definition of identifier "**Z**". The operation on the sub-tree3 is the same as the sub-tree2. The last result of the AST is shown in Figure 2-6 and the Hashtable and Dictionary are shown in Figure 2-7.

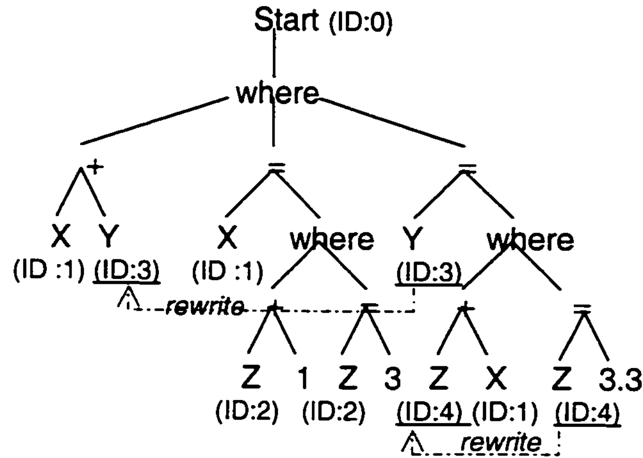


Figure 2-6 The result of the AST after traversing the sub-tree3

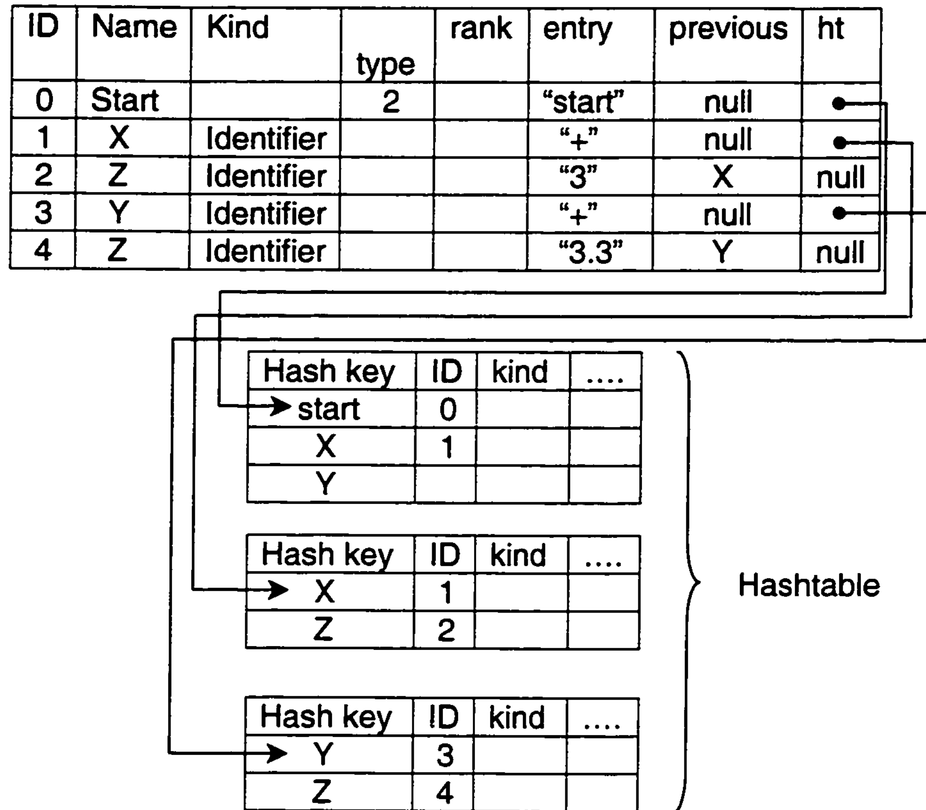


Figure 2-7 The Hashtable and Dictionary after traversing the sub-tree3

2.1.4 Error reporting and error recovery

As we mentioned in chapter 2.1.1, during the process of building a dictionary, we only consider two categories of errors.

1. If there is an identifier that is not defined in an expression, the system will report the error message: "**Identifier is not defined**" and make the error number increased by one. The system will not write the information of the identifier into the dictionary.

2. If a definition for this identifier has already been encountered in the same scope, the system will report the warning error message: "**Identifier is re-defined.**" and make the warning error number increased by one. The system will re-write the new information of the identifier into the dictionary, but the ID of the identifier in the AST will be covered with the new ID.

3. If there is an identifier to be defined again under the same scope, the system will report the warning error message: "**Identifier might be overriding.**" and make the warning error number increased by one. The system will give a new ID to the identifier and write the information of the identifier into the dictionary; however, the ID of higher original level identifier will not be changed.

2.2 Type Checking

2.2.1 General Introduction

As we noticed in the last step, we do not consider the type of each identifier. The type attribute in the dictionary is blank. In this step, the main task is to decide the type of each identifier, rewrite it into the dictionary and modify each related node in the AST.

In the Lucid program, there is no identifier declaration. We can only deduce the type of identifiers by each definition. We still use the example in Figure 2-2 (on page 16) to illustrate the method. First, we encounter the expression "**X+Y**" and do not know the type of these variables. Then, we encounter the definitions "**X=3**" and "**Y=5.5**"; the former means the type of identifier "**X**" is integer and the latter means the type of identifier "**Y**" is float. Finally, we can deduce that the type of expression "**X+Y**" is float.

2.2.2 Type checking rules

During semantic analysis, the main focus is on the type checking. By convension, if we encounter two operands with different types, we permit implicit type casting from integer to float. We define the rules in Table 2-4 as the basic rules in the type checking procedure [7].

Rule1: x OP y If the type of **x** and **y** are different, implicit promotion is from **integer** to **float**. **x** or **y** are of type dimension will be reported as semantic errors. Semantic analysis is after function elimination, so type function is not considered here. **Rule2**: IF cond THEN x ELSE y **cond** must be a result of logic operators. **x** and **y** must be of the same type. **Rule3**: x @ (#.d) N **d** must be of type dimension, and that dimension must already have been defined. The type of **N** must be integer. **Rule4**: #.d **d** must be of type dimension, and that dimension must already have been defined. The type of **#.d** is always integer.

Table 2-4 The rules used in the type checking

2.2.3 Main data structure

We adopt a bottom-up algorithm to do the type checking. There are two stacks used to help do the type checking. While we build the dictionary, we push all terminal leaves into a semantic stack. When we do type checking, we use another temp stack to store the temporary results of a long expression.

For example, if a Lucid program is the same as in Figure2-2 (on page 16), after building the dictionary, the semantic stack is in Table 2-5.

5.5
Y
3
X
Y
X

Table 2-5 The semantic stack

There is no long expression, so we do not use the stack. If there is an expression as **X=A+B+C**, the AST of the expression is shown in Figure 2-8.

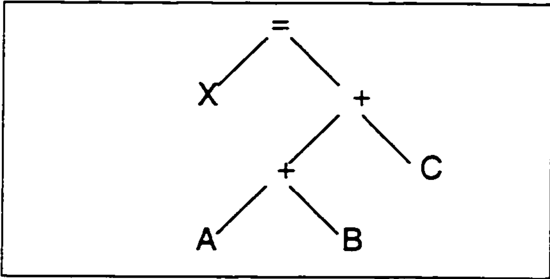


Figure 2-8 The AST of the long expression

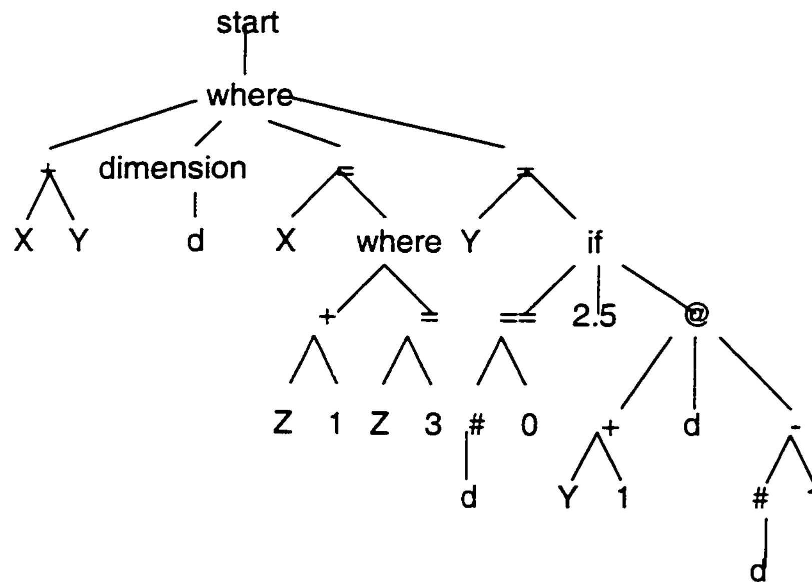
Because we use a bottom-up algorithm and the direction is from right to left, we first have the result of identifier "**C**" and we should push the result of "**C**" into the stack. After finishing to calculate the result of **A+B**, we pop the result of **C** and continue to do the type checking.

2.2.4 Algorithm

The main algorithm of this step is to traverse a tree in bottom-up from right to left. We will explain it with an example in Figure 2-9.

```
(1) X + Y
(2) where
(3)         dimension d;
(4)         X = Z + 1
(5)         where
(6)             Z = 3;
(7)         end;
(8)         Y = if (#.d)= =0 then 2.5 else (Y+1) @ .d (#.d)-1;
(9) end;
```

(a) A Lucid program



(b) The corresponding AST of the above program

Figure 2-9 The example used to do the type checking

After building the dictionary, the semantic stack is shown in Table 2-6.

1
d
d
1
Y
2.5
0
D
Y
3
Z
1
Z
X
Y
X

Table 2-6 The semantic stack after building the dictionary

1. We begin the type checking from popping the first element "**1**" in the semantic stack. The element "**1**" is the rightmost leaf in the AST. Because the left child of its parent node "**-**" is not a terminal, we push the element "**1**" into the temp stack.

2. Next, we pop the second element "**d**" from the semantic stack and face the operation about **#**. First, we must check whether "**d**" is a dimension or not, and whether it has already been defined. Then, according to Rule4, the value of **(#. d)** is integer, so we can deduce the type of **(#. d)-1** is integer. Finally, we must rewrite the type of corresponding nodes "**#**" and "**-**" in the AST.

3. Continue to pop the third element "**d**" from the semantic stack, we face the operation about **@**. We must check whether "**d**" is a dimension or not, and whether it has already been defined. Then, pop the next two elements "**1**" and "**Y**" from the semantic stack, we consider the expression "**Y+1**". Because the const "**1**" is an integer, we can deduce

the type of node "+" is integer. However, The expression is not a definition, so we cannot deduce the type of identifier "**Y**" is integer. According to the type of three children of node "**@**", we can deduce the type of node "**@**" is integer. We do not rewrite the type of "**Y**" into the dictionary at this time instead of modifying the node "**+**" and "**@**" of the AST.

4. Continue to pop the sixth element "**2.5**" from the semantic stack, we encounter the "**if**" expression. Comparing the node "**2.5**" and the node "**@**", the types are not same. So the program will report the error, and modify the type of node "**if**" as float. After poping the next two elements "**0**" and "**d**", and dealing with **(#. d)= = 0**, we should judge whether the condition is a Boolean type or not.

5. Finally, we pop the identifier "**Y**" from the semantic stack and face its definition. At this time, we can deduce the type of identifier "**Y**" is float according to the type of node "**if**". After traversing the sub-tree of "**Y=...**", the new AST is shown in Figure2-10 and the dictionary is shown in Table 2-7.

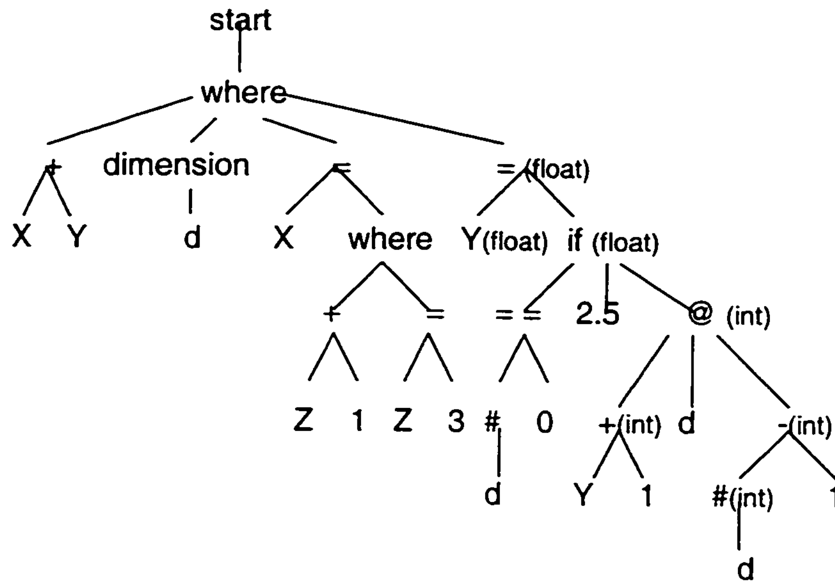


Figure 2-10 The new AST after type checking on the sub-tree of Y

Dictionary

ID	Name	Kind	type	rank	entry	previous	ht
0	Start		2		"start"	null	*
1	D	dimension	2		null	null	null
2	X	Identifier			"+"	null	*
3	Z	Identifier			"3"	X	null
4	Y	Identifier	1		"if"	null	null

Table 2-7 The Dictionary after type checking on the sub-tree of Y

6. The operation on the sub-tree of **X** is the same as **Y**. After the whole type checking, we will encounter the "**start**" node. This signals the end of this step. The final result of the AST and the dictionary is shown in Figure 2-11 and Table 2-8.

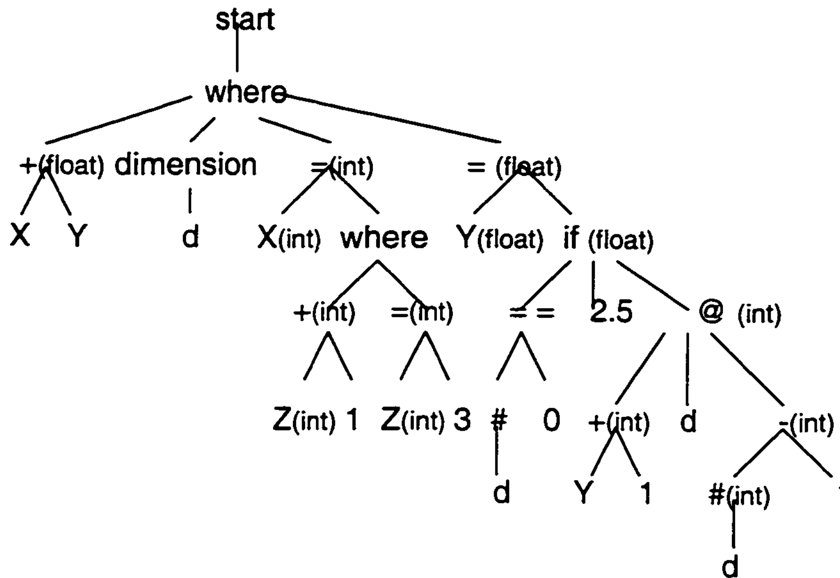


Figure 2-11 The final AST after type checking

Dictionary

ID	Name	Kind	type	rank	Entry	previous	ht
0	Start		2		"start"	null	*
1	d	dimension	2		Null	null	null
2	X	Identifier	0		"+"	null	*
3	Z	Identifier	0		"3"	X	null
4	Y	Identifier	1		"if"	null	null

Table 2-8 The dictionary after type checking

2.2.5 Error reporting and error recovery

In this part, we will do type checking based on the rules in Table 2-4 (on page 22). Because there are only four possibilities that should be considered in the GIPL, we will discuss errors under these four conditions. Generally, we will report errors and make the error number increased by one. The specific information can be seen in Table 2-9.

Error reporting	Explanation	Operation
Variable is not dimension.	In #.**d** or @.**d**, the variable **d** is not dimension.	Do not generate the attribute of the variable.
Dimension is not defined.	In #.**d** or @.**d**, the dimension d is not defined.	Do not generate the attribute of the variable.
MOD operands must be integer.	**X** mod **Y**, the type of **X** and **Y** must be integer.	Do not do Mod operation.
Left and right operands' type does not match.	**X>Y**, the type of **X** and **Y** is not the same.	Implicit type casting from integer to float.
The condition is not boolean type.	If **condition** then **X** else **Y**. The **condition** must be a result of logic operator.	Do not do judgement.
If Statement's type does not match.	If **condition** then **X** else **Y**. The type of **X** and **Y** must be same.	Implicit type casting from integer to float.

Table 2-9 Error information in the type checking

2.3 Rank Analysis

Rank analysis means that we should analyze the effective dimension of expressions and each identifier under multi-dimensions.

2.3.1 General introduction

The Lucid language permits the operation on multi-dimensions. This is the character of Lucid program and also is a very abstract concept. We can illustrate the meaning of rank by Figure 2-12.

There is a matrix **A**, the effective dimension is **X** and **Y**, then the rank of **A** is **{X, Y}**.

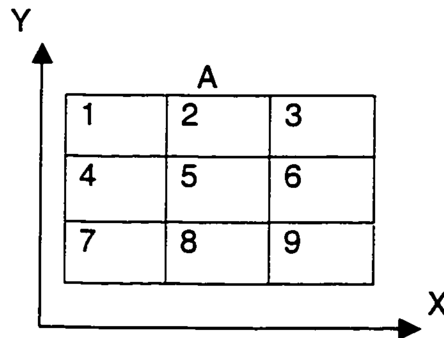


Figure 2-12 Rank[A] = {X, Y}

If there is an expression: **A+3**. Rank of the constant "**3**" is ∅. This means in dimension **X** and **Y**, each element of **A** should plus **3**. So, rank [A+3] = {X, Y} U ∅ = {X, Y}. Then, the result is in Figure 2-13:

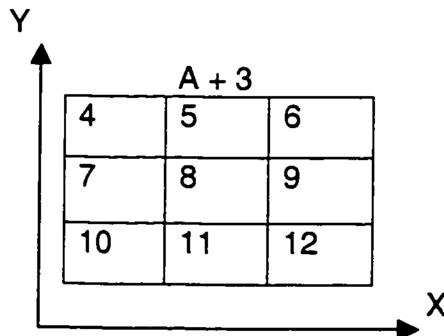


Figure 2-13 Rank [A+3] = {X, Y} U ∅ = {X, Y}

The rank definition is different from variable definition. In Lucid programs, the definition of dimension is declared in the "**where**" clauses. For example, there is a simple Lucid program in Table 2-10. If there is no declaration "**dimension**

d" before the definition of **X**, compiler will report the error message: **"dimension d is not defined."**

```
X
where
    dimension d;
    X = if (#.d) = = 0 then 0 else (X+1)@#.d (#.d-1);
End
```

Table 2-10 A Lucid program with dimension declaration

2.3.2 Rank analysis rules

After we know the meaning of the rank and basic rules, based on [1] and [6], we can give the rules of how to analyze the rank of variables in Table 2-11.

Rule1:	rank [constant] = ∅;
	This means that constants are constant in all dimensions;
Rule2:	rank [identifier] = rank [expression defining the identifier];
Rule3:	rank [E1 op E2] = rank [E1] U rank [E2]
	*op can be Arithmetic operators: +, -, *, /, mod ;*
	Boolean operators: = =, >=, <=, >, <, !=.
Rule4:	rank[if E1 then E2 else E3 fi] = rank[E1] U rank[E2] U rank[E3]
Rule5:	rank[E1 @ .d E2] = rank[E1] – {d}
Rule6:	rank[#.d] = {d};
Rule7:	if an expression contains "#.d", its rank includes dimension "d".

Table 2-11 The rules used in the rank analysis

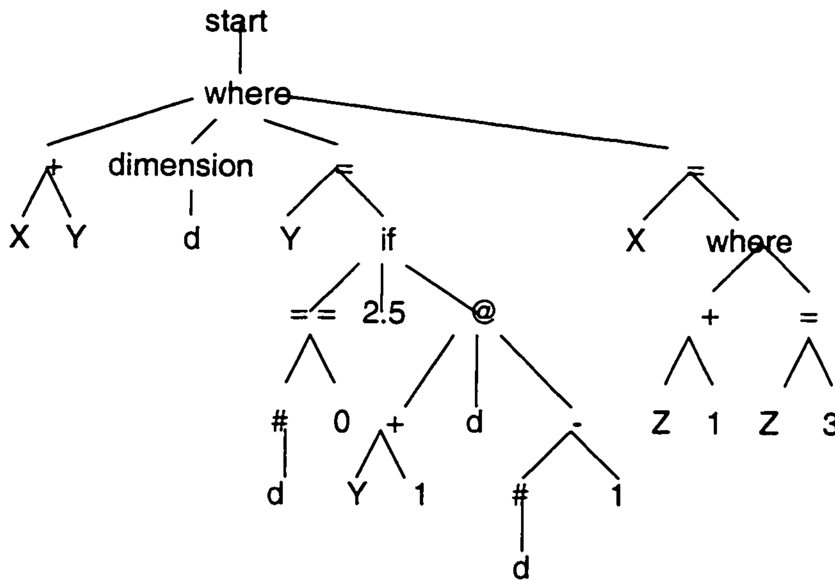
2.3.3 Algorithm

Because we do rank analysis accompanying the type checking, there is the same data structure as in the type checking. The main algorithm of this step

is to traverse a tree bottom-up from right to left. We will explain it with an example in Figure 2-14.

```
(1) X + Y
(2) where
(3)       dimension d;
(4)       Y = if (#.d)= =0 then 2.5 else (Y+1) @ .d (#.d)-1;
(5)       X = Z + 1;
(6)       where
(7)             Z = 3;
(8)       end;
(9) end
```

(a) A Lucid program



(b) The corresponding AST of the above program

Figure 2-14 The example used to do the rank analysis

1. First, we encounter the definition "**Z=3**". According to **Rule1** and **Rule2**, we can deduce **rank [Z]=** ∅. Then, we should rewrite the rank of

identifier "**Z**" into the dictionary and modify the related nodes in the AST. We show the change of the AST in Figure2-15 and the change of the dictionary in Table 2-12.

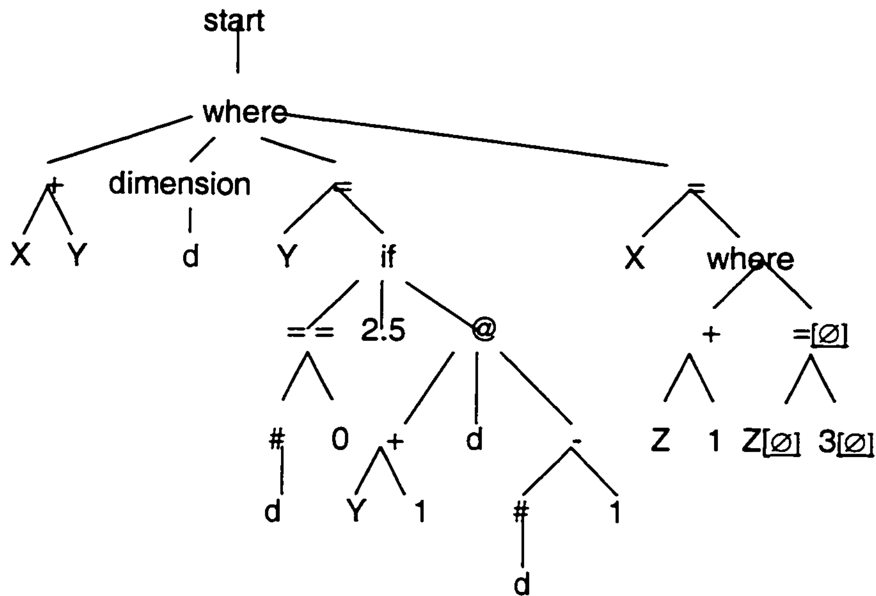


Figure 2-15 The corresponding AST after analyzing "Z=3"

Dictionary

ID	Name	Kind	type	rank	entry	previous	ht
0	Start		2	''	"start"	null	*
1	d	dimension	2	''	null	null	null
2	Y	Identifier	1		"if"	null	null
3	X	Identifier	0		"+"	null	*
4	Z	Identifier	0	∅	"3"	X	null

Table 2-12 The dictionary after analyzing "Z=3"

2. Next, we encounter the expression "**Z+1**". According to **Rule3**, we can deduce **rank [Z+1]=** ∅. Then, we encounter the definition "**X=Z+1**". According to **Rule2**, rank [X]=rank [Z+1]= ∅. At this time, we should

rewrite the rank of identifier **"X"** into the dictionary and modify the related nodes in the AST. We show the change of the AST in Figure2-16 and the change of the dictionary in Table2-13.

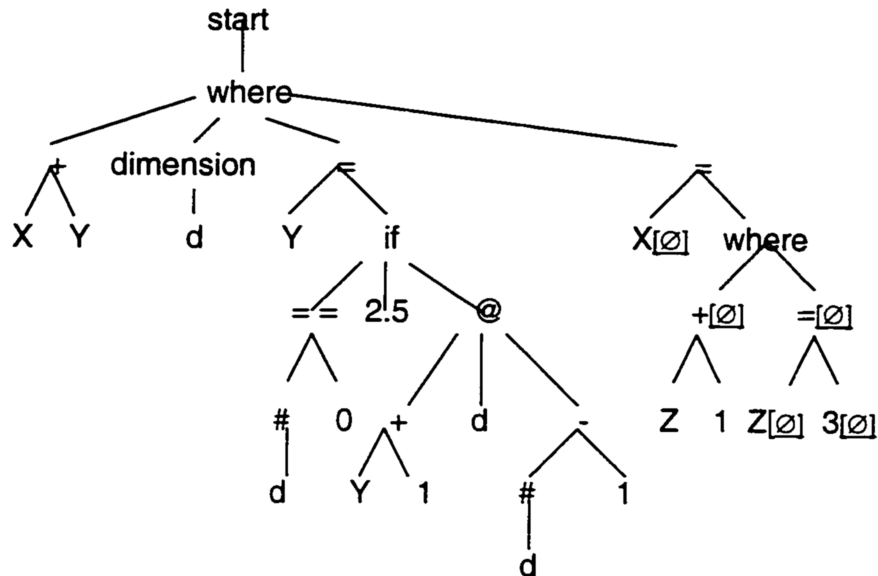


Figure 2-16 The corresponding AST after analyzing "X=Z+1"

Dictionary

ID	Name	Kind	Type	rank	entry	previous	Ht
0	Start		2	"	"start"	null	*
1	d	dimension	2	"	null	null	null
2	Y	Identifier	1		"if"	null	null
3	X	Identifier	0	∅	"+"	null	*
4	Z	Identifier	0	∅	"3"	X	Null

Table 2-13 The dictionary after analyzing "X=Z+1"

3. Next, we encounter the "@" expression. The first child of the node "@" is the recursive **Y**, but we do not consider this condition first; according

to **Rule7**, we can deduce the rank of the @ expression should include "**d**". Then, according to **Rule5**, we can deduce that the final rank of the @ expression is "∅". After modifying the related nodes, the new AST is shown in Figure2-17.

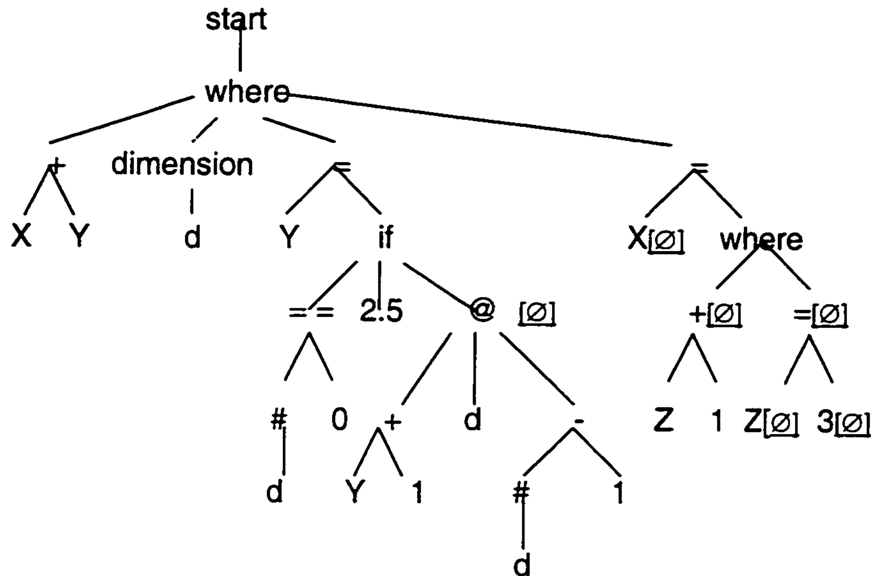


Figure 2-17 The corresponding AST after analyzing the @ expression

4. Next, we encounter the "**if**" expression. The first child of the node "**if**" a condition **(#.d)==0**; according to **Rule7**, we deduce that the rank of the "**if**" expression includes "**d**". Then, we use **Rule4** to deduce that the final rank of the "**if**" expression equals **{d} U ∅ U ∅ = {d}.**

5. Finally, we encounter the definition of "**Y**". According to **Rule2**, we can deduce that the rank of "**Y**" is **{d}**. The final AST and dictionary is shown in Figure 2-18 and Table 2-14.

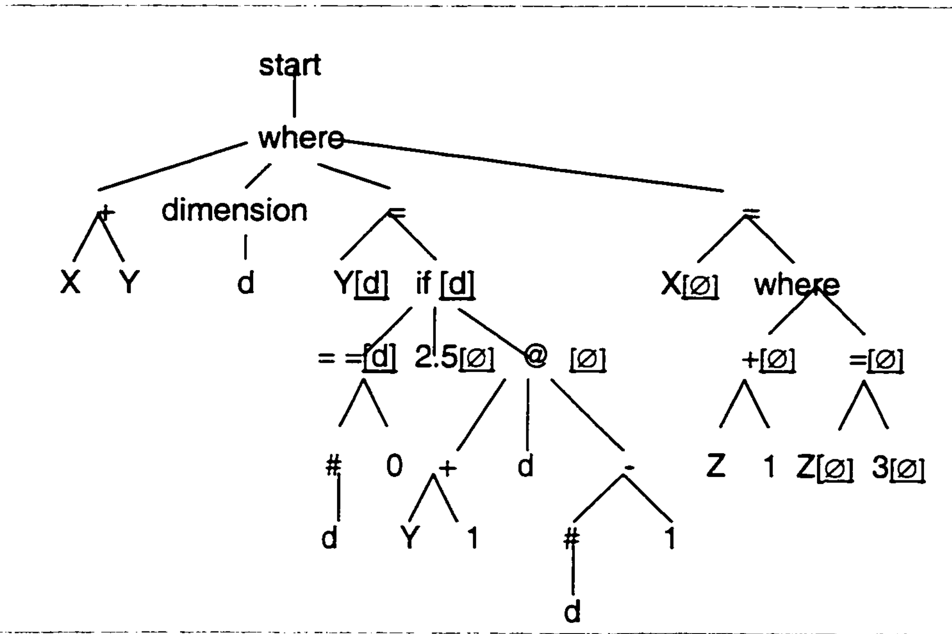


Figure 2-18 The final AST

Dictionary

ID	Name	Kind	Type	rank	Entry	previous	ht
0	Start		2	"	"start"	null	*
1	D	dimension	2	"	Null	null	null
2	Y	Identifier	1	D	"if"	null	null
3	X	Identifier	0	∅	"+"	null	*
4	Z	Identifier	0	∅	"3"	X	Null

Table 2-14 The final dictionary

2.3.4 Error reporting and error recovery

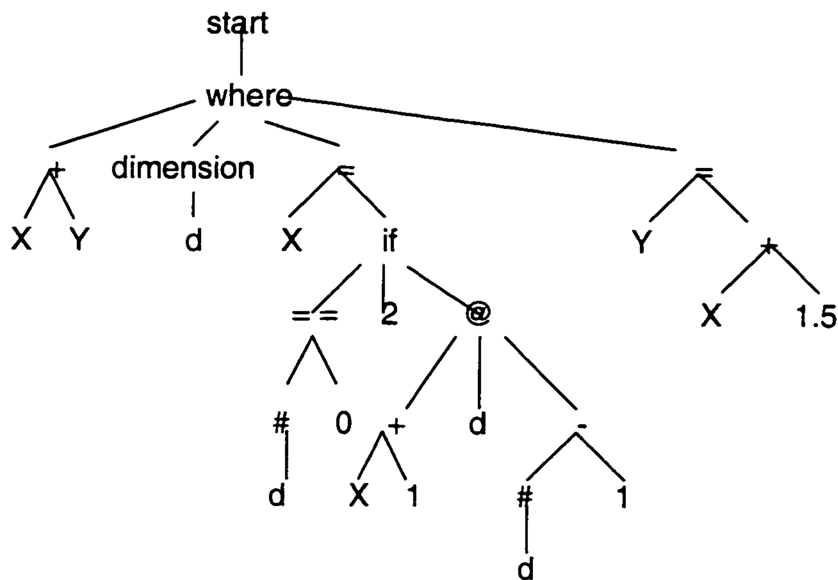
Because rank analysis accompanies type checking, error operation here only concerns the dimension. If a variable after the "#" and the "@" is not a dimension or is a dimension which is not defined, the system will report the error and make the error number increased by one. If any error happens on the dimension, the system will return a blank rank to the present operators.

2.4 Semantic Analysis: Second Pass

When we do the first semantic check, some identifiers' type and rank are not available. So, we adopt the second semantic check. Figure 2-19 is a typical example.

```
x+y
where
    dimension d;
    x = if (#.d)==0 then 2 else (x+1)@.d (#.d)-1 fi;
    y=x+1.5;
end
```

(a) A Lucid program



(b) The corresponding AST of the above program

Figure 2-19 A typical Lucid program that need the second semantic analysis

After building the dictionary, we give an ID to each identifier. Then, we adopt a bottom-up algorithm to do the type checking and rank analysis. When we do the expression "**X+1.5**", the **type (0)** and the **rank {d}** of **X** are not figured out

and are not available in the dictionary at this time. So, we must skip the point and do the second semantic check after the finish of first type checking and rank analysis of the whole AST.

2.5 Summary

By the end of the three steps, we figure out the type and rank of each identifier. We also modify the corresponding nodes in the AST. During running time, RIPE and GEE can get useful information both in the AST and in the dictionary, which will improve both convenience and efficiency.

However, we do not consider the condition that if there are function nodes in the AST. So in next chapter, we will introduce function elimination.

3. Function Elimination

3.1 Introduction

The reason why we consider eliminating functions in the AST is because at the first step, the project will only deal with the simplest condition, so the AST used by the GEE will be the basic GIPL AST which only has **#**, **@**, **if** and **where**. We can regard function as a Macro and expand a function call with its definition. Table 3-1 illustrates the meaning of function elimination.

```
A
where
      dimension d;
      A = 1 fby .d  merge .d (2*A, 3*A);
      merge .a (x,y) = if (xx<=yy) then xx else yy fi
       where
             xx = x upon .a (xx<=yy);
             yy = y upon .a (yy<=xx);
       end;
end
```

(a) A Lucid program with a function

```
A
where
      dimension d;
      A = 1 fby .d  if (xx<=yy) then xx else yy fi
      where
             xx = 2*A  upon .d  (xx<=yy);
             yy = 3*A  upon .d  (yy<=xx);
       end;
end
```

(b) The Lucid program after eliminating the function

Table 3-1 The meaning of function elimination

From the example, we can conclude some points about function elimination.

1. We must use the definition of a function to replace the function;
2. We must use actual parameters of a function call, for example 2*A, to replace the original;
3. It is possible to change the structure of the AST after function elimination.

After making clear the meaning, we will consider how to eliminate functions in the AST. First, we must eliminate the function and let the program have the same meaning. Second, we will correctly change the AST and make it accordant to the convensions. In the following, we introduce the process.

3.2 Function Table Construction

First of all, we need a function table in which records all functions of a Lucid program.

Similiar to the definition of identifiers, the function definition is also under the "**where**" clauses. During the process that we build the dictionary, we also can get information about functions. However, there are differences between building the identifier table and function table:

- A function has parameters;
- After storing the definition of a function into the table, the branch of the function will no longer exist;
- Function name is unique and there is no ID to each function.

According to these differences, we will use different algorithms to build the function table. The algorithm can be realized in this method: ***void SetFunc(SimpleNode FunT, int Sfn).*** The next section is the first step.

3.2.1 Use general symbols to replace parameters in the original function

As we discussed above, one of the differences with a variable declarations is that a function has parameters. In later function calls, it can use different actual parameters. So, when we build the function sets, we must use general symbols to replace the parameters first. Otherwise, we cannot make out what are parameters and what are identifiers in later function calls. Even though we can make another table to label each parameter, this algorithm will waste both time and space. So, by careful analysis, we prefer the present algorithm and this process can be finished by the first time we traverse the AST. There is an example in Table 3-2.

```
twoat.d (A,B)+1
where
    dimension d;
    twoat.a (x,y)=xx+yy;
    where
        xx=x @ .a 1;
        yy=y @ .a 1;
    end;
end
```

Table 3-2 A Lucid program with a function

After parsing, the branch of the function definition is shown in Table 3-3. The bold parts are the content of the function. The italic parts are information about the function parameters.

```
Equal
    id: twoat
        dimension: a
        parameters
          id: x
          id: y
    where
        add
            id: xx
            id: yy
        equal
            id: xx
            at
                id: x
                dimension: a
                const: 1
        equal
            id: yy
            at
                id: y
                dimension: a
                const: 1
```

Table 3-3 The branch AST of the function definition

In the above table, the dimension "**a**" and parameters "**x**" and "**y**" are all very difficult to distinguish from other normal variables "**xx**" and "**yy**". So, first we use array **Para[ ]** to store parameters "**x**" and "**y**", and array **Dim[ ]** to store dimension "**a**". Then, we introduce the method **Take(SimpleNode FunB)** to

change the parameters in function **twoat.a(x,y)** with some special symbols. Here, we use character '~' with a number to replace normal parameters, and we use character '#' with a number to replace dimensions. Excerpt 3-1 explains the process.

```
SimpleNode Take(SimpleNode FunB){

    if (FunB.children==null){
      if (FunB.id==25) //is a identifier
      {
        for (int fp=0; fp<PN; fp++)
          if (FunB.image.equals(Para(fp))) FunB.image="~"+Integer.toString(fp);
      }
      else if (FunB.id==23)
      {
        for (int fd=0; fd<DN; fd++)
          if (FunB.image.equals(Dim(fd))) FunB.image="#"+Integer.toString(fd);
      }
    }
    else{
      for (int f=0;f<FunB.children.length;f++)
        Take((SimpleNode)FunB.children(f));
      }

    return FunB;
}
```

Excerpt 3-1 The method to realize the function parameters generalized

Then, referring to the above example, we can get the generalized function definition as shown in Table 3-4. We use **~1** to replace **x**, **~2** to replace **y**, and **#1** to replace dimension **a**.

```
Equal
    id: twoat
        dimension: a
        parameters
          id: x
          id: y
```

```
where
  add
    id: xx
    id: yy
  equal
    id: xx
    at
      id: x ~1
      dimension: a #1
      const: 1
  equal
    id: yy
    at
      id: y ~2
      dimension: a #1
      const: 1
```

Table 3-4 The generalized function definition branch

3.2.2 Building the function table

Now that we have the generalized function definition branch, the next step is to build a function table to store each function. We use a hashtable to organize functions, and the structure of each item is defined in Excerpt 3-2.

```
public class Fun_Item {
  String FunName;
  int DimNum;
  int ParaNum;
  SimpleNode FunEntry;
}
```

Excerpt 3-2 The structure of items in the function table

In Table 3-4, the attribute of function **twoat.a (x, y)** can be filled with **(twoat, 1, 2, 'where' node)** and can be stored into the hashtable.

As we discussed in Section 3.2, after the functions are stored in the table, the original AST should be fixed by eliminating the branch of the function definition. Excerpt 3-3 illustrates how to make the branch of function definition empty.

```
ft.id=27; //is a function
ft.image="";
ft.children=null;
ft.parent=FunT;
FunT.children[m]=ft;
```

Excerpt 3-3 Eliminating the branch of function definition

3.3 Function Elimination

All prepared work in Section 3.2 will be used in this section. When we face the AST fixed in Table 3-5, we will use method: **void FunEliminate (SimpleNode begin, int Fcn)** to eliminate functions.

```
Where
   add
      id: twoat
         dimension: d
         parameters
            Id: A
            Id: B
      const: 1
   dimension: d
   function:   // be eliminated
```

Table 3-5 The AST after eliminating function

3.3.1 Using actual parameters to replace general symbols

Each function call will have its own parameters. For example, in Table 3-5, the use of function **twoat( )** with dimension **d** and parameters **A, B** can be seen. Of course, if we want to use the definition of function to replace the node here, we must use the actual parameters.

However, this introduces another problem. There is generalized function definition in the library. If we directly use the actual parameters to replace the function, the original data will be changed, which is harmful to the function if called again. So, we should duplicate the small AST first. The method: **void Duplicate (SimpleNode root, SimpleNode Ori_tree, int Child_Num)** will help duplicate a small tree. All changes will be done on this duplicated AST.

Then, according to the conventions, method: **void Replace (SimpleNode ReTree, String dim[], String para[])** will use actual parameters **d, A, B** to replace the general symbol with characters ~ and #. Now, we can get the function **twoat**'s substitute as shown in Table 3-6.

```
Where
    add
        id: xx
        id: yy
    equal
        id: xx
        at
            id: ~1 A
            dimension: #1 d
            const: 1
    equal
```

id: yy

at

id: ~~-2~~ B

dimension: ~~#1~~ d

const: 1

Table 3-6 The substitute of function twoat( )

Now that we have the substitute of a function, the remaining work involves using the substitute to replace the function in the Lucid program. This process will be discussed in the next two sections.

3.3.2 General function elimination

As discussed in Section 3.2, during the process of eliminating functions, the original AST may be changed. The difficult part is that all changes should keep the same conventions and ensure the AST keep the same meanning.

In this section, we only discuss the condition that involves no new variable being introduced in function definition. This means the beginning of the sub tree of the function is not "**where**" node. Table 3-7 gives a simple example of this condition.

```
F.d (A,B) + 3;
 where
      dimension t;
      F.t (x,y) = x @ .t y;
 end
```

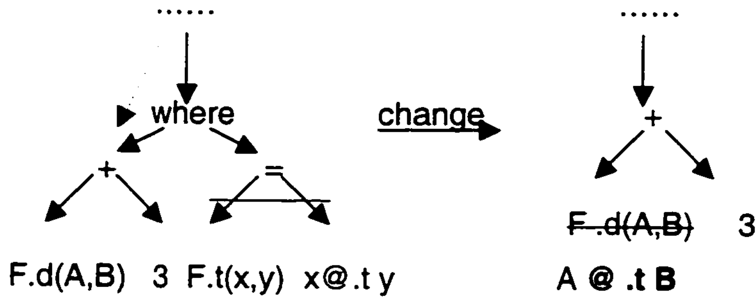
Table 3-7 A simple Lucid program with function

In this condition, the rules used to eliminate function are given following:

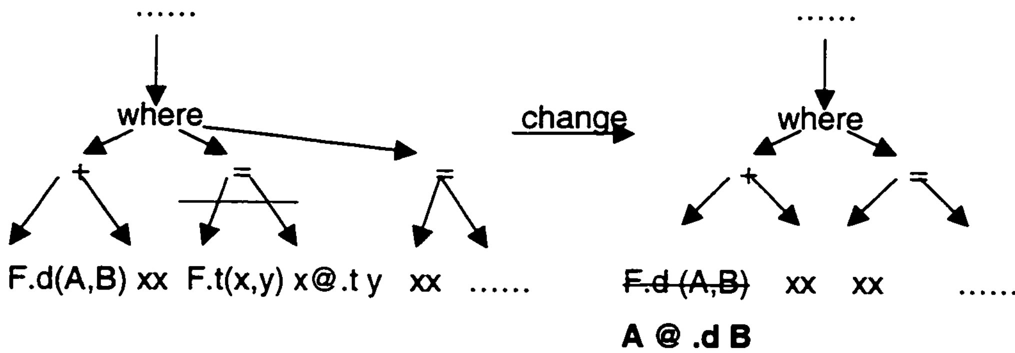
Rule 1: If function definition is only the right child under the "**where**" node, we must change the whole structure of AST: delete the "**where**" level, put up the left child of the original "**where**" node, and put it on the original "**where**" node.

Rule 2: If there is more than one child defined under the "**where**" node, we only set the function branch as null.

Figre 3-1 illustrates how to execute the two rules.



(a) The change process illustration of rule 1



(b) The change process illustration of rule 2

Figure 3-1 The change process illustration

We now clearly know how to eliminate functions under general condition. The method: **void FunEliminate(SimpleNode begin, int Fcn)** can also realize it successfully. Table 3-8 shows the result after eliminating the function based on the example in Table 3-7.

```
F.d (A,B) + 3;
 where
      dimension t;
      F.t (x,y) = x @ .t y;
 end
```

(a) The original Lucid program

```
A @ .d B + 3;
```

(b) The program after eliminating function

```
Add
   at
      id : A
      dimension: d
      id: B
   const: 3
```

(c) The AST after eliminating function

Table 3-8 Change process during general function elimination

3.3.3 Function elimination with new variable definitions

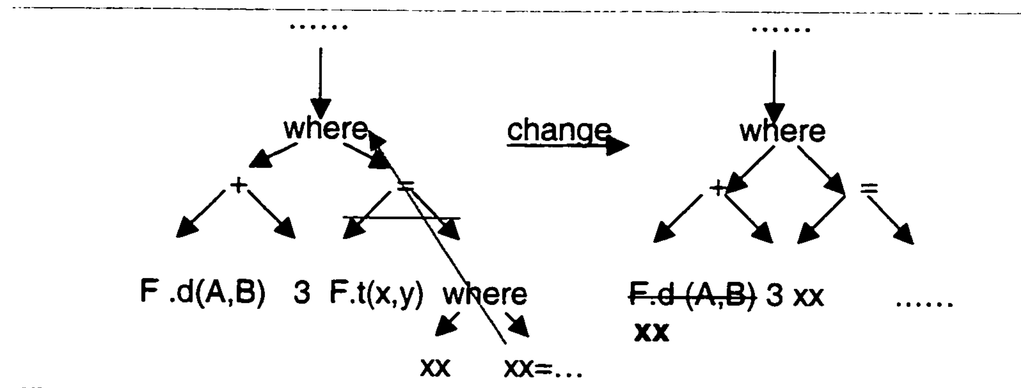
After discussing the general condition, in this section, we discuss the condition in which there are new variables introduced in function definition. This means the sub tree of the function begins with "**where**" node. The example in Table 3-2 is under this condition.

Here, no matter how many children are involved, only two conditions should be considered. We conclude them with rule 3 and rule 4.

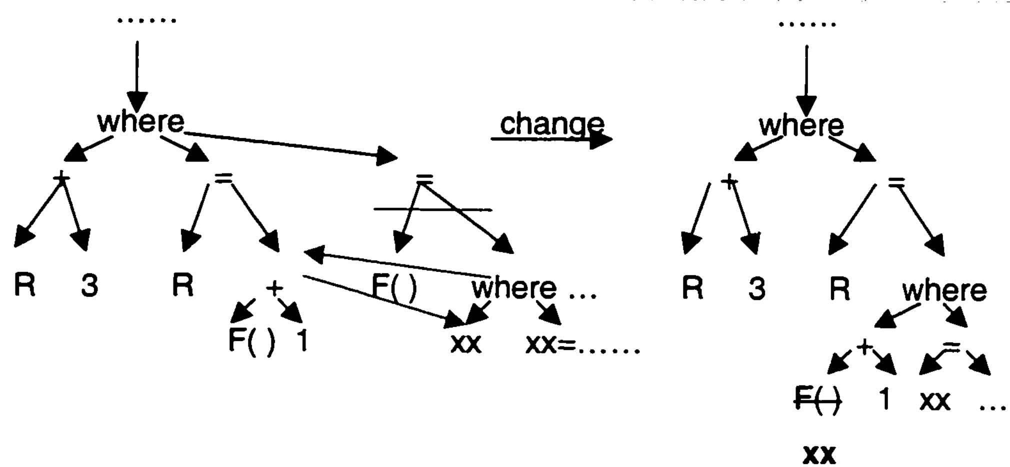
Rule 3: If a function's use and definition are at the same level, we will put up the variables' definition in the function and put them under the up-where level directly. Delete the function definition branch.

Rule 4: If a function's use and definition are at different levels, we will put all variables' definition in the function to be under the function's use branch. Delete the function definition branch.

Figre 3-2 illustrates how to execute the two rules.



(a) The change process illustration of rule 3



(b) The change process illustration of rule 4

Figure 3-2 The change process illustration

Function elimination in this condition is more complex than in the general condition; however, method: **void FunEliminate (SimpleNode begin, int Fcn)** can realize it successfully. Table 3-9 shows the result after eliminating functions based on the example in Table 3-2.

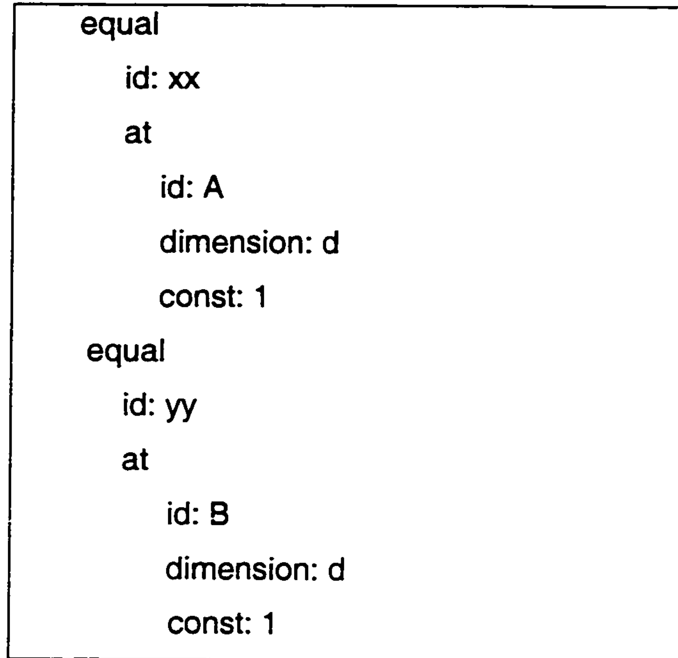
```
twoat.d (A,B)+1
where
    dimension d;
    twoat.a (x,y)=xx+yy;
    where
        xx=x @ .a 1;
        yy=y @ .a 1;
    end;
end
```

(a) The original Lucid program

```
xx + yy + 1
where
      dimension d;
      xx = A @ .d 1;
      yy = B @ .d 1;
end;
```

(b) The program after eliminating function

```
Where
   add
      add
         id: xx
         id: yy
      const: 1
   dimension: d
```



```
equal
   id: xx
   at
      id: A
      dimension: d
      const: 1
equal
   id: yy
   at
      id: B
      dimension: d
      const: 1
```

(c) The AST after eliminating function

Table 3-9 Change process during specific function elimination

3.4 Error Reporting and Error Recovery

It is always important to deal with errors. In this section, we will pay more attention to the errors on functions. These errors include whether functions are defined and whether the numbers of parameters are correct. Because we first eliminate function and then do semantic analysis, we don't consider the problem of parameters' type at this time. We just use actual parameters in the function call to replace the general symbol, and any type error will be detected in type checking procedure.

When we encounter any error in this section, we will report the error and stop the function elimination and return to the original AST. Maybe this method will lead the function to not be eliminated; however, if function elimination returns an error, the program will not work. In fact, it will cause much more problems

and errors in the result if we continue the translation process. So, this error recovery technique is reasonable.

Only after correct semantic check and generating a new AST with correct function eliminition, will the system continue to do other semantic analysis. Table 3-10 lists the error information and its recovery.

Error category	**Error reporting**	**Operation**
semantic	*Function parameters' number error.*	Stop function elimination and return to the original AST.
	No such function.	Stop function elimination and return to the original AST.

Table 3-10 Error information in function elimination

3.5 Summary

The whole process can help eliminate functions in the AST, which can simplify the work in the run-time. Till now, we only focus on GIPL AST. If there is a SIPL AST, as we mentioned in chapter 1, we need to translate it into a GIPL AST. In next chapter, we will introduce the translator generation.

4. SIPL AST Translator Generation

4.1 Introduction

The GIPL is the generic language of the Lucid family of languages, which only includes a basic set of operations, such as @ and #. However, practically, we always use Lucid languages (or other intensional programming languages, or SIPLs) that have a set of domain-specific operations. GIPL accompanied with an extended set of operations becomes a SIPL. Different sets of operations correspond to different SIPL versions. For example, here in this project, extended operations *first, next, prev, fby, wvr, asa*, and *upon* make the GIPL upto a SIPL named Indexical Lucid.

For different SIPLs, we have to generate different parser and construct different AST because the RIPE environment of the GIPSY system will display and evaluate the Lucid source code based on the result of the parser and the tree. These extended operations will be treated as the basic unit of that SIPL. However, the GEE module only takes the basic tree structure; that is the GIPL- like tree. So, we have to develop tree translators for each kind of SIPL. The translator will translate the SIPL-like tree output of that SIPL parser, which contains the nodes of extended operations, into GIPL-like tree, which only contains only nodes of basic operations.

The GEE only treats GIPL ASTs. Any SIPL AST constructed by a related generator has to be translated into the GIPL AST with the corresponding translator. Hence, this Indexical Lucid SIPL AST translator is also acting as the template for the other new-defined SIPL AST translators. Table 4.1 illustrates a translation.

```
A
where
      dimension d;
      A= 1 fby .d (A+1);
End
```

(a) A SIPL program

```
start
   where
        id: A
        dimension
            d
        assign
            id: A
            fby
                const: 1
                dimension: d
                add
                    id : A
                    const : 1
```

(b) The corresponding SIPL AST

SIPL AST Translator

```
start
   where
       id: A
       dimension
           d
       assign
           id: A
           if
               eaqul
                   hash
                       d
                   const : 0
               const: 1
               at
                   add
                       id : A
                       const : 1
                   dimension: d
                   minu
                       hash
                           d
                       const: 1
```

(c) The GIPL AST after translating

Table 4-1 A SIPL program and its translation

In this section, we will introduce the translator generation. Figure 4.1 illustrates the architecture of the translator.

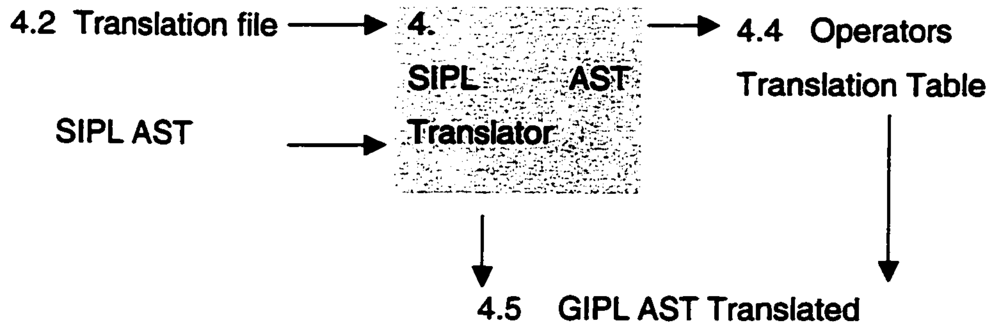


Figure4-1 The architecture of SIPL AST Translator Generation

The result of the translator is actually a "library" of functions in which each translates a specific operator. This would open the door for writing programs using any intensional operator.

4.2 The Style Definition for User Input

Being an automatic and flexible translator generation, it does not know what kind of specific operators will be translated before users input the translation file. The translation file can define how a specific operator is to be translated into GIPL. Users can arbitrarily define these rules. However, we give some constraints on the input file style.

- The file is a text file, with the suffix **.txt.**
- The nesting is not permitted in the input file. This means each translation must take place by basic operators in the GIPL.
- It is permitted to have several translation rules in one file.
- The expression is: **operator: translation statement**.
- **Operator** is just its name. It does not include its operands.
- In **translation statement**, it uses "**R**" to stand for **right operand**, "**L**" to stand for **left operand,** "**D**" to stand for **dimension.** It does not use specific operands in the translation statement.

- The use of the **@**, **#**, **if**, and **where** in translator is the same as that used in the GIPL.

Table 4.2 gives an example

```
Tran.txt

first: R @ .D 0

fby: if (#.D==0) then L else R@.D(#.D-1);

wvr: L @ .D T
       where
          T=if (#.D==0) then U else (U@.D(T+1))@ .D (#.D-1);
          where
             U=if R then #.D else U @ .D (#.D+1);
          end;
        end;

asa: (L @ .D T) @ .D 0
       where
          T=if (#.D==0) then U else (U@.D(T+1))@ .D (#.D-1);
          where
             U=if R then #.D else U @ .D (#.D+1);
          end;
       end;
```

Table 4-2 An example of the input file

In the example, we use "**R@.D 0**" to translate operator "**first**". Normally, operator "**first**" can be expressed in "**first .d A**". It only has a right operand, and according to translation rules, we only use keyword '**R**' and '**D**' to translate it. The other transitions obey the same convensions.

4.3 The SIPL Translator Generation Syntactical Specification

In this section, according to the input file definition and GIPL grammar, we give the SIPL translator generation grammatical specification.

4.3.1 Lexical conventions

The lexical conventions is defined as followings:

- **Case Insensitive**

  In translator, it is case insensitive.

- **Valid Characters**

  space, +, -, *, /, %, (,), !, &, |, #, @, ;, :, ., _, =, <, >, 0..9, a..z, A..Z

- **Reserved Words**

  if, then, else, where, end, L, R, D

- **White Space**

  Space, tab, return and new line are all treated as white space and will be skipped. We need one space to separate tokens.

- The maximum length of a number (real and integer number) is limited to 10 characters and an operator name 30 characters. When the input exceeds the limits, it will be considered wrong and the treatment will be specified later in Chapter 4.6.

- Other characters will be treated as illegal characters and an error message will be displayed.

4.3.2 Productions of Grammar

Based on the work that a member did on the GIPSY, we can get the basic grammar of GIPL [5], which eliminates left-recursion. According to the specific requirements in the AST translator generation and a great deal of analysis, we present the grammar as in Table 4-3.

<TranRule>	::=	<Rule> <RR>
<RR>	::=	<Rule> <RR>
		ε
<Rule>	::=	Id : <E>
<E>	::=	<term> <E1>
		<sign> <term> <E1>
		if <E> then <E> else <E> ; <E1>
		# . D <E1>
<E1>	::=	<addOp> <term> <E1>
		<relOp> <E> <E1>
		where <Q> <QR> end ; <E1>
		ε
<term>	::=	<factor> <term1>
<term1>	::=	<multOp> <term> <term1>
		ε
<QR>	::=	<Q> <QR>
		ε
<Q>	::=	Id = <E> ;
<factor>	:=	Id
		IntType
		RealType
		(<E>)
		L
		D
		R
		! (<E>)
<sign>	::=	+
		-
<addOp>	::=	+
		-
		||
<multOp>	::=	*
		/
		%
		&&
<relOp>	::=	<
		>
		<=
		>=
		==
		!=

Table 4-3 The grammar of the SIPL Translator Generation

4.4 Building Operator Translation Table

After finishing the grammatical specification, as shown in Figure 4.1, in this section, we discuss how operator translation tables are built. It can be realized in the following method: ***public void Parse() throws IOException.***

4.4.1 Table-Driven Parsing

We use table-driven parsing for easy maintainance. The table for lexical and syntactic can be found in Appendix II and Appendix III. Each token from the input file can be read by the parser and be treated according to grammar specifications.

In table-driven parsing, it is most important to insert suitable action in the proper position. For example, in Rule 8, the original is: **<E1> ::= @ . D <E>**. When the parsr encounters this rule, it will first create a structure of **@** operator; and then, it will deal with finishing the **@** expression. So, we modify Rule 8 as such: **~3<E1> ::= @ . D <E> ~4**. The operation of **~3** can be seen in Excerpt 4-1, and the operation of **~4** can be seen in Excerpt 4-2.

```
if (sub.indexOf('~')==0) {
        if (sub.charAt(1)=='0') substr=CreateOpe();//increase an operator
        else if (sub.charAt(1)=='1') substr=CreateIf();  /*if statement*/
        else if (sub.charAt(1)=='2') substr=CreateHash();  /*# statement*/
        else if (sub.charAt(1)=='3') substr=CreateAt();  /*@ statement*/
        else if (sub.charAt(1)=='4') substr=CreateWhere();/*where statement*/
        else if (sub.charAt(1)=='5') substr=CreateEquel(); /*= statement */
        else if (sub.charAt(1)=='6') substr=CreateAdd();  /* Addop statement*/
        else if (sub.charAt(1)=='7') substr=CreateRel();  /* Relop statement*/
        else if (sub.charAt(1)=='8') substr=CreateFac();  /* Factor statement*/
        else if (sub.charAt(1)=='9') substr=CreateMul();  /*Mulop statement*/
        else if (sub.charAt(1)=='A') substr=CreateSign();  /*sign statement*/
  }
```

Excerpt 4-1 Program that explains "~" is at the beginning of a rule

```
while (temp.indexOf('~')==0){
        if (temp.charAt(1)=='1') DoCond();
        else if (temp.charAt(1)=='2') DoThen();
        else if (temp.charAt(1)=='3') DoElse();
        else if (temp.charAt(1)=='4') DoAt();
        else if (temp.charAt(1)=='5') DoEqual();
        else if (temp.charAt(1)=='6') DoBrace();
        else if (temp.charAt(1)=='7') DoNotBrace();
        temp=SytacStack.pop().toString();
  }
```

Excerpt 4-2 Program that explains "~" is in the middle of a rule

In this parser, the position in which we will insert the operation and its operation number is shown in Table 4-4.

~0<Rule> ::= Id : <E>

~A<E> ::= <sign> <term> <E1>

~1<E> ::= if <E> **~1** then <E> **~2** else <E> **~3** ; <E1>

~2<E> ::= # . D <E1>

~6<E1> ::= <addOp> <term> <E1>

~7<E1> ::= <relOp> <E>

~3<E1> ::= @ . D <E> **~4**

~4<E1> ::= where <Q> <QR> end ;

~8<term> ::= <factor> <term1>

~9<term1> ::= <multOp> <term> <term1>

~5<Q> ::= Id = <E> **~5**

<factor> ::= (<E>) **~6**

<factor> ::= ! (<E>) **~7**

Table 4-4 Rules to insertting operation and its number

4.4.2 Building a Forest for Operators

The translator parser will build an AST for each operator. To do this, it uses a stack in which it pushes nodes after they have been created. When it finds a parent for them, it pops the children from the stack and adds them to the parent, and finally pushes the new parent node itself.

The main task now is to build and manage a sub abstract syntactic tree for operators. We use Hashtable to manage these separate small trees. Excerpt 4-3 illustrates the definition of Hashtable, and the name of operator is the key code of the Hashtable.

```
Hashtable TranTable = new Hashtable( );
TranItem Sym;
```

(a) Definition TranTable as a Hashtable

```
class TranItem{
   String TranName;
   SimpleNode TranEntry;
   public TranItem(){
      TranName="";
      TranEntry=null;
   }

   public TranItem(String TranName, SimpleNode TranEntry){
      this.TranName=TranName;
      this.TranEntry=TranEntry;
   }
}
```

(b) Definition of the item in the TranTable

Excerpt 4-3 Data structure of managing operators and their AST

As we discussed above, the **Create+name()** method will create a new branch of a sub AST of the correspongding operator and connect it with the original AST. For example, if the parser encounters the # expression, it will call the **CreateHash()** method as in Excerpt 4-4. This function will serve to create a branch: #.D, and the return value is the # node. To connect it with the original one, if there is no other node, that means it is the beginning node of the corresponding operator, then method **CreateHash()** will regard the # node as the corresponding operator's entry address. Otherwise, the method **CreateHash()** will push the # node into stack to wait.

```
String CreateHash() throws IOException{
    String s5="";
    String s6;

    s6=Rule(SyTable(SyTableLn)(SyTableCo));
    s5=s6.substring(s6.indexOf('=')+2);

    C0=new SimpleNode(3); //create # node
    C1=new SimpleNode(25); //create D node;
    C1.image="D";
    C1.parent=C0;
    C0.jjtAddChild(C1,0);
    if (CurrentNode==null)
    {
      C0.parent=null;
      Sym.TranEntry=C0;
      TranTable.put(Sym.TranName,Sym);
    }
    else  TranStack.push(C0);//push the "#" node
    return s5;
}
```

Excerpt 4-4 The CreateHash() method

The **DO+name()** method will deal with some medial conditions. For example, when the parser finishes the **condition** analysis in the **if** expression, it will call the **DoCond**() method as in Excerpt 4-5. This method will pop the value in the stack, and connect the current node with the original AST to make an integrated AST for the corresponding operator.

```
void DoCond(){
      SimpleNode T1;

      T1=(SimpleNode)TranStack.pop();
      if ((!TranStack.empty())&&(!IsOpe(T1.id)))
            T1=(SimpleNode)TranStack.pop();
      T1.parent=CurrentNode;
      CurrentNode.jjtAddChild(T1,0);

      if (!TranStack.empty()) TranStack.clear();
  }
```

Excerpt 4-5 The Docond() method

After the parsing, we will get a Hashtable with all the operators' names and the first node of the AST. Based on the example of tran.txt in Table 4-2, we will get Figure 4-2 as following. Each operator has an entry address that can extend a small AST. Then, all small sub trees can compose a forest of operators.

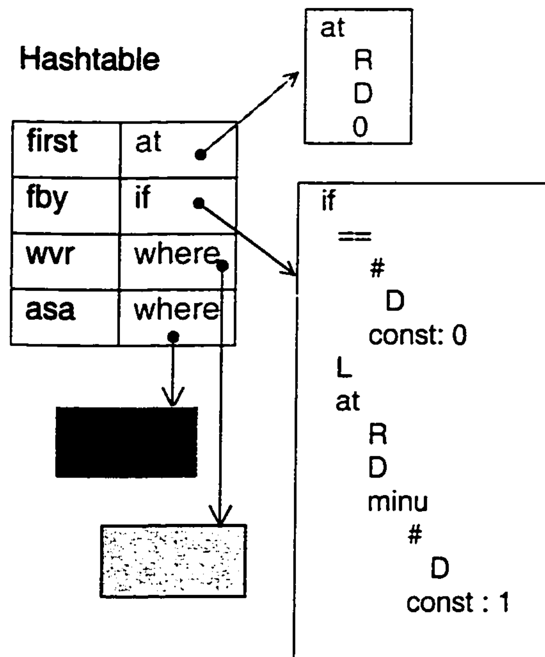


Figure 4-2 The forest of translation operators

4.5 Specific Operator Translation

Now, we have the operator translation table and that means we have a "library". As shown in Figure 4-1, in this section, we will discuss how to realize operators' translation. The method: ***SimpleNode AutoTran(SimpleNode SiplNode, int Num)*** will realize it.

In this method, the input is a SIPL AST. When the translator encounters SIPL operators, it will generate a new AST according to the translation rule in Table 4-2 and use the new one to replace the old node without changing the meaning of the original SIPL AST.

In the translation process, there are five important methods for this purpose:

- **void Duplicate(SimpleNode root, SimpleNode Ori_tree, int Child_Num).** This method will duplicate the AST of the corresponding operator through the "library". In the library, the small tree of each operator is with **L, R,** or **D** general operands. However, each specific node in the SIPL AST has real operands, so we need to change the original small tree with different oprends. To prevent the basic operator translation tree from changing, we should duplicate the tree before using it.

- **void Replace(SimpleNode ReTree, String para).** This method is used to replace one operand - except dimension. For example, the translation is just: **DD X→ R@ 0**. The method will use operand **X** to replace the **R**. This is unusual, but we should still consider it.

- **void Replace(SimpleNode ReTree, SimpleNode Dim).** This method is used to replace one operand - only dimension. For example, the translation is: **HH.time→ #.D**. The method will use dimension **time** to replace the original dimension **D**.

- **void Replace(SimpleNode ReTree, String para, SimpleNode Dim).** This method is used under the condition that there is one operand and one dimension. For example, if the translation is **first.d X→ R @ .D 0**, there is one right operand and one dimension, then this method will replace **R** with **X**, and **D** with **d**.

- **void Replace(SimpleNode ReTree, String Left, String Right, SimpleNode Dim).** This method is used under many normal conditions that there are left and right operands and dimension. For example, if the translation is **X fby .d Y→ if (#.D)==0 then L else R@.D (#.D)-1**, then the method will replace **L** with **X**, **R** with **Y**, and **D** with **d**.

Figure4-3 shows the condition at present.

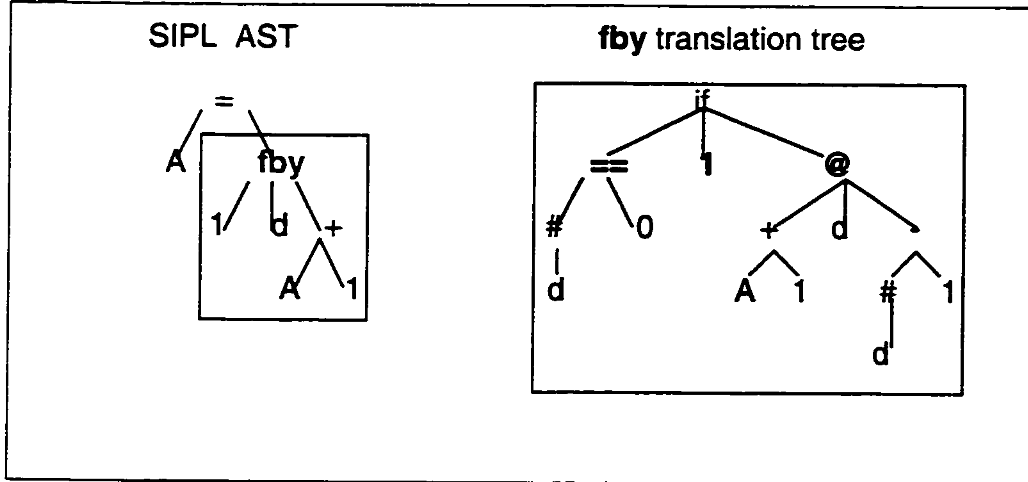


Figure 4-3 Slice after replacing the general operand with the real operand

The remaining task is to use the "**if**" node to replace the "**fby**" node in the SIPL AST. Excerpt 4-6 explains this operation.

```
SimpleNode AutoTran(SimpleNode SiplNode, int Num){
        ......
        Rp1=if ;
        Rp1.parent=SiplNode.parent;

        ((SimpleNode)SiplNode.parent).children[Num]=Rp1;
        ......
}
```

Excerpt 4-6 Program to realize translation

4.6 Error Reporting and Error Recovery

In this part, errors are divided into two catagories: one is error about parsing, the other is error about translation.

When there is an error in the parsing process, the parser will report the error, skip it, keep on reading the next token until it encounters the token that it needs at the error point, and then resume parsing. This method may cause some program to not be parsed, but some widely used compilers, such as Java, cannot also find all preceding syntactic errors after encountering the first few syntactic errors. Moreover, some syntactic errors have complicated relationships. This makes it impossible to identify these errors. So, this error recovery technique is reasonable.

Only after correct parsing and generating a correct "library", can the generator continue to do translations. During the translation process, if there is an error, the translator will report the error and return an empty tree. The translation operation will be stopped.

Table 4-5 lists the error information and its recovery.

Error category	**Error reporting**	**Operation**
Parsing(lexical)	_Illegal Char._	Skip it and do not output the token
	Illegal number	Do not output the token
	String is too long	Do not output the token
Parsing(syntax)	_Line:* Sytactic error: *_	Skip it.
	Line:* Parsing resume at:*	Encounter need token, parsing resume.
Translation	_Translation error: Error parameters._	1>The operand number>3, return null tree.
	Translation error: No such operator.	Return null tree.

Table 4-5 Error information in the Translator generation

4.7 Summary

We develop a parser to help translation by which GEE can only focus on GIPL AST. We all know the translation process will take time and space. However, if we make the system can directly deal with SIPL; then, if there is even a new operator introduction, we have to modify our whole system. So, the present design, having a translator, can improve the system flexibility.

After algorithm, we will discuss testing to make sure the design. So, in next chapter, we will introduce result evaluation based on three parts.

5. Result Evaluation

Quality is always the most important issue in software. In order to control bugs to the lower level, testing goes through the whole of semantic analysis and SIPL AST translator development. The testing mechanism we employed was a bottom-up testing process. The procedure of the semantic analysis and SIPL AST translator development and testing is as followings:

- Integrate with the parser part, begin the semantic part;
- Receive the GIPL AST from the parser, build the dictionary and print out the content of the dictionary;
- Modify type checking rules;
- Do type checking, add the type attribute into the dictionary;
- Modify rank analysis rules;
- Do rank analysis, add the rank attribute into the dictionary;
- Introduce function after making sure that the semantic analysis for the GIPL AST is correct;
- Build a GIPL AST with function;
- According to the AST, use general symbols to replace the function parameters;
- Build a function table, print out the content of the table;
- Eiminate the functions in the AST, print out the new AST;
- Add the function elimination into the original semantic analysis part;
- Input a Lucid program with functions, get the AST by the parser, do the semantic analysis after function elimination. Make sure that the whole process is correct;
- Introduce SIPL;
- Modify the SIPL AST translator generation grammar;
- Produce the SIPL AST translator parser;

- Users input translation file;
- The translator parser produces a translation table, prints it out to make sure that the translation is correct;
- Produce the translator;
- Generate a SIPL AST;
- Feed the SIPL AST to the translator, make sure the SIPL AST is translated into the GIPL AST correctly and print it out;
- Integrate the translator part with the original semantic analysis part;
- Input a SIPL, get the SIPL AST by the SIPL parser, do the semantic analysis after translation. Make sure the whole process is correct;
- Input a SIPL with functions and get a SIPL AST by the SIPL parser. First, eliminate the functions; then, translate it into a GIPL AST; finally, do the semantic analysis. Make sure that the whole process is correct.

Our testing of the module goes through every step of this process. Basically, we use black and white box, and develop different testing stubs to test the products at every stage.

5.1 Semantic Analysis

In this section, we will discuss evaluation results on the semantic analysis. First, we will introduce how to execute semantic analysis. We integrate the semantic analyzer with the parser, so when we execute the parser as in [5], the semantic part will be executed automatically. The following is the procedure of executing a Lucid program:

- Set up JDK1.3 environment under either Unix/Linux or Windows system;
- Compile all generated java files, input "javac *.java" under working directory;
- Input the command "**Java Lucid *.txt –G**" for the GIPL or the command "**Java Lucid *.txt –S**" for the SIPL;

In the semantic analysis step, the test is concerned with building the dictionary correctly and reporting errors effectively. We will give some typical examples in which include all kinds of semantic errors.

Test1: In this example, in the "**if**" expression, the type of "**then**" and "**else**" part is different; this is illegal. Also, we can check the type and the rank of each identifier. Specific conditions can be seen in Table 5-1.

```
X
where
  dimension d,m;
  x = if (#.d)==0 then 2 else (y+1.5)@.d (#.d)-1 fi
  where
    y=(#.m)-1;
  end;
end
```

(a) A Lucid program

```
Semantic error: If Statement's type does not match!

0 __ Start __ __ 2 __ __ START
1 __ d __ dimension __ 2 __ __ null
2 __ m __ dimension __ 2 __ __ null
3 __ x __ identifier __ 0 __ d,m, __ IF
4 __ y __ identifier __ 0 __ m, __ MIN

1 semantic errors.
0 semantic warning errors
```

(b) The test result of the above program

Table 5-1 The Lucid program and its test result of test 1

Comments: The semantic analyzer correctly judges the type mismatch of the "**if**" expression and reports the error. The fouth column of the dictionary is the type of identifiers; 0 stands for integer, 1 stands for real. The fifth column in

the dictionary is the rank of identifiers. The dictionary shows that the attributes of each variable are correct.

Test2: In this example, there are errors in the expressions before the "**where**" clauses because some identifiers in the expressions are not defined under the follow "**where**" clauses. This is illegal. Also, we can do type checking in the complicated relationship of identifiers. Specific conditions can be seen in Table 5-2.

```
y+t+ia
   where
        y=z+o+i
        where
           z=6;
           o=9.6;
        end;
        t=z+3
        where
           z=7;
        end;
   end
```

(a) A Lucid program

```
Semantic error: Identifier i is not defined!
Semantic error: Identifier ia is not defined!

0 __ Start __ __ 2 __ __ START
1 __ y __ identifier __ 1 __ __ ADD
2 __ z __ identifier __ 0 __ __ CONST : 6
3 __ o __ identifier __ 1 __ __ CONST : 9.6
4 __ t __ identifier __ 0 __ __ ADD
5 __ z __ identifier __ 0 __ __ CONST : 7

2 semantic errors.
0 semantic warning errors.
```

(b) The test result of the above program

Table 5-2 The Lucid program and its test result of test 2

Comments: The semantic analyzer correctly judges whether an identifier is defined or not and reports the errors. For example, the identifier "**ia**" in the expression "**y+t+ia**" is not defined. In the dictionary, the fouth column is the type of identifiers; 0 stands for integer, 1 stands for real. The result in the dictionary shows that the type of each variable is correctly analyzed.

Test3: In this example, there is a problem about identifier overriding. This is illegal. Specific conditions can be seen in Table 5-3.

```
y+z
   where
        y=z+1
        where
          z=6;
        end;
   end
```

(a) A Lucid program

```
Semantic Warning: z overridding!
Semantic error: Identifier z is not defined!
0 __ Start __ __ 2 __ __ START
1 __ y __ identifier __ 0 __ __ ADD
2 __ z __ identifier __ 0 __ __ CONST : 6
1 semantic error.
1 semantic warning error.
```

(b) The test result of the above program

Table 5-3 The Lucid program and its test result of test 3

Comments: There are two "**z**"s in a same scope, and the definition of "**z**" is in the second "**where**" clause. First, the "**z**" in the definition "**y=z+1**" is overriding; second, the "**z**" in the expression "**y+z**" is not defined. The result shows that the semantic analyzer correctly judges and reports the errors.

Test4: In this example, there are two "z"s in different scopes. This is legal. Also, we can check the type and the rank of each identifier. Specific conditions can be seen in Table 5-4.

```
x+y+t
where
  dimension d;
  x = if (#.d)==0 then 2.5 else (x+1) @.d (#.d)-1 fi;
  y=z+x
      where
         z=3;
      end;
  t=z+3
      where
         z=5.0;
      end;
end
```

(a) A Lucid program

```
Second Semantic Check!
0 __ Start __ __ 2 __ __ START
1 __ d __ dimension __ 2 __ __ null
2 __ x __ identifier __ 1 __ d, __ IF
3 __ y __ identifier __ 1 __ d, __ ADD
4 __ z __ identifier __ 0 __ __ CONST : 3
5 __ t __ identifier __ 1 __ __ ADD
6 __ z __ identifier __ 1 __ __ CONST : 5.0
0 semantic errors.
0 semantic warning errors.
```

(b) The test result of the above program

Table 5-4 The Lucid program and its test result of test 4

Comments: This is a complicated example. First, there are two "z"s in two scopes: "**y=z+x**" and "**t=z+3**". The system gives different ID to each "**z**". Second, the type and rank of the identifier "**y**" cannot be known if the

attributes of the identifier '**x**' are not available; so, the system needs semantic analysis again. Third, the type and rank of all identifiers are correct. All of results illustrate that the semantic analyzer work.

5.2 Function Elimination

In this section, we will test function elimination. Because function elimination connects with semantic analysis, there is no additional procedure needed to execute it. We will report each step in Table 5-5.

```
merge.d (A,B)+1
where
    Dimension d;
    merge.a (x,y)=xx+yy;
    where
        xx=x @ .a 1;
        yy=y @ .a 1;
    end;
end
```

(a) A Lucid program

```
WHERE
 ADD
  Function : merge
   DIMENSION
    DIMENSION : d
   Paras
    ID : A
    ID : B
  CONST : 1
 DIMENSION
  ID : d
 ASSIGN
```

```
Function : merge
 DIMENSION
  DIMENSION : a
 Paras
  ID : x
  ID : y
WHERE
 ADD
  ID : xx
  ID : yy
 ASSIGN
  ID : xx
  AT
   ID : x
   DIMENSION : a
   CONST : 1
 ASSIGN
  ID : yy
  AT
   ID : y
   DIMENSION : a
   CONST : 1
```

(b) The corresponding AST of the above program

```
WHERE
 ADD
  Function : merge
   DIMENSION
    DIMENSION : d
   Paras
    ID : A
    ID : B
  CONST : 1
 DIMENSION
  ID : d
 Function :
```

(c) The AST after building the function table

```
WHERE
 ADD
  ADD
   ID : xx
   ID : yy
  CONST : 1
 DIMENSION
  ID : d
 Function :
 ASSIGN
  ID : xx
  AT
   ID : A
   DIMENSION : d
   CONST : 1
 ASSIGN
  ID : yy
  AT
   ID : B
   DIMENSION : d
   CONST : 1
```

(d) The AST after eliminating the function

```
0 __ Start __ __ 2 __ __ START
1 __ d __ dimension __ 2 __ __ null
2 __ xx __ identifier __ 0 __ __ AT
3 __ yy __ identifier __ 0 __ __ AT
0 semantic errors.
0 semantic warning errors.
```

(e) The dictionary after all above processes

Table 5-5 The semantic analysis process of a Lucid program with functions

Comments: The whole process works very smoothly. Each step can be tracked by the output and can be validated as being correct.

5.3 SIPL AST Translator

To the SIPL, first we must use the SIPL AST translator. The following is the procedure of executing the Translator:

- Set up JDK1.3 environment under either Unix/Linux, Windows 2000, or MacOS X system;
- Edit translation grammar file with pure text editor; the file name has the suffix of .txt, such as "tran.txt";
- Press "run" button, enter the translation file name with path;
- Modify the translation file to customize the error messages;
- The remaining steps will be with the semantic analysis together.

Regarding result evaluation, even though during the real run time we did not separate each step, here we will give the results of each step. All results will be shown in Table 5-6.

```
bb: # .D
first: R @ .D 0
fby: if (# .D==0) then L else R@.D(# .D-1);
wvr: L @ .D T
   where
     T=if (# .D==0) then U else (U@.D(T+1))@ .D (# .D-1);
     where
       U=if R then # .D else U @ .D (# .D+1);
     end;
   end;
asa: (L @ .D T) @ .D 0
   where
     T=if (# .D==0) then U else (U@.D(T+1))@ .D (# .D-1);
     where
       U=if R then # .D else U @ .D (# .D+1);
     end;
   end;
```

(a) The translation rule file: Tran.txt

```
Please input the file name:
There are 5 operators in the HashTable.
The following is example of operator: 'asa' !
 WHERE
  AT
   AT
    ID : L
    ID : D
    ID : T
   ID : D
   CONST : 0
  WHERE
   ASSIGN
    ID : T
    IF
     EQ
      HARSH
       ID : D
      CONST : 0
     ID : U
     AT
      AT
       ID : U
       ID : D
       ADD
        ID : T
        CONST : 1
      ID : D
      MIN
       HARSH
        ID : D
       CONST : 1
   ASSIGN
    ID : U
    IF
     ID : R
     HARSH
      ID : D
     AT
      ID : U
      ID : D
      ADD
       HARSH
        ID : D
       CONST : 1
total 0 errors
total 0 warn errors
```

(b) The corresponding translation table of file Tran.txt

```
x fby .d y
where
        dimension d ;
        x = 1.5;
        y = 1;
end
```

(c) A SIPL example

```
This is the original AST with fby!
WHERE
 Function : fby
  ID : x
  DIMENSION : d
  ID : y
 DIMENSION
  ID : d
 ASSIGN
  ID : x
  CONST : 1.5
 ASSIGN
  ID : y
  CONST : 1
```

(d) The corresponding AST of the above program

```
This is the translated AST:

WHERE
 IF
  EQ
   HARSH
    DIMENSION : d
    CONST : 0
   ID : x
  AT
   ID : y
   DIMENSION : d
   MIN
    HARSH
     DIMENSION : d
     CONST : 1
 DIMENSION
  ID : d
 ASSIGN
  ID : x
  CONST : 1.5
 ASSIGN
  ID : y
  CONST : 1
total 0 errors
total 0 warn errors
```

(e) The AST after translating the SIPL to the GIPL

```
0 __ Start __ __ 2 __ __ START
1 __ d __ dimension __ 2 __ __ null
2 __ x __ identifier __ 1 __ __ 1.5
3 __ y __ identifier __ 0 __ __ 1

0 semantic errors.
0 semantic warning errors.
```

(f) The dictionary after all above processes

Table 5-6 The result of applying the SIPL AST translator

Comments: First, when we arbitrarily input the correct translation rule, the results show that the program generates a correct translation hashtable. If there are some errors in the input text file, the program will report errors and ask the user to fix it. Second, a SIPL can be correctly translated into the GIPL. Last, after translation, the information in the dictionary is correct. So, this SIPL AST translator is flexible and reliable.

5.4 Summary

In this chapter, we have provided many test cases, which can ensure the quality of our project. Till now, our introduction is separated, so in next chapter, we will introduce the relationship among components and how to integrate them.

6. Integration with other components in GIPSY

GIPSY is a large system. Each part is responsible for a different task. So, it is also very important to integrate the different parts especially developed by different people smoothly. In Figure1-3, we can see the architecture of the whole GIPSY system. Then, in this chapter, we provide Figure 6-1 to illustrate the specific relationship among the semantic analyzer, the translator, and the parser.

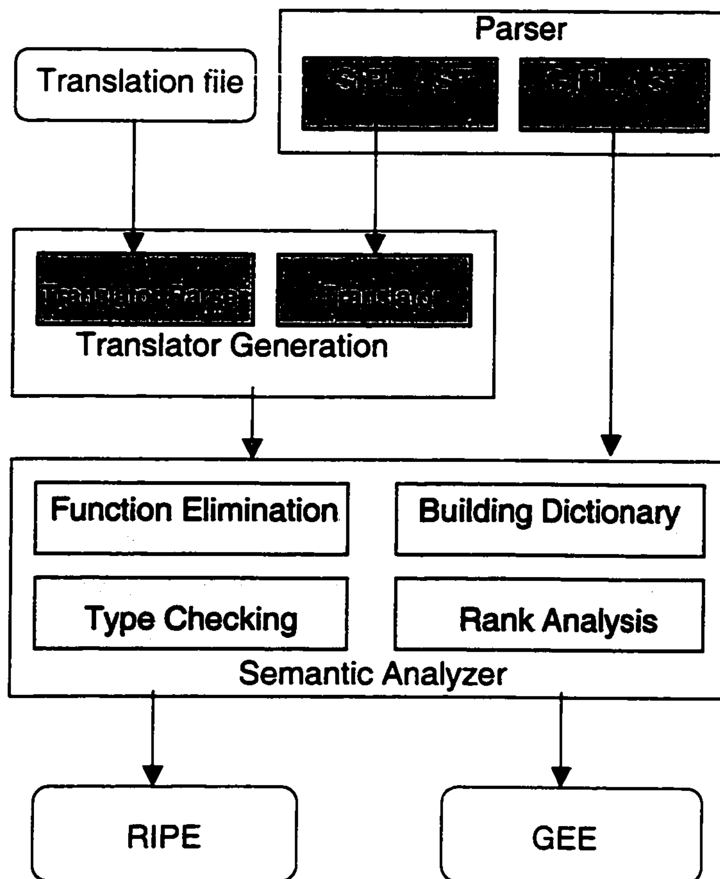


Figure 6-1 The relationship of the Semantic Analyzer and other components

6.1 Intergrate with the Parser

According to [5], we can know that there are two outputs after parsing. One is the entry node of the GIPL AST; the other is the entry node of the SIPL AST.

As shown in Figure6-1, the semantic analyzer is the back end. There is also a **façade** class as the interface of the Parser. To integrate the semantic analyzer with the parser is just to modify the **façade** class. Excerpt 6-1 explains the modification.

```
public class Facet {
        ......
        SimpleNode getNode(){
              return simpleNode;
        }
        ......
}
```

(a) The code added for semantic in the Facet class

```
Public class Lucid{
        ......
        SimpleNode ref;
        Semantic semantic = new Semantic();
        //new add for semantic
        ref=facet.getNode();
        semantic.SetupDictionary(ref);
        ......
}
```

(b) The code added for semantic in the Lucid class

Excerpt 6-1 Modification for integrating the semantic analyzer with the Parser

6.2 Relationship with the RIPE

In the RIPE, normally, it will use the AST from the Parser part. It will also show the original Lucid language including the SIPL with the specific operators. So, it acturally does not need the SIPL translator. It can directly use the SIPL AST and the GIPL AST. However, it still uses the output of the semantic analyzer because it needs the ID of each identifier. So, we can only provide the dictionary for the RIPE.

6.3 Output for the GEE

The GEE is run time environment, so there is a great relationship between the semantic analyzer and the GEE. There is no attribute in the original AST and all attributes are added later. This makes the GEE use the AST directly because there is enough information to calculate the value of each identifier.

Also, because the AST after semantic analysis is a basic AST, it only has @, #, if, and where, there is no function data structure. So, to some extent, this simplifies the GEE process.

However, the whole GIPC should have its framework. The semantic analyzer is only a part of this sub system. For example, there are many small Java function calls in the GIPC. All of these Java functions should be considered in the whole system, and the whole GIPC should have interface with the GEE and not only the samentic analyzer. All of these will be considered in next stage design.

6.4 Summary

We know how to integrate the semantic analyzer, the parsers, the SIPL AST translator, RIPE and GEE. All are what we have done to this project. Then, what are we will do in the future. In next chapter we will introduce the future work.

7. Future Work

Up to now, we finished the basic compiler part of the GIPSY system, which makes it work. However, in a complex system, this is only the first step. There is still much further work to be done.

7.1 Permit Nested Input in the SIPL Translator Part

In the translator generation, a rule used in user input style definition is that "*The nesting is not permitted in the input file. This means each translation must take place by basic operators in the GIPL*". We don't know which kind of operators will be defined, so if there is a condition such as those in Table 7-1, the translator will not understand.

```
Tran.txt
        first: R @ .D 0
        next: R @ .D (#.D +1)
        fby: if (#.D ==0) then L else R @ .D (#.D)-1;
        wvr: L @ .D T
              where
                 T = U fby .D U @ .D  (T+1) ;
                 U = if R then #.D else next U ;
              end
        asa: first.D ( L wvr.D R)
```

Table 7-1 The translation file with nesting

The limitation brings convenience to implement translation. However, it will ask users to translate the file first before they input it. To some extent, this kind of translation is complex.

So, in the next step, we should improve the AST translator generation with permitting nested input.

7.2 Recursion in Lucid Programs

Recursion is always a difficult aspect for a compiler. From this point of view, Lucid is not an exception, as recursion is very popular in many Lucid programs. However, a Lucid progam that uses recursion is considered a bad Lucid program, because in Lucid you should use dimensions instead of recursion. Consequently, the fact that we cannot handle recursion is not important -- in fact, its absence will encourage people to write better Lucid. Additionally, as we discussed above, we do not consider the recursive condition at present because we use function elimination algorithm to simplify the analysis procedure.

In the future, because of the widespread use of recursion in many Lucid legacy programs, we must resolve this problem. The basic idea is: if the program is without recursive, we can use the algorithm used now. However, if there is a recursive program, we must translate it into Java function and give the problem to the Java compiler.

7.3 Graphic Interface Design for GIPC

Now, the interface of the Parser part is under DOS environment with a command line interface. With the develoment of the whole project, there will be more and more functions added to the GIPC part. To some extent, how to co-operate the parser, the translator, and the semantic analyzer is very difficult for users. So, in the future, we consider designing a graphic interface for the GIPC. In this interface, we can provide the selection as to what kind of Lucid language the users must use. If it is SIPL, we will remind the users to first input the translation file. Then, we will permit the users to use Java functions and to integrate them with the Lucid compiler.

8. Bibliography

[1] Joey Paquet. Intensional Scientific Programming. Ph.D. Thesis, Departement d'Informatique, Universite Laval, Quebec, Canada, 1999.

[2] Joey Paquet, Peter Kropf. The GIPSY Architecture, Proceedings of Distributed Computing on the Web, Quebec City, Canada, 2000.

[3] P.G.Kropf. Overview of the Web Operating System (WOS) project. *In Proceedings of the 1999 Advanced Simulation Technologies Conference (ASTC 1999)*, pages 350-356, San Diego, California, April 1999.

[4] J.Plaice and W.W.Wadge. A new approach to version control. *IEEE transactions on Software Engineering*, 3(19):268-276, 1993.

[5] Chun Lei Ren. Parsing and Abstract Syntax Tree Generation in the GIPSY System. M.Sc, Thesis, Computer Science Department, Concordia University, Quebec, Canada, September 2002.

[6] Raganswamy Jagannathan, Chris Dodd. GLU programmer's guide. Technical report, SRI International, Menlo Park, California, 1996.

[7] Peter Grogono. A simple demand-driven interpreter. Unpublished, May 2002.

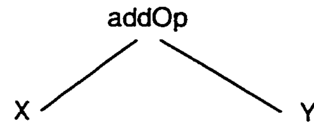
[8] Brian T. Kurotsuchi. Welcome to the wonderful world of design patterns. http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/, 2002.

[9] Douglas C. Schmidt. Design Patterns, Pattern Languages, and Frameworks. http://www.cs.wustl.edu/~schmidt/patterns.html, 2002.

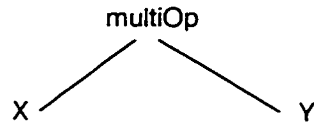
[10] The Java Tutorial. http://java.sun.com/docs/books/tutorial/, 2002.

9. Appendix I: GIPL AST structures

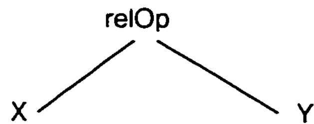
1. X **addOp** Y



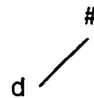
2. X **multiOp** Y



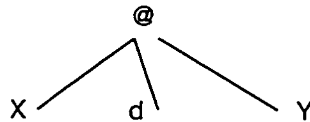
3. X **relOp** Y



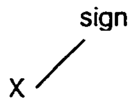
4. **#.** d



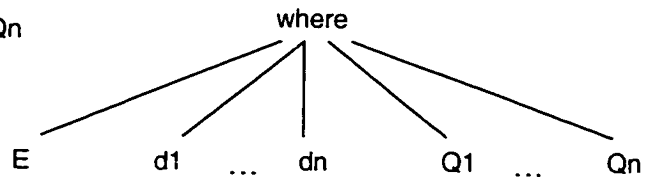
5. X **@.** d Y



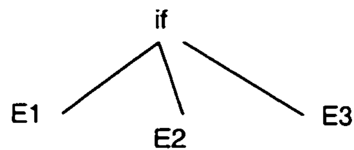
6. **sign** X



7. E **where** d1, .. dn Q1, .. Qn



8. **if** E1 **then** E2 **else** E3 **fi**



10. Appendix II: Table used for Lexical in Auto-translator

	+	-	*	/	%	<	>	=	!	&	|	.	:	:	_	(	)	#	@	sp	D	L	err	bk	ty
1	2	3	4	5	6	7	10	13	16	19	21	23	24	25	37	26	27	28	29	1	30	34	36		
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	8
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	9
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	10
5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	11
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	12
7	8	8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	0	-1
8	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15
9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	17
10	11	11	11	11	11	11	11	12	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	0	-1
11	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	16
12	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	18
13	14	14	14	14	14	14	14	15	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	0	-1
14	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	26
15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	13
16	37	37	37	37	37	37	37	18	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	0	-1
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
18	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	14
19	17	17	17	17	17	17	17	17	17	20	17	17	17	17	17	17	17	17	17	17	17	17	17	0	-1
20	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	19
21	17	17	17	17	17	17	17	17	17	17	22	17	17	17	17	17	17	17	17	17	17	17	17	0	-1
22	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	20
23	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	21
24	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	22
25	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	23
26	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	24
27	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	25
28	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	27
29	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	28
30	31	31	31	31	31	31	31	31	31	31	31	32	31	31	31	31	31	31	31	31	30	31	31	0	-1
31	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	31
32	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	32	33	33	0	-1
33	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32
34	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	34	35	0	-1
35	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	30
36	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	-1
37	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	29

Appendix III: Table used for Syntactic in Auto-translator

	+	-	||	*	/	%	&&	;	(	)	#	@	==	!=	<	>	<=	>=	if	then	else	where	end	Id	Int	Real	L	R	!	$
TranRule																								1						p
RR																								2						3
Rule																								4						p
E	6	6	p					p	5	p	8	p	p	p	p	p	p	p	7	p	p	p		5	5	5	5	5	5	
E1	9	9	9					13		13		11	10	10	10	10	10	10		13	13	12	13	13	13	13				
term	p	p	p	p	p	p	p	p	14	p		p	p	p	p	p	p	p		p	p	p		14	14	14	14	14	14	
term1	16	16	16	15	15	15	15	16	16	16		16	16	16	16	16	16	16		16	16	16	16	16	16	16				
QR																							18	17						
Q																							p	19						
factor	p	p	p	p	p	p	p	p	23	p		p	p	p	p	p	p	p		p	p	P		20	21	22	24	26	25	
sign	27	28							p															p	P	P	p	P	p	
addOp	29	30	31						p															p	P	P	P	p	p	
multOp				32	33	34	35		p															p	P	P	p	P	p	
relOp	p	P							p		p		40	41	36	37	38	39	p					p	P	p	p	p	p	

In the table: **P** stands error; program will pop the element from the stack and read the next token.
Blank stands error; program will read the next token directly.
Number stands for doing the corresponding rule.

12. Appendix IV: First/Fellow sets in Auto-translator

<table>
<tr><th>Name</th><th>First sets</th><th>Follow sets</th></tr>
<tr><td><TranRule></td><td>‘Id’</td><td>$</td></tr>
<tr><td><RR></td><td>‘Id’ , ε</td><td>$, ‘^’</td></tr>
<tr><td><Rule></td><td>‘Id’</td><td>$</td></tr>
<tr><td><E></td><td>‘Id’, ‘IntType’, ‘RealType’, ‘(’, ‘+’, ‘-’, ‘if’, ‘#’, ‘L’, ‘D’, ‘R’</td><td>‘^’, ‘then’, ‘else’, ‘;’, ‘)’, ‘||’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘= =’, ‘!=’, ‘@’, ‘where’,</td></tr>
<tr><td><E1></td><td>‘+’, ‘-’, ‘||’, ‘<’, ‘>’, ‘<=’, ‘@’, ‘>=’, ‘= =’, ‘!=’, ‘where’, ε</td><td>‘^’, ‘then’, ‘else’, ‘;’, ‘)’, ‘Id’, ‘IntType’, ‘RealType’, ‘end’</td></tr>
<tr><td><term></td><td>‘Id’, ‘IntType’, ‘RealType’, ‘(’, ‘L’, ‘D’, ‘R’</td><td>‘*’, ‘/’, ‘%’, ‘&&’, ‘+’, ‘-’, ‘||’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘= =’, ‘!=’, ‘@’, ‘where’, ‘^’, ‘then’, ‘else’, ‘;’, ‘)’</td></tr>
<tr><td><term1></td><td>‘*’, ‘/’, ‘%’, ‘&&’</td><td>‘+’, ‘-’, ‘||’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘= =’, ‘)’, ‘!=’, ‘@’, ‘^’, ‘where’, ‘then’, ‘else’, ‘;’, ‘Id’, ‘IntType’, ‘RealType’</td></tr>
<tr><td><QR></td><td>‘Id’, ε</td><td>‘end’</td></tr>
<tr><td><Q></td><td>‘Id’</td><td>‘end’</td></tr>
<tr><td><factor></td><td>‘Id’, ‘IntType’, ‘RealType’, ‘(’, ‘L’, ‘D’, ‘R’</td><td>‘*’, ‘/’, ‘%’, ‘&&’, ‘+’, ‘-’, ‘||’, ‘<’, ‘>’, ‘<=’, ‘>=’, ‘= =’, ‘!=’, ‘@’, ‘^’, ‘where’, ‘then’, ‘else’, ‘;’, ‘)’</td></tr>
<tr><td><sign></td><td>‘+’, ‘-’</td><td>‘Id’, ‘IntType’, ‘RealType’, ‘(’, ‘L’, ‘D’, ‘R’</td></tr>
<tr><td><addOp></td><td>‘+’, ‘-’, ‘||’</td><td>‘Id’, ‘IntType’, ‘RealType’, ‘(’ ,‘L’, ‘D’, ‘R’</td></tr>
<tr><td><multOp></td><td>‘*’, ‘/’, ‘%’, ‘&&’</td><td>‘Id’, ‘IntType’, ‘RealType’, ‘(’, ‘L’, ‘D’, ‘R’</td></tr>
<tr><td><relOp></td><td>‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’</td><td>‘Id’, ‘IntType’, ‘RealType’, ‘(’, ‘+’, ‘-’, ‘if’, ‘#’, ‘L’, ‘D’, ‘R’</td></tr>
</table>