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

Parsing and Abstract Syntax Tree Generation in the GIPSY System

Chun Lei Ren

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Science at

Concordia University

Montreal, Quebec, Canada

September 2002

Abstract

The GIPSY system is the implementation of a new programming paradigm: Intensional Programming. Parsing and abstract syntax tree generation of the Lucid language are the basis of the system. The objective of the thesis is to develop GIPL and SIPL Indexical Lucid parsers and to generate abstract syntax trees for the programs of both languages. We chose JavaCC/JJTree from tens of compiler construction tools. By modifying the original Lucid lexical and syntactical specifications, and customizing the JavaCC/JJTree generated files, we implemented target AST-building-abled SIPL and GIPL Lucid parsers. As well, we implemented an abstract syntax tree translator for the Indexical Lucid SIPL. Finally, we successfully integrated GIPL and Indexical Lucid SIPL parsers, their translator, and the simulating GIPSY stub.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, my mentor – Dr. Joey Paquet, for accepting me as his student, introducing me into this field and providing me financial support, kindly provided to him by NSERC. I am also thankful to Dr. Peter Grogono. Without his contribution, this thesis would not have been possible. Thanks also to all GIPSY project teammates for their generous help and efforts. Specially, I would like to thank my dearest parents, my sisters and my brothers. Their greatest love always my invincible backup.

Table of Contents

List of Tables

List of Figures

List of Code Excerpts

1. Introduction

This thesis is mainly about the design and implementation of a paser and abstract syntax tree generator for the GIPSY system. The GIPSY (General Intensional Programming System) is a very ambitious project involving a new programming paradigm, namely Intensional Programming. The project is directed by Dr Joey Paquet and Dr. Peter Grogono of Concordia University, Montreal, Canada. There are currently three PhD students and five Msc students working on the various parts of the project.

1.1 Background

GIPSY is the implementation of a new programming paradigm: Intensional Programming. The basic idea behind this paradigm is the programming and execution of expressions in an inherent context of evaluation. In the GIPL (Generic Intensional Programming Language), expressions are explicitly multidimensional, and their evaluation varies in this context space created by the explicit dimensionality of the programs. However, unlike its unidimensional counterparts, such as time logic or other modal logics, the context of evaluation is inherent in GIPL programs. That allows writing programs or functions that are dimensionally abstract, e.g. functions that act upon data elements of any number of dimensions. [1]

Conventional programming languages (e.g. Fortran, C, C++, Java, etc.) can be used to solve problems of multidimensional nature, such as particle in-cell simulation of plasma using differential equations. However, this most often leads to code that has no direct relationship with the simulated equations, that is bound to the simulation in a fixed dimensional context, and that makes a ubiquitous reference to the dimension names in the code. It has been proven that intensional programming can be used to build programs to solve such

problems that are much more related to the original equations, and that are dimensionally abstract, and thus much more flexible. [1]

GIPL programs are also inherently parallel, thus enabling massive comptations to be undertaken on distributed or parallel architectures. In this execution mode, GIPL programs can even include sequential functions, written in any mainstream programming language, as computation units executed in parallel, where the GIPL itself becomes a skeleton language describing how the parallelism is to be exploited. [1]

The GIPL is the latest evolution of Lucid, the first intensional programming language. The GIPSY system is itself an evolution of GLU (Granular Lucid), the first intensional programming system to enable parallel execution of Lucid programs using the notion of sequential threads. GLU enabled the compilation of Indexical Lucid programs, together with the use of sequential threads written either in C or Fortran. It has proven to be usable and a highly efficient solution for the parallelization of sequential programs. However, the latest version of GLU, developed at the Stanford Research Institute (SRI), could not cope with the latest evolution of Lucid. [1]

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]

Even more troublesome is the totally alien nature of the GIPL language. Programmers have a very heavy 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 parallelize programs through code reuse. [1]

1.2 Architecture of the GIPSY System

In order to reach the flexibility and adaptability goals of the system, defining a clear and adaptive architecture is a key aspect of this project. The system has been decomposed in three main subsystems, as depicted in Figure 1-1.

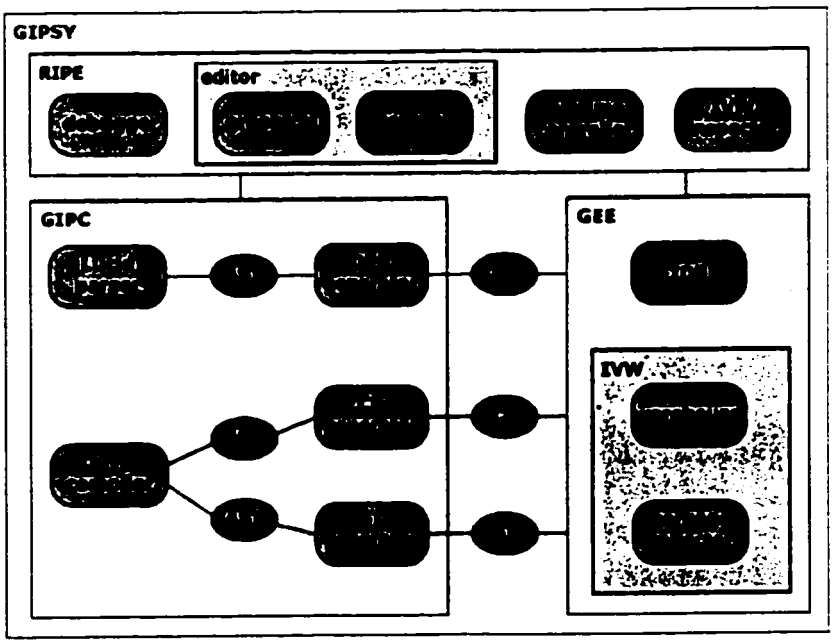


Figure 1-1 : High-level architecture of the GIPSY system

(**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)

1.2.1 General Intensional Programming Compiler (GIPC)

GYPSY programs are compiled in a two-stage process. First, the intensional (GIPL) part of the GYPSY program is parsed, and then translated in Java data structures, then the resulting Java program compiled in the standard way.

The source code consists of two parts: the GIPL part that defines the intensional data dependencies between variables and the sequential part that defines the granular sequential computation units (written in Java). The GIPL part is compiled into demand propagation resources (DPR) describing the dependencies between each variable involved in the GIPL part. These resources are to be used by the general eduction engine (GEE).

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 structures definitions written in the GIPSY program, yielding 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 second 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, will be flattened using a well-know efficient technique. [1]

1.2.2 General Eduction Engine (GEE)

The 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. The

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. Every computed value is placed in the warehouse, and 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) is 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, that in turn raise 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 resolves to a value and is put in the warehouse for further use.

For functional demands (i.e. demands associated with the execution of a sequential thread), the demands are sent to the demand dispatcher (IDD) that takes care of sending the demand to one of the workers or to resolve 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 demand dispatcher receives some information about the liveness and efficiency of all workers from the demand monitor (IDM), to help it take better decisions in dispatching functional demands.

The demand monitor, after some functional demands are sent to workers, starts to gather various informations about each worker:

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

These informations are accessed by the IDD to make better decisions about the load balancing of the workers, and thus achieving better overall run-time efficiency.

1.2.3 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 part of the GIPSY programs. The user can interact with the RIPE at run-time in the following ways

- dynamically inspect the IVW;
- change the input/output channels of the program;
- recompile sequential threads;
- change the communication protocol;
- change parts of the 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 [7]. 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 table 1-1.

```
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 1-2 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.

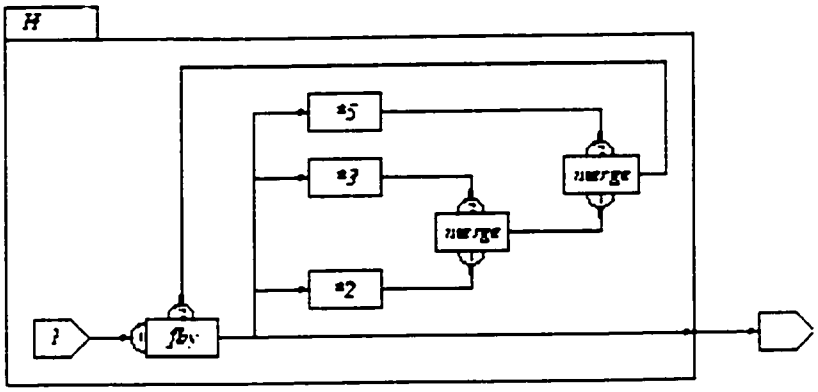


Figure 1-2 : Dataflow graph for the Hamming problem

Using this visual technique, the RIPE will even 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 solution is 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. [1]

1.3 Contributions

This thesis is a part of the development of the compiler (GIPC) component of the GIPSY system, as depicted in more details in Figure 1-3.

More specifically, this thesis aims at:

- Developing a parser for the GIPL (Generic Intensional Programming Language). GIPL is the common and basic set of all SIPL languages. So, the GIPL parser is the starting-point of later works.
- Developing an abstract syntax tree generator for the GIPL.

- Developing a parser for the Indexical Lucid SIPL, which is a superset of the GIPL. This language has been chosen as the first one to be implemented because it has proven to be a strong basis for the development of other languages of intensional nature.
- Developing an abstract syntax tree generator for the same language.
- Developing a translator, which translates abstract syntax tree of Indexical Lucid SIPL into GIPL abstract syntax tree. Since the GIPSY system will treat GIPL AST only, any SIPL AST constructed by related generator has to be translated into GIPL AST with corresponding translator. Hence, this Indexical Lucid SIPL AST translator is also acting the template for the other new-defined SIPL AST translators.
- Implementing a flexible and efficient way of developing the different parsers and abstract syntax tree generators eventually involved in the system. Eventually, other languages of intensional nature, that we call Specific Intensional Programming Languages (SIPLs), are going to be made available in the system. For their integration to be reasonably easy, we have to clearly define how parsers are to be designed and implemented.
- Showing how the classes generated are to be integrated in the whole system.

1.4 Structure of the Thesis

Chapter 2 presents a survey of the compiler construction tools available on the market, and shows how we made a choice on a particular tool for the implementation of the parser and abstract syntax tree generator, namely JavaCC and JJTree. Chapter 3 presents the lexical and syntactical specifications of the language parsed, as well as the format of the JavaCC grammar file format, into which our specifications have to be translated. Chapter 4 presents how to generate a lexical analyzer using JavaCC. Chapter 5 presents how to generate a syntactical analyzer using JavaCC.

Chapter 6 presents how to generate an abstract syntax tree using JJTree. Chapter 7 presents how to translate SIPL tree to GIPL tree. Chapter 8 presents how to merge the GIPL parser, SIPL parser and the translator into the GIPSY system. Chapter 9 presents the result of evaluation. Chapter 10 presents the future work.

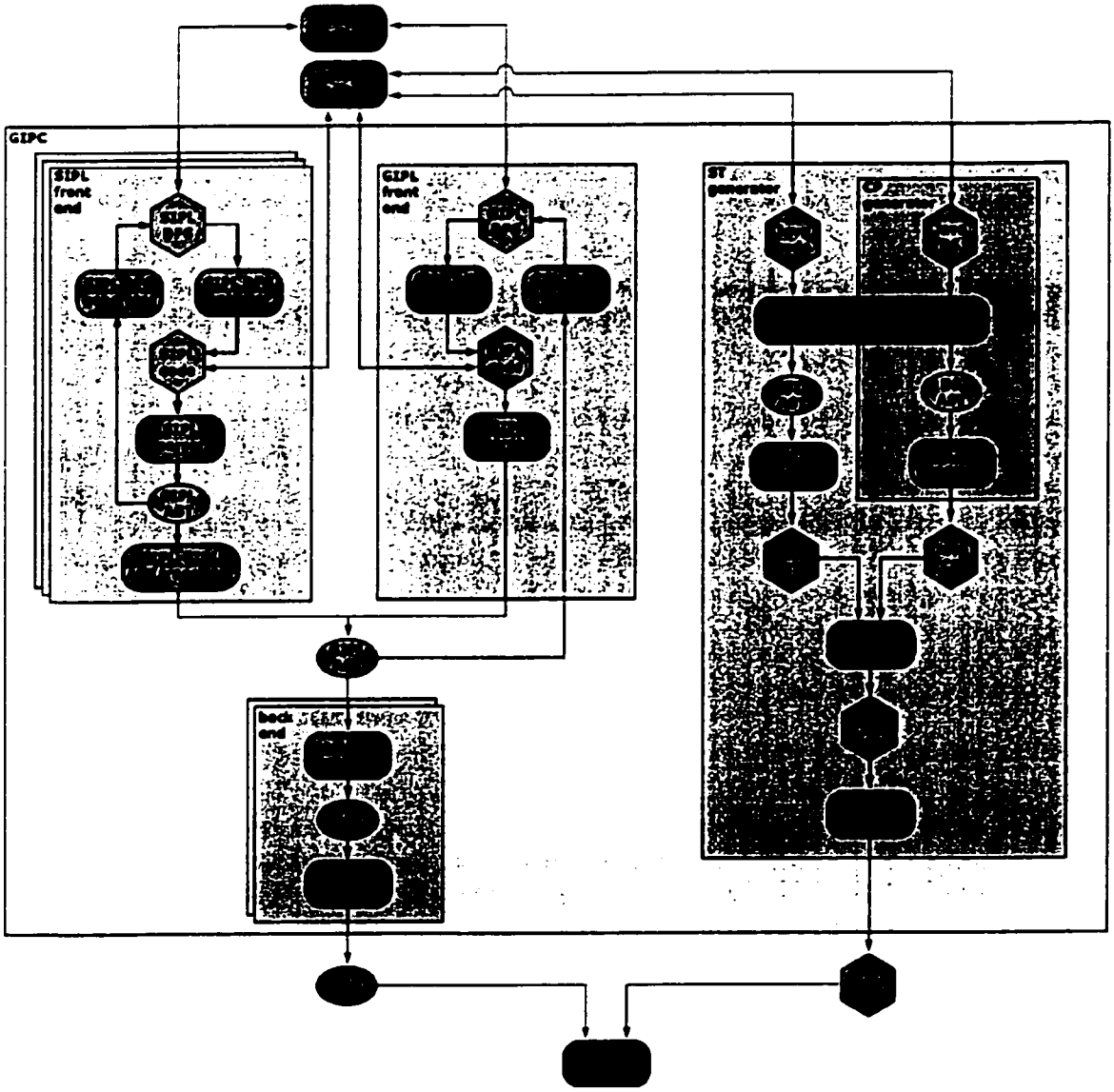


Figure 1-3 : Architecture of the compiler module (GIPC)

2. Compiler Construction Tools

2.1 Background

Our first task is to find an appropriate compiler construction tool to facilitate the development of a parser for the Lucid programming language. Using compiler construction tool can maximize the efficiency of compiler development and minimize parser faults. The products of compiler construction tools to be selected will compose a sub-system of GIPSY system, which is supposed to run cross-platform and has defined Java as its host language, and have to work well when integrated into the system.

The GIPSY system is an integrated development environment for Lucid programming language. GIPSY, as well as Lucid itself, is always evolving. The Lucid grammar has been changing since it was invented in 1974. So, introducing compiler construction tools into the development of GIPSY system is necessary. Generally, compiler construction tools group into the following categories:

- Compiler Construction Kits (Integrated systems that cover the full or part of spectrum of compiler construction.)
- Lexer and/or Parser Generators
- Trees and Transformation (i.e. MEMPHIS supports definition and processing of abstract syntax trees)
- Attribute Grammar Systems (tools that supports definition and processing of attribute grammar)
- Backend Generators (Code generator-generator)
- Program Analysis and Optimization (i.e. BANE is a toolkit for constructing program analyses such as dataflow and type inference systems.)

- Environment Generators (i.e. The Synthesizer Generator is a tool for creating language-sensitive editing environments and interfaces)
- Tools, Frameworks, Infrastructures (offer applications and technologies that facilitate the development of compiler or even more complicated software system)

The research work for the thesis involves Lexer and Parser generator, as well as some compiler construction related auxiliary tools, such as error handling tools and parse-tree building tools, etc. Later in this chapter, we are going to discuss the requirements of which my work concerns to the compiler construction tools, analyze currently available compiler construction tools, and finally choose the most suitable tool(s). [7]

In short, my research work is to implement parsers for both basic Lucid programming language (GIPL) and for the Indexical Lucid programming SIPL, build the abstract syntax trees for them, and finally, translate the AST of Indexical Lucid into GIPL AST. The parsers are plug-in modules of the GIPSY system and must be implemented in Java. Since the GIPSY system is intended to operate across platforms, the target parser must have this capability too. The abstract syntax tree concerns intermediate code generation, which is due to other participant of the project. Issues concerning abstract syntax tree will be discussed later in other chapter.

2.2 Comparative Study of Compiler Construction Tools

Compiler construction tools will definitely improve the quality of our work. When we refer to parser, sometimes, this parser also implicitly covers the scanner. Because of the special relationship between parser and scanner and different applications, there are, now, many different kinds of generators available. While they all fall into the following three categories:

- Scanner generator,
- Parser generator,
- And integrated scanner/parser generator.

Generally, compared to the integrated scanner/parser generator, combining scanner generator and parser generator to create a parser involves more files, and the process is much more complicated. For example, the following diagram illustrates the process of building a parser with Lex and Yacc.

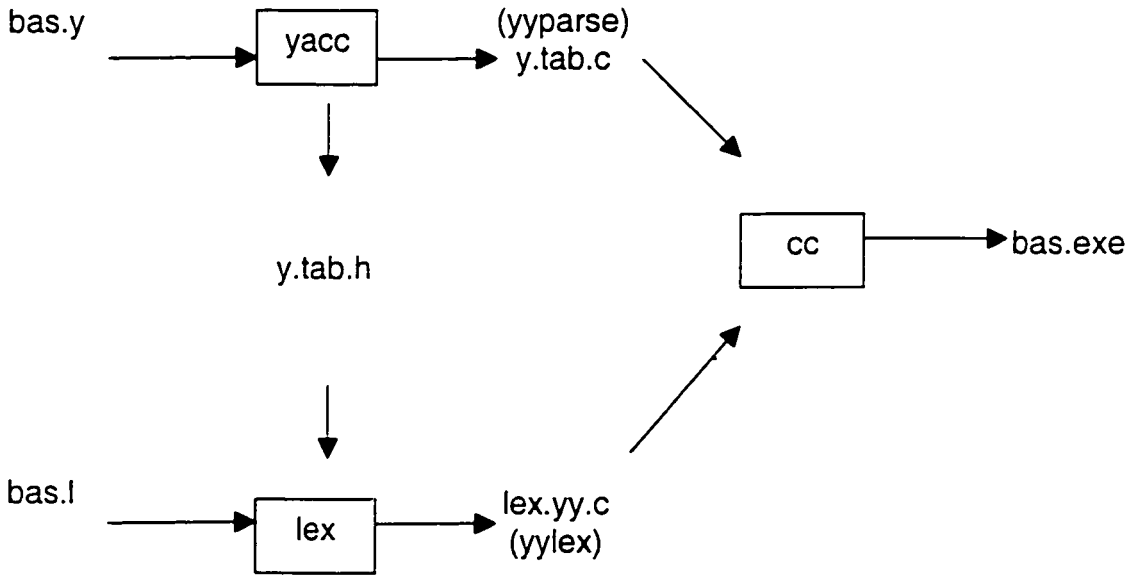


Figure 2-1 : Constructing a parser with lex and yacc

Yacc reads the grammar descriptions in bas.y and generates a parser, function yyparse, in file y.tab.c. Included in file bas.y are token declarations. These are converted to constant definitions by yacc and placed in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, function yylex, in file lex.yy.c. Finally, the lexer and parser are compiled and linked together to form the executable, bas.exe. [7]

Commonly, integrated scanner/parser generators, such as JavaCC, feature lexical and grammar specifications in one file. The lexical specifications such as regular expressions, strings, etc. and the grammar specifications (the BNF) are both written together in the same file. It makes grammars easier to read (since it is possible to use regular expressions within the grammar specification) and also easier to maintain. So, we prefer integrated scanner/parser generator as our compiler construction tool.

Scanner or parser generators are written in different programming language, such as Yacc is written in C, YOOCC is written entirely in Eiffel. While target parser generated in what programming language is what we really concern about. Fortunately, there are various parser generators, which can generate parsers in almost all popular programming languages. For example, AFLEX & AYACC are written in Ada and generate Ada output, JavaCC generate parser in Java code, ANTLR can even generate parser both in Java and C++. [7]

Why is target parser code so important? First of all, the GIPSY system is a team project. The Lucid parsers and AST generators are just a part of it. The generated code type of Lucid parser and AST affect the other parts of the system and the other participants. Secondly, since the GIPSY system is intended to run on different platforms, the parser should be implemented in a portable programming language. Finally, GIPSY, as well as Lucid, is always evolving. Only object-oriented programming language is capable to satisfy these features. The Java programming language seems to be the best candidate that can satisfy those three criteria.

The generated parser's source code needs to be in a portable programming language. Likewise, lexer/parser construction tools also need to have different installed versions for different platform. So far, we require that the candidate

tools can operate at least on both Unix/Linux, Microsoft Windows 2000, and MacOS X (based on FreeBSD Unix) operating systems.

Another important issue about the parser generator is the supporting grammar. Supporting grammar is the way by which the parser generator recognizes and treats the input grammar. It determines how much effort we have to do on the raw Lucid grammar. Due to different effort and emphasis, parser construction tools accept grammars in different forms. Some tools, which support LALR grammar, are much more powerful tools. In terms of this kind of tools, we almost don't need to change the grammar and just express it in EBNF. While for some tools that support LL(1) only, we have to eliminate the left recursions and ambiguities existing in the original Lucid grammar before applying the tools onto the grammar.

It is hard to get a perfect parser generator, especially a free one. Most LALR grammar-supporting tools can run only under Unix system, or generate parser in non-object-oriented code, or have other defects. While some LL grammar-supporting tools always bear the complementary functionality (or capabilities). For example, Jay (a Yacc for Java) and Jax (lexer generator) generate parser in Java code and supporting grammar is LR(1), but they can operate only under a Unix platform; in addition, they are not integrated. On the contrary, ANTLR, an LL(K)-parser generator, generates parsers both in Java and C++ code, and it can run on multi-platform. So we have to do more trade-offs between supporting grammar and other capabilities.

Reliability is an important feature for lexer/parser generators. How do we evaluate the reliability? Generally we refer to the following factors:

- **Release date.** Generally, the earlier the tool was released, the more reliable it is. Time is the best measurement for quality. In addition,

exposing and recovering the defects of the tool take time. For example, Lex/Yacc was published in 1975. They are the first and most reliable lexer/parser generator so far. [7]

- **Popularity.** If a software tool is very good quality, more and more users will choose it through the comments or recommendations from the previous users. So is the lexer/parser generator.
- **Developer, development organization, or sponsor of the tools.** Famous software development organization always has stronger technical and financial supports. Consequently their software products are expected to be of good quality.
- **Testimonial.** Some users intend to write papers and share their experience and lessons from using a certain software product. These papers are so valuable that they can help us to judge the quality and other features of software products.

From financial aspect, lexer/parser generators are categorized into commercial products and free products. Free products can be downloaded from related websites. Typically, commercial parser generator tools are superior to the free ones in functionality. While compiler construction related techniques are quite mature so far and many free tools can satisfy our requirements to this case. For example, JavaCC is a free lexer/parser generator. It has different install versions for different platforms and generates parser in Java code. So we will focus on the free tools and choose an appropriate one from them.

My research work involves not only generating Lucid parsers, also creating abstract syntax trees. Some compiler construction tools themselves, such as JavaCC, provide other standard capabilities related to parser generation such as tree building, actions, debugging, etc. Those auxiliary capabilities will definitely facilitate my research work. So when we evaluate a lexer/parser

generator, we also need to check if there are handy auxiliary capabilities attached to that tool.

The candidate lexer/parser generator must be easy to learn and easy to use. First of all lexer/parser generators are just tools. It is not wise to spend much more effort on learning a tool rather than on doing the project itself. Furthermore, the GIPSY system is always evolving. The successors also have to learn how to use the tools. We wish that they would benefit from our decision and could use efficient and easily learned tools. Secondly, using those tools to generate a parser is far from the ending of my work. In order to create a qualified parser in terms of lexer/parser generator, besides preparing the input file for tools we also have to modify the output code. So a lexer/parser generator, featuring easy to learn and easy to use, will promote the efficiency and quality of our work.

Software efficiency is an important issue in most applications, so is it in our case. We have different requirements to lexer/parser generator and generated parser. Since the lexer/parser generator only works off-line, we don't need it to be efficient. As to the generated parser, efficiency has to be taken into account because it generally works during run-time. However, the GIPSY system is a research project in laboratory and the research work just sets off, efficiency of generated parser is not so critical. We can accept the generated lexer/parser generator as long as their performance is not too low.

There are over hundred lexer/parser generators available nowadays. As to our research work, we need the most suitable one rather than the best one. The analysis above actually is the requirements (of our research work) to the lexer/parser generator. The following criteria concluded from the analysis above will be used to assist in the choice of the lexer/parser generator for our research work.

- Integrated scanner/parser generator
- Free download.
- Target parser code in object-oriented programming language.
- More powerful capability in treating the input grammar.
- Operate on as many platforms as possible, at least on Microsoft Windows 2000, Unix/Linux and MacOS X (based on FreeBSD Unix).
- Good quality and reliability.
- Auxiliary capability, including tree building, debugging, and so on.
- Easy to learn and use.
- Complete and well-organized document.

2.3 Result

After analyzing nearly 30 compiler construction tools, we decided to use JavaCC as Lucid lexer/parser generator. Appendix I lists those compiler construction tools and their information. Compared to the other tools, JavaCC most satisfies the criteria listed above. The following is JavaCC features corresponding to those criteria:

- First, JavaCC is an **integrated lexer/parser generator**. Lexical specifications such as regular expressions, strings, etc. and the grammar specifications (the BNF) are both written together in the same file. It makes grammars easier to read and also easier to maintain.
- JavaCC can be **freely** downloaded from the website of WebGain. The URL is http://www.webgain.com/download/index.html.
- JavaCC **produces parsers in Java** and it is completely written in Java. JavaCC runs on all Java compliant platforms Version 1.1 or later. It has been used on countless different machines with no special

porting effort - a testimonial to the "Write Once, Run Everywhere" aspect of Java.

- JavaCC generates **top-down** (LL(K)) (recursive descent) parsers as opposed to bottom-up parsers generated by YACC like tools. This allows the use of more general grammars (although left-recursion is disallowed). Top-down parsers have many other advantages, such as being easier to debug, the ability to parse to any non-terminal in the grammar, and the ability to pass values (attributes) both up and down the parse tree during parsing.

- JavaCC **runs on all Java compliant platforms** Version 1.1 or later.

- The latest release of JavaCC is Version 2.0. This version was released jointly by Metamata, Inc., now as wholly owned subsidiary of WebGain, Inc., and Sun Microsystems. JavaCC has a **large user community.** JavaCC is by far the most popular parser generator used with Java applications. They have had tens of thousands of downloads and estimate serious users in the thousands. Their mailing list and newsgroups together have a few thousand participants.

- JavaCC comes with JJTree, an extremely powerful **tree building preprocessor**. JavaCC has extensive debugging capabilities. Using options DEBUG_PARSER, DEBUG_LOOKAHEAD, and DEBUG_TOKEN_MANAGER, one can get in-depth analysis of the parsing and the token processing steps. Still JavaCC has very good error reporting ability. JavaCC error reporting is among the best in parser generators. JavaCC generated parsers are able to clearly point out the location of parse errors with complete diagnostic information.

- JavaCC is relatively **easy-to-learn and easy-to-use**. Its documentation is complete and well-organized. In addition, the

JavaCC release includes a wide range of examples including Java and HTML grammars. The examples along with their documentation are a great way to get acquainted with JavaCC.[3]

Besides the features above, JavaCC is also:

- **Extremely customizable**: JavaCC offers many different options to customize its behavior and the behavior of the generated parsers. Examples of such options are the kinds of Unicode processing to perform on the input stream, the number of tokens of ambiguity checking to perform, etc.

- **Document Generation**: JavaCC includes a tool called JJDoc that converts grammar files to documentation files (optionally in html).

- **Internationalized:** The lexical analyzer of JavaCC can handle full Unicode input, and lexical specifications may also include any Unicode character. This facilitates descriptions of language elements such as Java identifiers that allow certain Unicode characters (that are not ASCII), but not others.

- **Syntactic and semantic lookahead specifications**: By default, JavaCC generates an LL(1) parser. However, there may be portions of the grammar that are not LL(1). JavaCC offers the capabilities of syntactic and semantic lookahead to resolve shift-shift ambiguities locally at these points. In other words, the parser is LL(k) only at such points, but remains LL(1) everywhere else for better performance. Shift-reduce and reduce-reduce conflicts are not an issue for top-down parsers.

- **Permits extended BNF specifications**: JavaCC allows extended BNF specifications - such as (A)*, (A)+, etc. - within the lexical and the grammar specifications. Extended BNF relieves the need for left-

recursion to some extend. In fact, extended BNF is often easier to read as in A ::= y(x)* versus A ::= Ax|y.

- **Lexical states and lexical actions**: JavaCC offers "lex" like lexical state and lexical action capabilities. Specific aspects in JavaCC that improve over other tools are the first class status it offers concepts such as TOKEN, MORE, SKIP, state changes, etc. This allows cleaner specifications as well as better error and warning messages from JavaCC.

- **Case-insensitive lexical analysis**: Lexical specifications can define tokens not to be case sensitive either at the global level for the entire lexical specification, or on an individual lexical specification basis.

- **Special tokens**: Tokens that are defined as special tokens in the lexical specification are ignored during parsing, but these tokens are available for processing by the tools. A useful application of this is in the processing of comments. [3]

3. GIPL Grammatical Specifications

3.1 Syntactical Specifications of the GIPL

3.1.1 Abstract Syntax for Intensional Programming

Lucid syntax can be in 2 modes, abstract syntax and concrete syntax. Abstract syntax of GIPL show in table 3-1, generally, is used for theoretical analysis. [1]

E	::=	id
	|	<E>(<E>$_1$,…,<E>$_n$)
	|	if <E> then <E>' else <E>''
	|	#<E>
	|	<E>@<E>' <E>''
	|	<E> where <Q>
Q	::=	dimension id
	|	id=<E>
	|	id(id$_1$,…,id$_n$)=<E>
	|	<Q><Q>

Table 3-1 : Abstract syntax of intensional programming

3.1.2 Concrete Syntax for Intensional Programming

Concrete syntax is the one that is applied to practice and that really has to be taken into account in this project. Table 3-2 shows the concrete syntax of GIPL.

E	::=	id
	|	<E>[<E>$_1$,…,<E>$_m$](<E>$_{m+1}$,…,<E>$_n$)
	|	if <E> then <E>' else <E>'' fi
	|	#.<E>
	|	<E>@.<E>' <E>''
	|	<E> where <Q> end
Q	::=	dimension id$_1$,…,id$_n$;
	|	id = <E>;
	|	id [id$_1$,…,id$_m$](id$_{m+1}$,…,id$_n$)=<E>;
	|	<Q><Q>

Table 3-2 : Concrete syntax for the GIPL

The non-terminals E and Q respectively refer to expressions and definitions. There are a number of operations that should be found in almost all Lucid variants. The standard predefined intensional operators can be found in table 3-3.

3.1.3 Concrete Syntax for Indexical Lucid

According to the concrete Indexical Lucid syntax, we drew up the concrete Indexical Lucid grammar for the GIPSY system, as shown in the following Table:

```
            E  ::=  <term>
               |    <sign><term>
               |    <E><addOp><term>
               |    <E><relOp><E>
               |    if <E> then <E> else <E> fi
               |    #.<E>
               |    <E>@.<E><E>
               |    <E>where<Q>end
         term  ::=  <term><multOp><factor>
               |    <factor>
       factor  ::=  id
               |    const
               |    <E>[<E>,…,<E>](<E>,…,<E>)
               |    (<E>)
               |    not(<E>)
               |    <indexicalExp>
            Q  ::=  dimension id,…,id;
               |    id=<E>;
               |    id[id,…,id](id,…,id)=<E>;
               |    <Q><Q>
indexicalExp  ::=  <unOp>[<E>,…,<E>](<E>)
               |    (<E>)<binOp>[<E>,…,<E>](<E>)
         unOp  ::=  first | next | prev | iseod
        binOp  ::=  fby | wvr | asa | upon
         sign  ::=  + | -
        addOp  ::=  + | - | or
       multOp  ::=  * | / | % | and
        relOp  ::=  < | > | <= | >= | == | !=
```

Table 3-3 : Concrete syntax for the Indexical Lucid SIPL

In table 3-3, production *indexicalExp* (indexical expression) along with operations *unOp* (unary operation) and *binOp* (binary operation) make the GIPL Lucid up into a SIPL -- Indexical Lucid.

Note that this original concrete Indexical Lucid grammar is going to be transformed in different development stages due to the use of Javacc and JJTree. For example, to eliminate the left recursions, while the original syntax will remain.

3.2 The JavaCC Grammar Specification Language

The objective of this project is, according to the original Indexical Lucid grammar, to generate a Lucid parser, and embed abstract syntax tree building function into the parser.

Java Compiler Compiler (JavaCC) is a parser generator for use with Java applications. JavaCC is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building via a tool called JJTree included with JavaCC. So, JavaCC and JJTree can be utilized to facilitate the development of our target parser. [3]

The common procedure of developing a parser that has tree building capability, for example, the target parser of this project, happens to match the learning procedure of JavaCC and JJTree. That is to develop the lexical analyzer, implement the syntactical analyzer and finally, add the abstract syntax tree building function. This chapter and following chapters will, based on the procedure, introduce JavaCC and JJTree, how to make up the original Lucid lexeme and syntax to JavaCC/JJTree format, and how to generate the target parser.

Since JavaCC is an integrated parser/scanner construction tools, the lexical specifications and syntactical specifications of Lucid have to be written into the same file in JavaCC format. JavaCC takes this file as input, then generates a lexical analyzer and syntactical analyzer in the same time. As to tree building, JJTree is a pre-processor, which is based on the JavaCC grammar file. So, we will introduce JJTree and add tree-building function after we successfully get the parser.

The rest of this chapter will depict the outline of JavaCC grammar file. Then the later chapters will give detailed descriptions on how to implement the target parser with JavaCC and JJTree.

3.2.1 The JavaCC Grammar File

The grammar file contains the lexical and syntactical specifications that are written in a particular format, which is determined by the parser generator that will be used. Parser/scanner generator takes the grammar file, analyzes the specifications and generates parser. This is illustrated by figure 3-1.

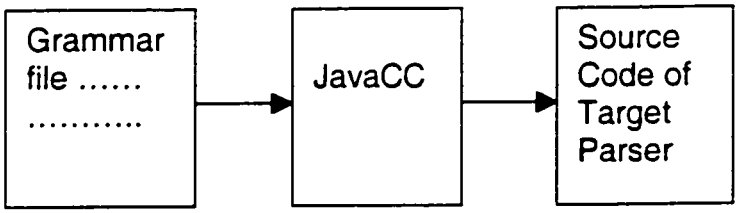


Figure 3-1 : Grammar file and parser generator

3.2.2 Basic JavaCC Grammar File Conventions

To generate Lucid parser with JavaCC, Lucid grammar has to be written in JavaCC grammar file format. First of all, there are some basic rules for grammar file:

Keywords

The following are the reserved words in the JavaCC grammar file:

EOF, IGNORE_CASE, JAVACODE, LOOKAHEAD, MORE, options, PARSER_BEGIN, PARSER_END, SKIP, SPECIAL_TOKEN, TOKEN, TOKEN_MGR_DECLS [3]

Token Convention

Tokens in the grammar files follow the same conventions as for Java. Hence identifiers, strings, characters, etc. used in the grammars are the same as Java identifiers, Java strings, Java characters, etc. [3]

White Spaces and Comments

White space in the grammar files also follows the same conventions as for Java. This includes the syntax for comments. Most comments present in the grammar files are generated into the generated parser/lexical analyzer. [3]

Unicode

Grammar files are preprocessed for Unicode escapes just as Java files are (i.e., occurrences of strings such as **\uxxxx** - where xxxx is a hex value - are converted the corresponding Unicode character before lexical analysis). [3]

Grammar File Structure

Table 3-4 is the highly abstracted JavaCC grammar file structure. The sequence of each part must be kept. [3]

```
Javacc_options
"PARSER_BEGIN" "(" <IDENTIFIER> ")"
java_compilation_unit
"PARSER_END" " ("<IDENTIFIER> ")"
(production)*
```

Table 3-4 : Structure of JavaCC grammar file

3.2.3 JavaCC Options

The grammar file starts with a list of options, which is optional also. We can put the options right here, alternatively, we can put them on JavaCC command line. If the option is set from the command line, that takes precedence.

The options if present, starts with the reserved word "options" followed by a list of one or more option bindings within braces. Each option binding specifies the setting of one option. The same option may not be set multiple times. A sample JavaCC_options would be like: [3]

```
options {
     LOOKAHEAD = 2;
     COMMON_TOKEN_ACTION = true;
     FORCE_LA_CHECK = true; }
```

Table 3-5 : A sample of `javacc_option`

3.2.4 The Java Compilation Unit

Java compilation unit enclosed between **`PARSER_BEGIN(name)`** and **`PARSER_END(name)`**. The name that follows **`PARSER_BEGIN`** and **`PARSER_END`** must be the same and this identifies the name of the generated parser. For example, if name is "Lucid", then the target parse is also generated with the name “Lucid”.

Java compilation unit may be any arbitrary Java compilation unit so long as it contains a class declaration whose name is the same as the name of the generated parser. JavaCC does not perform detailed checks on the compilation unit, so it is possible for a grammar file to pass through JavaCC and generate Java files that produce errors when they are compiled. In general, this part of the grammar file looks like (table 3-6): [3]

```
PARSER_BEGIN(parser_name)
...  //  package names, import declarations, etc.
class parser_name . . . {
...
}  // should be the entry class or principle class
...
PARSER_END(parser_name)
```

Table 3-6 : Structure of Java compilation unit

In Java compilation unit, the class whose name is same as the parser name should be the entry class. Hence, the main method, important global variables, command line parameters processing, I/O initializing, and parsing initializing all have to be here. Table 3-7 shows a sample of Java compilation unit.

```
PARSER_BEGIN(Simple1)
public class Simple1 {
  public static void main(String args[])
     throws ParseException {
        Simple1 parser = new Simple1(System.in);
        parser.Sta();
     }
}
PARSER_END(Simple1)
```

Table 3-7 : A sample of Java compilation unit

This compilation unit declares a parser named Simple1, directs standard I/O as the input of target parser, and finally, initiates the parser (The starting state of the parser is **Sta**).

3.2.5 Productions

There are 4 kinds of productions: javacode production (**javacode_production**), BNF production (**bnf_production**), regular expression production (**regular_expr_production**) and token manager declaration (**token_manager_decls**). The sequence of all productions is arbitrary. Except that token manager declaration can occur at most once, all other productions' appearance can be arbitrary. [3]

javacode_production and **bnf_production** are used to define the grammar from which the parser is generated, **regular_expr_production** is used to define the tokens - the token manager is generated from this information, and **token_manager_decls** is used to introduce declarations that get inserted into the generated token manager. [3]

3.2.5.1 Javacode production

The syntax of javacode production is (table3-8):

```
Javacode_production ::= "JAVACODE"
      java_return_type
      java_identifier
      "(" java_parameter_list ")"
      java_block
```

Table 3-8 : Syntax of javacode production

The JAVACODE production is a way to write Java code for some productions instead of the usual EBNF expansion. This is useful when there is the need to recognize something that is not context-free or for whatever reason is very difficult to write a grammar for.

An example of the use of JAVACODE is shown below (table 3-9). In this example, the non-terminal "skip_to_matching_brace" consumes tokens in the input stream all the way upto a matching closing brace (the opening brace is assumed to have been just scanned): [3]

```
JAVACODE
   void skip_to_matching_brace() {
        Token tok;
        int nesting = 1;
        while (true) {
               tok = getToken(1);
               if (tok.kind == LBRACE) nesting++;
               if (tok.kind == RBRACE) {
                     nesting--;
                     if (nesting == 0) break;
               }
               tok = getNextToken();
        }
    }
```

Table 3-9 : A sample of a javacode_production

3.2.5.2 BNF production

Basically, bnf_production allows us to write our syntax productions in EBNF form. It also provides us more powerful way to customize our parser. The complete syntax of bnf_production is showed as table 3-10.

```
bnf_production ::=
      java_return_type
      java_identifier
```

```
"(" java_parameter_list ")" ":"
java_block
"{" expansion_choices "}"
```

(a) Syntax of BNF production

```
expansion_choices ::=
    (expansion_unit)*
    ( "|" (expansion_unit)* )*
```

(b) Syntax of expansion choices (expansion_choices)

```
expansion_unit ::=
      local_lookahead
    | java_block
    | "(" expansion_choices ")" [ "+" | "*" | "?" ]
    | "[" expansion_choices "]"
    | [ java_assignment_lhs "=" ]
regular_expression
    | [ java_assignment_lhs "=" ]
      java_identifier "(" java_expression_list ")"
```

(c) Syntax of expansion unit (expansion_unit)

```
local_lookahead ::=
    "LOOKAHEAD"
    "(" [ java_integer_literal ] [ "," ]
    [ expansion_choices ] [ "," ]
    [ "{" java_expression "}" ] ")"
```

(d) Syntax of local lookahead (local_lookahead)

Table 3-10 : Syntax of BNF production

The java_block in table 3-10-(c) is actually the place where we put syntactical action. The local_lookahead is setting a local LOOKAHEAD specification that affects only a specific choice point. This way, the majority of the grammar can remain LL(1) and hence perform better, while at the same time one gets the flexibility of LL(k) grammars. Table 3-12 shows the **bnf_production** of a Lucid syntax production and table 3-11 is the original production:

```
factor ::= id
           const
           E[E,…,E](E,…,E)
           (E)
           not(E)
           indexicalExp
```

Table 3-11 : A production of original Lucid syntax

JavaCC does not support production with left-recursion. Before using **bnf_production**, all syntax productions must be left-recursion free. [3]

```
void Factor() :
{ }
{       <ID> {System.out.println("Factor-> <ID>");}
      | <CONST>
      | E() "[" (E())+ "]" "(" (E())+ ")"
      | "(" E() ")"
      | "not" "(" E() ")"
      | indexicalExp()
}
```

Table 3-12 : A concrete BNF production

3.2.5.3 Regular expression production

A regular expression production is used to define lexical entities that get processed by the generated token manager. The complete syntax of regular_expr_production is table 3-13.

```
regular_expr_production ::=
      [ lexical_state_list ]
      regexpr_kind [ "[" "IGNORE_CASE" "]"] ":"
      "{" regexpr_spec ( "|" regexpr_spec )* "}"
```

(a) Syntax of regular expression production (regular_expr_production)

```
lexical_state_list ::=
        "<" "*" ">"
      | "<" java_identifier ("," java_identifier)*
">"
```

(b) Syntax of lexical state list (lexical_state_list)

```
regexpr_kind ::=
        "TOKEN"
      | "SPECIAL_TOKEN"
      | "SKIP"
      | "MORE"
```

(c) Syntax of regular expression kind (regexpr_kind)

```
regexpr_spec ::=
      regular_expression
      [ java_block ]
      [ ":" java_identifier ]
```

(d) Syntax of regular expression specification (regexpr_spec)

Table 3-13 : Syntax of regular expression production

A regular expression production starts with a specification of the lexical states for which it applies (the lexical state list). There is a standard lexical state called "DEFAULT". If the lexical state list is omitted, the regular expression production applies to the lexical state "DEFAULT". [3]

A token is matched as follows: All regular expressions in the current lexical state are considered as potential match candidates. The token manager consumes the maximum number of characters from the input stream possible that match one of these regular expressions. That is, the token manager prefers the longest possible match. If there are multiple-longest matches (of the same length), the regular expression that is matched is the one with the earliest order of occurrence in the grammar file. As mentioned above, the token manager is in exactly one state at any moment. At this moment, the token manager only considers the regular expressions defined in this state for matching purposes After a match, one can specify an action to be executed as well as a new lexical state to move to. If a new lexical state is not specified, the token manager remains in the current state. There are four kinds of token. The regular expression kind specifies what to do when a regular expression has been successfully matched:

- **SKIP**: Simply throw away the matched string (after executing any lexical action).
- **MORE**: Continue (to whatever the next state is) taking the matched string along. This string will be a prefix of the new matched string.
- **TOKEN**: Create a token using the matched string and send it to the parser (or any caller).
- **SPECIAL_TOKEN**: Creates a special token that does not participate in parsing. Already described earlier.

The java_block in table 3-13-(d) is the place where we put lexical action. Table 3-14 shows how JavaCC regular_expr_production specify token identifier. The expressions, like "\u0024", means that this handles uncode. [3]

```
TOKEN : /* IDENTIFIERS */
{       < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
      | < #LETTER:[
            "\u0024", "\u0041"-"\u005a",
            "\u005f", "\u0061"-"\u007a",
            "\u00c0"-"\u00d6", "\u00d8"-"\u00f6",
            "\u00f8"-"\u00ff", "\u0100"-"\u1fff",
            "\u3040"-"\u318f", "\u3300"-"\u337f",
            "\u3400"-"\u3d2d", "\u4e00"-"\u9fff",
            "\uf900"-"\ufaff"]>
      | < #DIGIT:[
            "\u0030"-"\u0039", "\u0660"-"\u0669",
            "\u06f0"-"\u06f9", "\u0966"-"\u096f",
            "\u09e6"-"\u09ef", "\u0a66"-"\u0a6f",
            "\u0ae6"-"\u0aef", "\u0b66"-"\u0b6f",
            "\u0be7"-"\u0bef", "\u0c66"-"\u0c6f",
            "\u0ce6"-"\u0cef", "\u0d66"-"\u0d6f",
            "\u0e50"-"\u0e59", "\u0ed0"-"\u0ed9",
            "\u1040"-"\u1049"]>}
```

Table 3-14 : The regular expression of token identifier in JavaCC format

3.2.5.4 Token manager declaration

There is a set of Java declarations and statements (Java block) in token_manager_decls. These declarations and statements are written into the generated token manager and are accessible from within all lexical actions. So, they are also called common token action. There can only be one token manager declaration in a JavaCC grammar file. Table 3-15 is the syntax of token_manager_decls. And Table 3-16 is a concrete sample of token_manager_decls. [3]

```
token_manager_decls ::=
      "TOKEN_MGR_DECLS" ":"
      java_block
```

Table 3-15 : The syntax of token manager declaration

```
TOKEN_MGR_DECLS :
{
    static int count=0;
    static void CommonTokenAction(Token t) {
        System.out.println(t.image);
    }
}
```

Table 3-16 : Sample of token manager declaration

4. Building a Lexical Analyzer with JavaCC

4.1 GIPL and Indexical Lucid SIPL Lexical Specifications

According to the original Lucid grammar, we drew up the lexical definition as shown in Table 4-1:

```
                    ID  ::=  <LETTER> ( <LETTER> | <DIGIT> )*
                LETTER  ::=  "\u0024","\u0041"-"\u005a","\u005f",
                             "\u0061"-"\u007a","\u00c0"-"\u00d6",
                             "\u00d8"-"\u00f6","\u00f8"-"\u00ff",
                             "\u0100"-"\u1fff","\u3040"-"\u318f",
                             "\u3300"-"\u337f","\u3400"-"\u3d2d",
                             "\u4e00"-"\u9fff","\uf900"-"\ufaff"
                 DIGIT  ::=  "\u0030"-"\u0039","\u0660"-"\u0669",
                             "\u06f0"-"\u06f9","\u0966"-"\u096f",
                             "\u09e6"-"\u09ef","\u0a66"-"\u0a6f",
                             "\u0ae6"-"\u0aef","\u0b66"-"\u0b6f",
                             "\u0be7"-"\u0bef","\u0c66"-"\u0c6f",
                             "\u0ce6"-"\u0cef","\u0d66"-"\u0d6f",
                             "\u0e50"-"\u0e59","\u0ed0"-"\u0ed9",
                             "\u1040"-"\u1049"
                 CONST  ::=  <INTEGER_LITERAL>
                          |  <FLOATING_POINT_LITERAL>
       INTEGER_LITERAL  ::=  <DECIMAL_LITERAL> ["l","L"]
                          |  <HEX_LITERAL> ["l","L"]
                          |  <OCTAL_LITERAL> ["l","L"])
       DECIMAL_LITERAL  ::=  ["1"-"9"] (["0"-"9"])*
           HEX_LITERAL  ::=  "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+
         OCTAL_LITERAL  ::=  "0" (["0"-"7"])*
FLOATING_POINT_LITERAL  ::=  (["0"-"9"])+"."(["0"-"9"])*[<EXPONENT>]
                             ["f","F","d","D"]
                          |  "."(["0"-
                             "9"])+[<EXPONENT>]["f","F","d","D"]
                          |  (["0"-"9"])+<EXPONENT>["f","F","d","D"]
                          |  (["0"-"9"])+[<EXPONENT>]["f","F","d","D"]
            EXPONENT  ::=  [ 'e' , 'E' ] [ '+' , '-' ] [ '0' - '9' ]
```

Table 4-1 : Indexical Lucid Lexical Specifications

4.2 Indexical Lucid SIPL Lexical Convention

Besides the definition show above, we also defined the following Lucid lexical conventions.

Case Sensitiveness

The target parser is case-sensitive.

White Space

Space, tab, return and new line are all treated as white space and will be skipped.

Comments

Our target Lucid parser is going to have three kinds of comments, just like the common Java compiler. They are single line comment, formal comment (/** ..... */) and multi line comment (/* ..... */).

4.3 Indexical Lucid SIPL JavaCC Lexical Specification

Based on Lucid lexical definition and convention, Lucid lexical specification in JavaCC format is showed as excerpt 4-1. The reason why we give each operator or reserved word a name is that JavaCC will assign a unique code for each kind of token. The name of that kind of token has a unique connection to its code.

```
SKIP : /* WHITE SPACE */{
        " " | "\t" | "\n" | "\r" | "\f"}
SPECIAL_TOKEN : /* COMMENTS */{
        <SINGLE_LINE_COMMENT:"//"(~["\n","\r"])*
        ("\n"|"\r"|"\r\n")>
      | <FORMAL_COMMENT:"/**" (~["*"])* "*" ("*" | (~["*","/"]
        (~["*"])* "*"))* "/">
      | <MULTI_LINE_COMMENT: "/**" (~["*"])* "*" ("*" |
        (~["*","/"] (~["*"])* "*"))* "/">}
TOKEN : /* RESERVED WORDS AND LITERALS */{
```

```
          < DIMENSION: "dimension" >
        | < ELSE: "else" > | < END: "end" >    | < FI: "fi" >
        | < IF: "if" >   | < THEN: "then" > | < WHERE: "where" >}
TOKEN : /* OPERATORS */{
          < ASSIGN: "=" >
        | < GT: ">" >         | < LT: "<" >          | < EQ: "==" >
        | < LE: "<=" >        | < GE: ">=" >         | < NE: "!=" >
        | < OR: "or" >       | < AND: "and" >       | < NOT: "not" >
        | < PLUS: "+" >       | < MINUS: "-" >
        | < STAR: "*" >       | < SLASH: "/" >      | < REM: "%" >
        | < ASA: "asa" >     | < FBY: "fby" >      | < UPON: "upon" >
        | < WVR: "wvr" > | < FIRST: "first" > | < NEXT: "next" >
        | < PREV: "prev" >   | < ISEOD: "iseod" >
        | < AT: "@" >         | < WHEN: "#" >}
TOKEN : /* LITERALS */{
          < CONST:
                <INTEGER_LITERAL>
              | <FLOATING_POINT_LITERAL>>
        | < INTEGER_LITERAL:
                <DECIMAL_LITERAL> (["l","L"])?
              | <HEX_LITERAL> (["l","L"])?
              | <OCTAL_LITERAL> (["l","L"])?>
        | < #DECIMAL_LITERAL: ["1"-"9"](["0"-"9"])*>
        |<#HEX_LITERAL:"0"["x","X"](["0"-"9","a"-"f","A"-"F"])+>
        | < #OCTAL_LITERAL: "0"(["0"-"7"])*>
        | < FLOATING_POINT_LITERAL:
                (["0"-"9"])+"."(["0"9"])*(<EXPONENT>)?
                (["f","F","d","D"])?
              | "."(["0"-"9"])+(<EXPONENT>)?(["f","F","d","D"])?
              | (["0"-"9"])+<EXPONENT>(["f","F","d","D"])?
              | (["0"-"9"])+(<EXPONENT>)?["f","F","d","D"]>
        | < #EXPONENT: ["e","E"](["+","-"])?(["0"-"9"])+>}
TOKEN : /* ID */{
          < ID: <LETTER> (<LETTER>|<DIGIT>)* >
        | < #LETTER: [
                "\u0024","\u0041"-"\u005a",
```

```
                "\u005f","\u0061"-"\u007a",
                "\u00c0"-"\u00d6","\u00d8"-"\u00f6",
                "\u00f8"-"\u00ff","\u0100"-"\u1fff",
                "\u3040"-"\u318f","\u3300"-"\u337f",
                "\u3400"-"\u3d2d","\u4e00"-"\u9fff",
                "\uf900"-"\ufaff"]>
        | < #DIGIT: [
                "\u0030"-"\u0039","\u0660"-"\u0669",
                "\u06f0"-"\u06f9","\u0966"-"\u096f",
                "\u09e6"-"\u09ef","\u0a66"-"\u0a6f",
                "\u0ae6"-"\u0aef","\u0b66"-"\u0b6f",
                "\u0be7"-"\u0bef","\u0c66"-"\u0c6f",
                "\u0ce6"-"\u0cef","\u0d66"-"\u0d6f",
                "\u0e50"-"\u0e59","\u0ed0"-"\u0ed9",
                "\u1040"-"\u1049"]>}
TOKEN : /* SEPARATORS */{
          < LPAREN: "(" >      | < RPAREN: ")" >
        | < LBRACKET: "[" >    | < RBRACKET: "]" >
        | < SEMICOLON: ";" >   | < COMMA: "," >}
```

excerpt 4-1 : Indexical Lucid lexical specification in JavaCC format

4.4 Notes on the Generated Lexical Analyzer

4.4.1 General Notes

Based on lexical specifications, JavaCC generates a set of Java source files, which constitute the token manager. The main method of the target parser will create and initiate an object of token manager. This token manager creates a Token object for each match of a regular expression and returns it to the parser.

Although lexical specifications and syntactical specifications should be put together to form a JavaCC grammar file, although lexical specifications can be defined even inside JavaCC BNF-production, the relationship between

scanner and analyzer can be much looser. This looser relationship is manifested with the following:

- in grammar file, we can totally separate those 2 kinds of specifications, as long as we do not introduce the lexical specification into BNF-production.
- Lexical action resides in regular expression production and common token action, while syntactical action resides in BNF-production only.
- When JavaCC option BUILD_PARSER is set to false (default is true), JavaCC will generate token manager related Java files only.
- Whe JavaCC option BUILD_TOKEN_MANAGER is set to false (default is true), JavaCC will re-generate parser related Java files only, while the scanner related Java files will remain the formal version.

This loose relationship can facilitate the development and debugging of lexical and syntactical specifications by confining the errors into related area. Note that defining regular expression inside BNF-productions is prohibited.

4.4.2 Token-related API

4.4.2.1 Token class

The Token class is the type of token objects that are created by the token manager after a successful scanning of the token stream. Each Token object has the following important fields:

- **int kind :** this is the index for this kind of token in the internal representation scheme of JavaCC. When tokens in the JavaCC input file are given labels, these labels are used to generate "int" constant that can be used in actions. We can find the internal representation scheme in generated Java file, *Parsername*Constants.java.
- **int beginLine, beginColumn, endLine, endColumn :** these indicate the beginning and ending positions of the token as it appeared in the input stream.

- **String image :** this represents the image of the token as it appeared in the input stream.
- **Token next** : a reference to the next regular (non-special) token from the input stream. If this is the last token from the input stream, or if the token manager has not read tokens beyond this one, this field is set to null.
- **Token specialToken** : this field is used to access special tokens that occur prior to this token. If there are no such special tokens, this field is set to null. [3]

We can get more detailed token information by directly accessing these fields in JavaCC grammar file. For example, the following code will print out the image and position of the current token, see excerpt 4-2:

```
Token t;
...
System.out.println(t.image + "   " +
                   t.beginLine + "   " +
                   beginColumn);
...
```

excerpt 4-2 : Code for printing information about tokens

4.4.2.2 Reading Tokens from the Input Stream

In the generated parser, the parser class has two methods available for this purpose:

- **Token *ParserName*.getNextToken( ) throws ParseError.** This method returns the next available token in the input stream and moves the token pointer one step in the input stream. If there are no more tokens available in the input stream, the exception ParseError is thrown. Care must be taken when calling this method since it can

interfere with the parser's knowledge of the state of the input stream, current token, etc.

- **Token TheParser.getToken(int index) throws ParseError**. This method returns the index-th token from the current token ahead in the token stream. If index is 0, it returns the current token (the last token returned by getNextToken or consumed by the parser). If index is 1, it returns the next token (the next token that will be returned by getNextToken of consumed by the parser) and so on. The index parameter cannot be negative. This method does not change the input stream pointer (i.e., it does not change the state of the input stream). If an attempt is made to access a token beyond the last available token, the exception ParseError is thrown. If this method is called from a semantic lookahead specification, which in turn is called during a lookahead determination process, the current token is temporarily adjusted to be the token currently being inspected by the lookahead process. [3]

4.4.3 Generated Token Manager-Related Source Files

If our JavaCC grammar file names Lucid.jj, JavaCC will generate the following token manager-related Java files.

- **`Token.java`:** this file defines the class of Token.
- **`LucidTokenManager.java`:** class of token-manager of Lucid parser.
- **`TokenMgrError.java`:** class that handles the lexical error during the parsing.
- **`SimpleCharStream.java`:** manager the input char stream.
- **`LucidConstants.java`:** include both lexical constants and syntactical constants, i.e. the internal integer representation of each kind of token.

- **`Lucid.java:`** the entrance of the parser. It constructs and initiates token manager.

Generally, we do not have to do any modification to those generated files (except Lucid.java) when we want to customize the target parser. [3]

4.5 Lexical Error Management

4.5.1 General Error Treatment Mechanism of JavaCC

Whenever the token manager detects a problem, it throws the exception TokenMgrError, and whenever the parser detects a problem, it throws the exception ParseException. If the thrown exceptions are never caught, then a standard action is taken by the Java virtual machine that normally includes printing the stack trace and also the result of the "toString" method in the exception.

Exceptions in Java are divided into two broad categories, namely Errors and Exceptions. Errors are exceptions that one is not expected to recover from. Exceptions other than errors are typically defined by subclassing the exception "Exception". These exceptions are typically handled by the user program. [8]

The exception TokenMgrError is a subclass of Error, while the exception ParseException is a subclass of Exception. The reasoning here is that, in JavaCC, the token manager is never expected to throw an exception - we must be careful in defining the token specifications such that we cover all cases. We do not have to worry about this exception - if we have designed the Lucid tokens well, it should never get thrown. Although if we still want to recover from token manager errors, it is just that we are not forced to catch them. Hence, all JavaCC error report and recovery tasks are taken by parser exception. [3]

As to error reporting, after version 0.6 (we are using version 0.7.1), JavaCC never explicitly print error message, rather this information is stored inside the parser exception objects that are thrown. User can customize the information and style of printed error message by modifying the generated java file, ParseException.java. While the error recovery has to be implemented in JavaCC grammar file.

4.5.2 GIPL and Indexical Lucid SIPL Lexical Error Specification

Tokens that do not match any lexical specification specified in section 2.3, or chars that do not appear in any regular expression production in section 2.3 are considered as lexical errors.

4.5.3 GIPL Lexical Error Treatment

According to JavaCC error reporting and recovery mechanism (section 2.5.1), JavaCC only provides fully support for reporting and recovery of parsing error. As matter of fact, generated parser will not capture any lexical error. To have a lexical-error-free parser, we have to go through the following way:

- First, be careful in defining token specifications such that they are bug-free and can cover all cases, for example, define a token type "ILLEGAL CHARACTERS", which covers all possible chars excluded by Lucid lexical specifications.
- Second, all errors are going to be treated as parser errors. For example, when encounter an ILLEGAL CHARACTER, say '?', parser will just print '?' as a token and report that an unexpected token encountered. Another example is that suppose there is a wrong decimal, say "34.65.08", in input stream, the parser will report a syntactical error that the second point is an unexpected token. [3]

5. Building a Syntactical Analyzer with JavaCC

5.1 Problems in the Original Grammar

JavaCC is a LL(K) parser generator. That is to say, JavaCC cannot accept the syntax productions that contain left-recursion, while it can handle any ambiguities in syntax productions. Nevertheless, we have to reduce the ambiguities as much as possible, since the ambiguity may reduce the efficiency of the target parser due to the necessity of backtracking.

Checking the original grammar, we see that there are many left-recursions, some of them are immediate left-recursion and some are non-immediate left-recursion. Productions E, and term have immediate left-recursion. We may use regular method to eliminate the left-recursion. As to production Q, the case is a bit different. Following is the original production Q:

```
Q  ::=  dimension id,…, id;
     |  id = E ;
     |  id [id,…,id] (id,…,id) = E;
     |  Q Q
```

After eliminate the immediate left-recursion with the regular method, production Q turns to be:

```
Q  ::=  dimension id, … , id; Q1
     |  id = E ; Q1
     |  id [id,…,id] (id,…,id) = E; Q1
Q1 ::=  ε
     |  Q Q1
```

First of all, suppose that Q1 :: = ε, then:

```
Q  ::=  dimension id,…,id;
     |  id = E ;
     |  id [id,…,id] (id,…,id) = E;
```

Secondly, suppose that Q1 ::= QQ1, and the second Q1 (the one in right hand side) ::= ε, yet then:

```
Q  ::=  dimension id,…,id; Q
     |  id = E ; Q
     |  id [id,…,id] (id,…,id) = E; Q
```

Thirdly, suppose that Q1 ::= QQ1, then the Q1 that is in right hand side is yet replaced by QQ1. If we let the last Q1 to be ε, the original production is going to be Q1 ::= QQ. Hence

```
Q  ::=  dimension id,…,id; QQ
     |  id = E ; QQ
     |  id [id,…,id] (id,…,id) = E; QQ
```

Repeat the 2nd and 3rd step, we will finally get the following result:

```
Q  ::=  dimension id,…,id; (Q)*
     |  id = E; (Q)*
     |  id [id,…,id] (id,…,id) = E; (Q)*
```

There is a non-immediate left-recursion chain between production E, term and factor. If we directly eliminate the left-recursion with the regular method, we are going to make the situation worse. The updated productions are going

to be much more complicated. The better solution is to break the chain. By adjusting some original productions, we can break the non-immediate left-recursion chain. After the modification, there is no more left-recursion in production factor. Table 5-1 shows the comparison of the original productions to the updated. The original syntax and implication are not influenced by the modification. Table 5-2 shows the productions after eliminating the left-recursion.

original

```
    E ::= term
        | sign term
        | E addOp term
        | E relOp E
        | if E then E else E fi
        | #.E
        | E@.E E
        | E where Q end

 term ::= term multOp factor
        | factor
facto ::= id
    r
        | const
        | E[E,...,E](E,...,E)
        | (E)
        | not(E)
        | indexicalExp
```

updated

```
    E ::= term
        | sign term
        | E addOp term
        | E relOp E
        | if E then E else E fi
        | #.E
        | E@.E E
        | E where Q end
        | E[E,...,E](E,...,E)
 term ::= term multOp term
        | factor
facto ::= id
    r
        | const

        | (E)
        | not(E)
        | indexicalExp
```

Table 5-1 : Original productions (E, term and factor) and updated version (in order to break the non-immediate left-recursion)

```
     E  ::=  term E1
          |  sign term E1
          |  if E then E else E fi E1
          |  # . E  E1
    E1  ::=  addOp term E1
          |  relOp E E1
          |  @ . E E  E1
          |  where Q end E1
          |  [E,…,E](E,…,E) E1
          |  ε
  term  ::=  factor term1
 term1  ::=  multOp term term1
          |  ε
```

Table 5-2 : Productions E and term after eliminating the left-recursion

Finally, concerning the ambiguity, JavaCC is capable to figure out the ambiguity in the grammar and give related comments. After eliminating all left-recursions, we can set out transforming the Lucid grammar into JavaCC grammar file format. JavaCC processes the grammar file and figures out that Lucid grammar is basically a LL(2) grammar. Two lookahead to here is acceptable for this project (the more lookahead, the lower the efficiency). So, we don't have to do any further work on ambiguity.

The grammar above is not the final cut for Lucid parser yet. We will re-shape it for the sake of tree building. We will discuss this later.

5.2 GIPL Syntactical Specification in JavaCC Format

Excerpt 5-1 shows how to present the Lucid syntax specifications in JavaCC grammar file.

```
void StartP() : { }
      { E() <EOF> }
void E() : { }
      { "if" E() "then" E() "else" E() "fi" E1()
      | "#" "." E()  E1()
      | Term() E1()
      | Sign() Term() E1() }
void E1() : { }
      {
      [ AddOp() Term() E1()
      | RelOp() E() E1()
      | "@" "." E() E() E1()
      | "where" Q() "end" E1()
      | "[" E() ( "," E() )* "]" "(" E() ( "," E() )* ")" E1()
      ] }
void Term() : { }
      { Factor() Term1() }
void Term1() : { }
      { [ MultOp() Term() Term1() ] }
void Factor() : { }
      { <ID>
        | <CONST>
        | "(" E() ")"
        | "not" "(" E() ")"
        | indexicalExp() }
void Q() : { }
      { "dimension" <ID> ( "," <ID> )* ";" ( Q() )*
      | <ID> "=" E() ";" (Q())*
      | <ID>
        "[" <ID> ( "," <ID> )* "]"
        "(" <ID> ( "," <ID> )* ")"
        "=" E() ";" ( Q() )* }
void IndexicalExp() : { }
      { UnOp() "[" E() ("," E())* "]" "(" E() ")"
      | "(" E() ")" BinOp() "[" E() ("," E())* "]" "(" E() ")" }
```

```
void UnOp() : { }
      { "first" | "next" | "prev" | "iseod" }
void BinOp() : { }
      { "fby" | "wvr" | "asa" | "upon" }
void Sign() : { }
      { "+" | "-" }
void AddOp() : { }
      { "+" | "-" | "or" }
void MultOp() : { }
      { "*" | "/" | "%" | "and" }
void RelOp() : { }
      { "<" | ">" | ">=" | "<=" | "==" | "!=" }
```

excerpt 5-1 : Indexical Lucid syntactical specification in JavaCC format

Comments on JavaCC-format-Lucid-grammar above:

- This grammar does not implement error recovery, which will be introduced later in section 5.4.
- This grammar does not consider tree-building related issues, which will be covered later in chapter 6.
- The start state of original grammar is E, and there exists left-recursion in productions of E. Context-free grammar disallows that the start state appears in right-hand-side of production of start state. In addition, unpredicted parsing result occurs if we break the rule in JavaCC. So, here we introduce a new start state, StartP.
- There is no counterpart of ε in JavaCC. However, we may use "[...]" to express the same implication.

5.3 Notes on the Generated Source File

If the JavaCC grammar file names Lucid.jj, JavaCC will generate the following parser-related Java files.

- Lucid.java, includes operations that construct and initiate parsing.

- ParseException.java, defines class ParseException, which is the subclass of Java Exception, to treat all parsing exceptions. We may modify this file to customize our own error reporting mechanisms.

5.4 Error Reporting and Error Recovery

Section 4.5.1, General error treatment mechanism of JavaCC, describes the basic rule of JavaCC error reporting and recovery. That is: parser generated by JavaCC does not handle lexical error; all possible errors will be reported and recovered as syntactical errors.

JavaCC provides basic error reporting and recovery mechanism. Whenever an error occurs, the generated parser creates an exception object and throws it. Default error message (prepared by JavaCC) is stored in this object. The error can be captured and recovered if we properly place try-catch block in JavaCC grammar file. If the thrown exceptions are never caught, then a standard action is taken by Java virtual machine, which normally includes printing the stack trace and also the result of the "toString" method in the exception.

By properly editing JavaCC grammar file and modifying the generated Java file "ParseException.java", we may customize our own error message and error recovery method.

5.4.1 Error Reporting and Customizing Error Message

Once an error occurs and is captured, a ParseException object will be created and thrown. Error message is stored in this object. The object's method "toString()" will transform the error message into string.

JavaCC provides the default error message. Excerpt 5-2 illustrates the default error message.

```
ParseException: Encountered ";" at line 2, column 16.
Was expecting one of:
    "if" ...
    "#" ...
    <ID> ...
    <CONST> ...
    "(" ...
    "not" ...
    "first" ...
    "next" ...
    "prev" ...
    "iseod" ...
    "+" ...
```

excerpt 5-2 : JavaCC default error message format

This message indicates the location of the current error, giving the clue of what causes the error (according to the follow set).

The error message shown in excerpt 5-2 is a bit complicated. We can customize our own error message by modifying getMessage() method in generated Java file "ParseException.java". But keep it in mind that whenever you edit grammar file and re-generate parser, ParseException.java will be replaced by new file.

In this case, we consider that the error location is enough for error reporting. So, our error message is going to be like the following:

```
ParseException: Encountered ";" at line xx, column
xx.
```

5.4.2 Error Recovery

JavaCC offers two kinds of error recovery - shallow recovery and deep recovery. Shallow recovery recovers if none of the current choices have succeeded in being selected, while deep recovery is when a choice is selected, but then an error happens sometime during the parsing of this choice. [3]

Shallow Error Recovery

Excerpt 5-3 is an example of shallow error recovery. Let us assume that IfStm starts with the reserved word "if" and WhileStm starts with the reserved word "while". When neither IfStm nor WhileStm can be matched by the next input token. That is the next token is neither "if" nor "while". Method error_skipto(SEMICOLON) can recover the error by skipping all the way to the next semicolon. error_skipto(SEMICOLON) is defined as excerpt 5-4. [3]

```
void Stm() :
{}
{ IfStm()
| WhileStm()
| error_skipto(SEMICOLON) }
```

excerpt 5-3 : An example of shallow error recovery

```
JAVACODE
   void error_skipto(int kind) {
        ParseException e = generateParseException();
       // generate the exception object.
       System.out.println(e.toString());
```

```
        // print the error message
        Token t;
        do { t = getNextToken();
        } while (t.kind != kind); }
```

excerpt 5-4 : Source code of method **error_skipto** (SEMICOLON)

Shallow error recovery can only handle certain problems. In this case, we use deep error recovery.

Deep Error Recovery

A similar example is shown in excerpt 5-5. We wish to recover even when there is an error deeper into the parse. For example, suppose the next token was "while" - therefore the choice "WhileStm" was taken. But suppose that during the parse of WhileStm some error is encountered - say one has "while (foo { stm; }" - i.e., the closing parentheses has been missed. Shallow recovery will not work for this situation. We need deep recovery to achieve this. In this case, we use the try-catch-finally block. [3]

```
void Stm() :
{}
{ try {
        ( IfStm()
        | WhileStm()
        )
  catch (ParseException e) {
       error_skipto(SEMICOLON);
        } }
```

excerpt 5-5 : An example of deep error recovery

Try-catch-finally block can be applied quite flexibly. For example, we can even apply try-catch block to an expansion_unit.

5.5 Complete GIPL and Indexical Lucid SIPL Grammar File

So far, we can use JavaCC to generate a complete Lucid parser, which can do basic error report and error recovery. While tree building action has not been available in this stage untill later in chapter 6. Appendix I lists the complete source code of GIPL Lucid grammar file (Lucid.jj) in JavaCC format, and updated ParseException.java file, which includes the customized error message. Parser generated with this grammar file can parse any Lucid program and display the error message on standard output. Next chapter, we will modify this grammar file and add tree building related elements.

5.6 Making an Executable Parser

The following is the procedure of generating an executable Lucid parser:

- Set up JDK1.3 environment under either Unix/Linux, Windows 2000, or MacOS X system
- Download and setup JavaCC, then copy JavaCC application to working directory
- Edit Lucid grammar file (Appendix I) with pure text editor, the file name has the suffix of .jj, in this case Lucid.jj i.e.
- Invoke JavaCC to process the grammar file, the command line is like "javacc Lucid.jj"
- Modify ParseException.java source to customize the error message
- Compile all generated java files, input "javac *.java" under working directory
- A bunch of java class files is generated and they compose the first executable Lucid parser, the entry class is "Lucid"

6. Abstract Syntax Tree Building Using JJTree

6.1 JJTree and JavaCC

JavaCC can independently generate a no-tree-building-function parser. The only source that JavaCC takes is a .jj file, which mainly includes lexical and syntactical specifications. If we want to generate a parser that is able to construct parser tree, we have to add JJTree styled decorations into original .jj file, then change the suffix of the original file from .jj to jjt. JJTree, the pre-processor, treats this .jjt file, and outputs a .jj file that includes not only the original lexical and syntactical specifications, also tree building actions. By taking this specific .jj grammar file, JavaCC will generate a tree-building-abled-parser.

6.2 JJTree Basics

JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser.

By default JJTree generates code to construct parse tree nodes for each non-terminal in the language. This behavior can be modified so that some non-terminals do not have nodes generated, or so that a node is generated for a part of a production's expansion, or so that some terminals can have nodes generated.

JJTree operates in one of two modes, simple and multi. Multi mode just features better term. (In this case, we use SimpleNode) In simple mode each parse tree node is of concrete type SimpleNode; in multi mode the type of the parse tree node is derived from the name of the node. If you do not provide implementations for the node classes JJTree will generate sample

implementations based on SimpleNode for you. You can then modify the implementations to suit. [3]

Although JavaCC is a top-down parser, JJTree constructs the parse tree from the bottom up. To do this it uses a stack where 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 stack is open, which means that you have access to it from within grammar actions: you can push, pop and otherwise manipulate its contents however you feel appropriate.

6.3 JJTree Decorations

JJTree provides decorations for two basic varieties of nodes (simple and multi), and some syntactic shorthand to make their use convenient.

Definite Nodes

A definite node is constructed with a specific number of children. That many nodes are popped from the stack and made the children of the new node, which is then pushed on the stack itself. We notate a definite node like this:

```
#ADefiniteNode(INTEGER EXPRESSION)
```

Where after # and before (is the node name. A definite node descriptor expression can be any integer expression, although literal integer constants are by far the most common expressions. [3]

Conditional Nodes

A conditional node is constructed with all of the children that were pushed on the stack within its node scope if and only if its condition evaluates to true. If it

evaluates to false, the node is not constructed, and all of the children remain on the node stack. You notate a conditional node like this:

```
#ConditionalNode(BOOLEAN EXPRESSION)
```

A conditional node descriptor expression can be any Boolean expression. There are two common shorthands for conditional nodes:

- Indefinite nodes, **`#IndefiniteNode`** is short for **`#IndefiniteNode(true)`**
- Greater-than nodes, **`#GTNode(>1)`** is short for **`#GTNode(jjtree.arity() > 1)`**

The indefinite node shorthand (1) can lead to ambiguities in the JJTree source when it is followed by a parenthesized expansion. In those cases the shorthand must be replaced by the full expression. For example, (...) #N (a()) is ambiguous; we have to use the explicit condition: (...) #N(true) (a()).

In this case, Lucid parser generation and abstract syntax tree construction, we adopt Indefinite node decoration, since this kind of decoration is simple yet can satisfy our requirements.

By default JJTree treats each non-terminal as an indefinite node and derives the name of the node from the name of its production. We can give it a different name with the following syntax:

```
void P1() #MyNode : { ... } { ... }
```

When the parser recognizes a P1 non-terminal it begins an indefinite node. It marks the stack, so that any parse tree nodes created and pushed on the

stack by non-terminals in the expansion for P1 will be popped off and made children of the node MyNode.

If one want to suppress the creation of a node for a production one can use the following syntax:

```
void P2() #void : { ... } { ... }
```

We can further customize the generated tree, for example, like the following:

```
void P3() : {}
{
    P4() ( P5() )+ #ListOfP5s P6()
}
```

Now the P3 node will have a P4 node, a ListOfP5s node and a P6 node as children. The #Name construct acts as a postfix operator, and its scope is the immediately preceding expansion unit.

Generally, JJTree just generates node for non-terminal. That leads that terminals, which appear in the production of any non-terminal, cannot be recorded as a child in node structure. For example, if P3 is expressed as the following:

```
void P3() : {}
{
    P4() <CONST> P5()
}
```

P3-related part of tree is going to be like P3->(P4, P5), <CONST> is missed. If we re-write the example above like this:

```
void P3() : {}
{
     P4() <CONST> #Const P5()
}
```

That part of tree will be: P3->(P4, Const, P5). [3]

6.4 JJTree Options

JJTree supports options on the command line and in the JavaCC options statement. There are two options we may use in our project.

- **NODE_DEFAULT_VOID** (default: false): Instead of making each non-decorated production an indefinite node, make it void instead.
- **VISITOR** (default: false): Insert a jjtAccept() method in the node classes, and generate a visitor implementation with an entry for every node type used in the grammar. [3]

6.5 Exception Handling

An exception thrown by an expansion within a node scope is not caught within the node scope is caught by JJTree itself. When this occurs, any nodes that have been pushed on to the node stack within the node scope are popped and thrown away. Then the exception is re-thrown. The intention is to make it possible for parsers to implement error recovery and continue with the node stack in a known state.

Note that JJTree currently cannot detect whether exceptions are thrown from user actions within a node scope. Such an exception will probably be handled incorrectly. [3]

6.6 Visitor Support

JJTree provides some basic support for the visitor design pattern. If the VISITOR option is set to true JJTree will insert a jjtAccept() method into all of the node classes it generates, and also generate a visitor interface that can be implemented and passed to the nodes to accept.

The name of the visitor interface is constructed by appending Visitor to the name of the parser. The interface is regenerated every time that JJTree is run, so that it accurately represents the set of nodes used by the parser. This will cause compile time errors if the implementation class has not been updated for the new nodes. This is a feature. [3]

6.7 Tree Building of GIPL/SIPL Parser

Before we set off inserting parse tree building actions in the JavaCC source, Lucid.jj i.e. we have to concern about the following issues:

- **Use simple node.** JJTree operates in one of two modes, simple and multi. Multi mode just features better term (one java class for one kind of node). In this case, we adopt simple mode. Under simple mode, whole tree structure is much simpler and clearer, generated node-related source code is easy to read, understand and manage.
- **The content of the abstract syntax tree.** The generated abstract syntax tree is going to be used by IDS generator. So, the content of the AST is mainly determined by the requirement of IDS. For example, by default, JJTree doesn't generate node for terminal, however, semantic analysis (IDS) does need some terminals, such as <ID> and <CONST>, to exist in parse tree structure. So we have to decorate those terminals with postfix operator (section 6.3) that the decorated terminals will have nodes generated and those nodes will be pushed into tree building stack. In addition, apparently, some non-terminals are useless to be a node when construct parse tree, E1 i.e. E1 is just an

intermediate node, which is introduced for the sake of eliminating left-recursion. E1 doesn't make any sense when appear in AST. Hence, to set NODE_DEFAULT_VOID option to true, and to decorate only demanded terminals and non-terminals are simple but flexible way.

- **Data structure of AST.** Data structure of AST is basically fixed. Also, we can customize the data structure (add new attribute, i.e.) by modifying generated java source file "SimpleNode.java". Generally, we suggest the only customization is to add new attributes and related methods, because the original data structure implemented by JJTree is quite mature.
- **The interface to IDS generator modules.** Generated AST will be stored in an object of SimpleNode class. Invoking method of this object, such as dump(), IDS can trace the whole parse tree.

6.8 Manipulating AST construction

By default, JavaCC/JJTree constructs a tree in common sense, and the tree is going to have a common structure. That is, without any special treatment, the node of a production's left-hand-side is the parent of all nodes of the production's right-hand-side; the node of group decoration is the parent of the node of each sub-decoration. For example, for the following case when JJTree option NODE_DEFAULT_VOID is set to false:

```
void P1( ): { }
{
      P2( )  P3( )
}
```

The parent node is P1, and its children are P2 and P3.

For another example when option NODE_DEFAULT_VOID is set to true,

```
void P1( ): { }
{
        ( P2( ) #P2  P3( ) #P3  ) #P1
}
```

The parent node is going to be P1, and its children are, still, P2 and P3.

If we do not have any special requirements to the structure of the generated AST, or if the original syntactical specification can naturally lead to our demanded tree structure, we could have just left the Lucid syntactical specifications there as the examples above. However, as a matter of fact, the demanded AST structure is really a distortion to the original Lucid syntactical specifications. So, we have to get help from other facilities provided by JJTree.

JJTree generates a class JJTreeState and declares an object 'jjtree' in the target parser. JJTreeState class has 2 important methods by which we can manipulate the tree construction stack during tree building. They are: [3]

- **void pushNode(Node n) :** Push a node on to the stack.
- **Node popNode() :** Return the node on the top of the stack, and remove it from the stack.

As well, class SimpleNode, which is derived from class Node has the following 2 tree structure related methods:

- **void jjtSetParent(Node n) :** Set node *n* as the parent of this node.
- **void jjtAddChild(Node n, int order) :** Add node *n* as the number *order* child of this node.

By employing those methods into .jjt file, we can customize the target AST structure reaching almost any requirements.

6.9 GIPL Grammar File with Tree Building Actions

After insert abstract syntax tree building actions and change the suffix of the file name, Lucid grammar file "Lucid.jj" is transformed into "Lucid.jjt".

6.10 Generating an Executable AST-building-abled-Parser

The following is the procedure of generating an executable Lucid parser, which has tree building function, from Lucid.jjt:

- Set up JDK1.3 environment under either Unix/Linux or Windows system
- Download and setup JavaCC, then copy JavaCC, and JJTree application to working directory
- Edit Lucid.jjt (Appendix II) with pure text editor
- Invoke JJTree to pre-process Lucid.jjt, the command line is like "jjtree Lucid.jjt", jjtree application generates Lucid.jj automatically
- Then invoke JavaCC to process generated Lucid.jj, command line is "javacc Lucid.jj"
- Modify ParseException.java source to customize the error message
- Modify SimpleNode.java to customize the data structure of parse tree, if necessary
- Compile all generated java files, input "javac *.java" under working directory
- A bunch of java class files is generated and they compose the first executable Lucid parser, the entry class is "Lucid"

7. Abstract syntax tree translation

The Lucid GIPL language is the basic Lucid language, which includes a basic set of operations only, such as @ and #, etc. However, practically, we always use the Lucid languages (or other intensional programming languages, or SIPLs) that have a set of domain-specific operations. GIPL accompanied with an extended operations set becomes a SIPL. Different operation sets corresponds 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 SIPL Lucid, we have to generate different parser, and construct different AST because the RIPE environment of GIPSY system will display and evaluate the Lucid source code based on the result of the parser and the tree. Those extended operations will be treated as the basic unit of that SIPL Lucid. However, the IDS module takes the basic tree structure only. 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 contains the node of basic operations only.

Unfortunately, so far, there are no any tools for assisting this translation. We have to develop it all by ourself. The following sections are going to depict how we implement the indexical Lucid translator. And this translator can also be a template for the other SIPL Lucid.

7.1 GIPL parser and tree construction

In order to get the demanded AST structure, at the beginning, we employed some direct tree manipulating methods mentioned in section 6.8, popNode( ) for example. The grammar file looks like excerpt 7-1 after adding parsing and tree building action.

```
void Term1( ) :
{ }
{
 [(("*" #TIMES |"/" #DIV |"%" #MOD |"and" #AND) Term( ))
    { Node midNode1=jjtree.popNode(); // right
      Node midNode2=jjtree.popNode(); // MultOp
      Node midNode3=jjtree.popNode(); // left
      midNode1.jjtSetParent(midNode2);
      midNode3.jjtSetParent(midNode2);
      midNode2.jjtAddChild(midNode3, 0);
      midNode2.jjtAddChild(midNode1, 1);
      jjtree.pushNode(midNode2);
    }
 Term1( )  ]
}
```

excerpt 7-1 : Example of direct tree building manipulation

7.2 Indexical SIPL parser and tree construction

Basically, SIPL grammar file looks similar to GIPL grammar file, except that SIPL has more productions about those extended operations and more tree building actions.

7.3 About extended operations

Extended operations are higher level and more powerful than those basic operations. Furthermore, those extended operations are in different levels too. Any extended operation can be expressed by a combination of basic operations and/or lower level extended operations. That is also the foundation of tree translation. The followings are some indexical Lucid extended operations and their resolutions.

```
first.d X = X @.d 0;

X upon.d Y = X @.d W
     where
     W = 0 fby.d if Y then (W+1) else W;
     end;
```

7.4 Upgrade the SIPL grammar file (parsing and tree building actions)

Based on the analysis above, all extended operations can be expressed by lower level operations. By extending this idea, we can speculate that any extended operation can be rendered as a series of individual or nesting functions of lower level operations.

We change the basic tree manipulation part as a function whose name is same as the operation's name so that the upgraded grammar can facilitate our tree translation. For example, for the basic operation *at*, excerpt 7-2 shows the grammar file code of before and after upgrade.

```
.....
   | ( "@" #AT "." E() E() )
   { SimpleNode midNode1=(SimpleNode) jjtree.popNode();
                                        // right1
     Node midNode2=jjtree.popNode(); // right2
     Node midNode3=jjtree.popNode(); // AT
     Node midNode4=jjtree.popNode(); // left
     midNode1.jjtSetParent(midNode3);
     midNode2.jjtSetParent(midNode3);
     midNode4.jjtSetParent(midNode3);
     midNode3.jjtAddChild(midNode4, 0);
     midNode3.jjtAddChild(midNode2, 1);
```

```
      midNode3.jjtAddChild(midNode1, 2);
      jjtree.pushNode(midNode3);
    }
 E1()
  ......
```

(a) Before upgrade

```
JAVACODE  SimpleNode  at(SimpleNode  node1,  SimpleNode
node2,
                      SimpleNode    node3,    SimpleNode
node4)
                      // right2 ,1, at, left (node1-4)
{    node1.jjtSetParent(node3);
     node2.jjtSetParent(node3);
     node4.jjtSetParent(node3);
     node3.jjtAddChild(node4, 0);
     node3.jjtAddChild(node2, 1);
     node3.jjtAddChild(node1, 2);
     return node3;
}
....
| ( "@" #AT "." E() E( ) ) {
SimpleNode   midNode1=(SimpleNode)   jjtree.popNode();//
right1          SimpleNode          midNode2=(SimpleNode)
jjtree.popNode();// right2
SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // AT
SimpleNode  midNode4=(SimpleNode)  jjtree.popNode();  //
left
```

```
SimpleNode node=at(midNode1, midNode2, midNode3,
midNode4);
jjtree.pushNode(node); }
E1( )
```

(b) after upgrade

excerpt 7-2 : Make tree building action into function

7.5 SIPL translator

After the improvement above, the structure and the algorithm of the translator turns much clearer than ever. The translator is very specific. Each SIPL parser has to accompany its own translator.

Here in this project, we declare a class *Translator*. The main part of the class is a recursive method, which recursively travels the SIPL tree, calls those basic operation functions to replace indexical Lucid extended operations in the generated SIPL tree and finally, outputs GIPL tree.

The interface of the translator is quite simple:

SimpleNode translate(SimpleNode n);

The SIPL tree reference is the input, and the translated GIPL tree reference is the output.

8. Merging the Generated Parsers and translator into the GIPSY System

The purpose of those two Lucid grammar files listed in Appendix I and II is to show how to use JavaCC and JJTree application to generate the GIPL parser and add tree-building actions. Lucid parsers generated with those two grammar files are independent and executable Java applications. We may invoke them to parse Lucid program and generate abstract syntax tree directly from command line. However, the goal of this project is to implement a Lucid parser and/or translator, if it was an SIPL parser, for GIPSY system where the parser/translator is treated as a module and invoked through GIPSY RIPE or other modules, and finally, transfer the generated/translated abstract syntax tree to other modules. In addition, this Lucid parser/translator has to be feasible to be updated in run-time. So what transformation of those grammar files will lead us to our goal?

8.1 Rationale

First of all, JavaCC does not require that the class, which has parser's name (Parser in this case), be the entry class. That makes it possible for us to invoke the parser from the Java class of other modules.

In addition, the compilation part of grammar file allows us to define and declare any attribute and method for the parser class. These attributes and methods can be the communication medium between parser module and other module.

Furthermore, JavaCC/JJTree makes it possible for more than one parser working together as long as we give them different name. So, the parser/translator module can be extended to the module, which contains a GIPL parser, one or more SIPL parsers and their translators.

Although generated Lucid parsers, and translators, are composed of a bunch of java classes, information exchanged between parser/translator module and other module is limited and predictable. Updating Lucid grammar will not change the component of the module, neither will it affect the way that parser/translator module communicates with other module. The parser/translator module exchange the following information with other modules:

- Construct parser and start parsing from other modules, those modules have to provide the filename to be parsed.
- Parser transfer error message(s) and/or other prompt messages, in pure string data type, to the calling module.
- Parser returns the constructed abstract syntax tree, a SimpleNode object, to IDS generator module.
- Other modules invoke a translator to translate the corresponding SIPL tree and get the GIPL tree.

Although there are only four kinds of messages exchanging between parser and other modules, still we have to introduce some object-oriented design pattern due to the following reasons:

- Due to frequent evolution of Lucid grammar, it is necessary for parser/translator module to be conveniently updated during run-time.
- Directly communicate with other module leads to high couple between modules/classes.
- The evolution of GIPSY system is unpredictable. So far, we know that only IDS need the parse tree generated by parser. It is possible, in the future, that we may dynamically check the parse tree from RIPE.

The Façade design pattern is easy but is the best candidate for handling those issues. Façade pattern may avoid excess coupling between modules/classes by providing an additional interface layer. One possible disadvantage of the Façade pattern is that one may lose some functionality contained in the lower level of classes. Fortunately, in this case, parser module has only limited and predictable information that has to be exchanged with other modules. [5]

8.2 Façade Design Pattern and the Parser/translator Modules

Before introduce any design pattern, parser/translator module has to directly communicate with IDS and RIPE module. Messages between parser/translator and other modules are:

- RIPE constructs GIPL or SIPL parser and initiates it.
- Parser transfers error messages and other messages to RIPE.
- Parser returns the generated abstract syntax tree to the Semantic Analyzer and Dataflow Graph Generators.
- RIPE invokes translator and gets the translated GIPL tree.

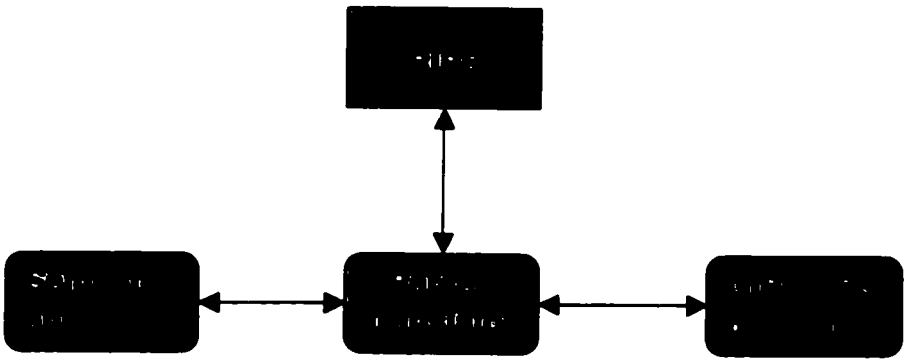


Figure 8-1 : Coupling between Parser/translator, Semantic Analyzer, Dataflow Graph Generator, and RIPE

In fact, the parser/translator exchanges only four kinds of information with other modules: initiating parser, transferring error/other messages, returning abstract syntax tree and returning translated GIPL abstract syntax tree. We may introduce an extra class, facade, and let parser communicate with facade only. This façade itself must be simple, clear, and have a limited amount of actual code. Figure 8-2 shows the structure after introduce the Façade class.

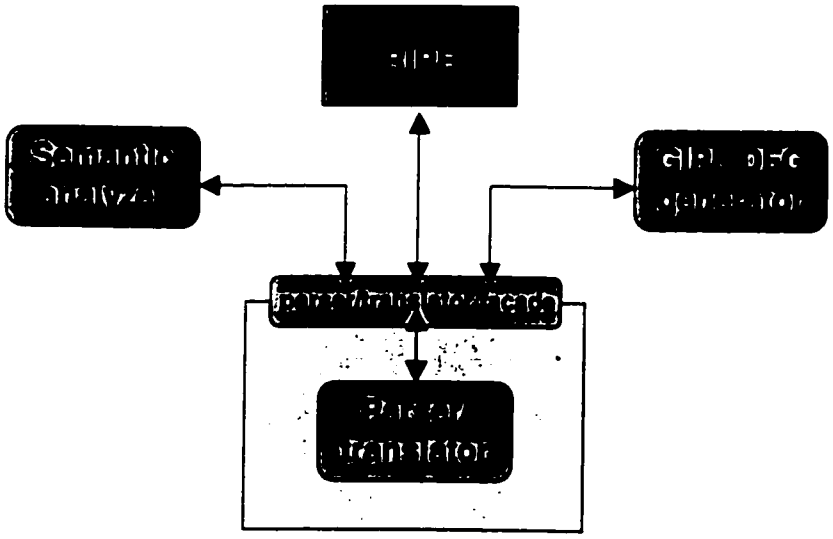


Figure 8-2 : Parser module with façade

8.3 Implementation of the parser module

So far, we have implemented a practical parser/translator module for the GIPSY system. This module includes the Façade class, GIPL Lucid classes and indexical Lucid SIPL classes generated by JavaCC/JJTree, and the translator class. Excerpt 8-1 is the interface of Façade class. Appendix IX shows the complete source code of façade.java.

```
public class Facade {
     SimpleNode simpleNode;    // reference of the AST
     Parser parser;            // object of GIPL parser
     SIPL sipl;                // object of SIPL parser
     String filename;          // file to be parsed
     String message;           // various messages
     Translator translator;    // object of the translator
     void parsing(String filename) { }
     // choose GIPL or SIPL and start parsing
     void display(String message) { }
     // display error message or other information
     void treepass(SimpleNode simpleNode) { }
     // parser will pass the tree to façade by this
method
     void showtree() { }
     // dump the tree, for testing only
     void translate() { }
     // call translator
}
```

excerpt 8-1 : A sample source of façade class

Lucid grammar file, consequently, has to be adjusted due to the façade class. While this adjustment is fixed and not affected by Lucid grammar evolution.

9. Result Evaluation

9.1 Parser Testing

The quality is always the most important issue for software. In order to control the bugs to the lower level, the testing goes through the whole course of the Lucid parser/translator development.

Because the development is an incremental process, the testing mechanism we employed was a bottom-up testing process. The procedure of the parser/translator development and testing is:

- Define lexical specifications of GIPL Lucid, and make the first grammar file;
- Generate lexical analyzer with JavaCC;
- Define syntactical specifications of GIPL Lucid, and add them into the previous grammar file;
- Generate stand-alone GIPL parser with JavaCC;
- Add tree-building action into the previous grammar file;
- Generate stand-alone AST-building-abled GIPL parser with JavaCC/JJTree;
- Develop indexical Lucid grammar file based on GIPL grammar file. The lexical part of any GIPL or SIPL is all the same. Syntactical part of SIPL is a bit different from the one of GIPL, because SIPL has to add its extended operations into GIPL syntax;
- Develop the translator for indexical Lucid;
- Introduce façade design pattern;
- Modify the compilation part of GIPL and SIPL grammar file so that the generated GIPL, SIPL parsers and SIPL translator can work together

as one module and have the façade be their interface to the other modules.

Our testing to 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.

9.1.1 GIPL Parser Testing

For GIPL parser testing, mainly, we check the parsing error report and/or the output abstract tree.

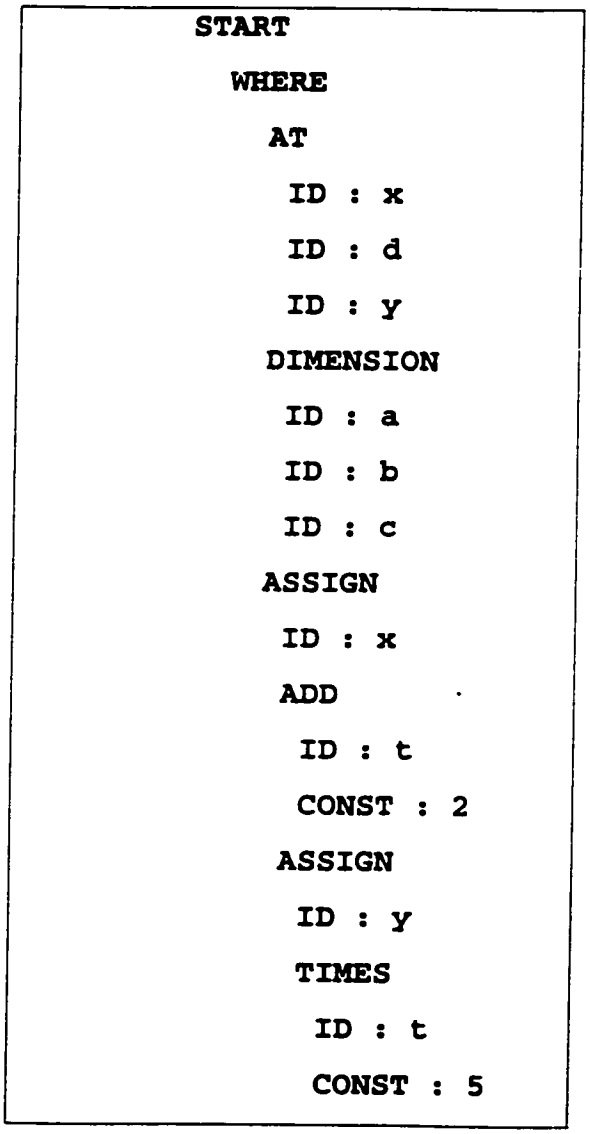
The following is the error message GIPL parser provides, when error occurred:

```
Lucid Parser Version 0.0 (GIPL):  Reading from file test.txt . . .
ParseException: Encountered "wvr" at line 1, column 5.
Lucid Parser Version 0.0:  1 errors.
```

Excerpt 9-1 shows an example of a correct GIPL program and the parsing result as well as the generated abstract syntax tree (dump).

```
( x @ . d y)
where
        dimension a,b,c;
        x = t + 2;
        y = t * 5;
end
```

(a) A GIPL program



(b) Generated abstract syntax tree

excerpt 9-1 : GIPL parsing result and generated AST

9.1.2 Indexical Lucid SIPL Parser Testing

Since the indexical Lucid SIPL is developed upon the GIPL by adding extended operations syntax, the testing, therefore, focus on those extended operations.

9.2 Translator testing

Testing translator, in fact, is to check if the translator can take the output of the SIPL, the translated tree is correct and accurate.

Here is another example. Excerpt 9-2 (a) shows an original indexical Lucid program, (b) is the SIPL AST, and (c) is the translated AST (GIPL).

```
( x wvr . d y)
where
  dimension a,b,c;
  q = x + y ;
end
```

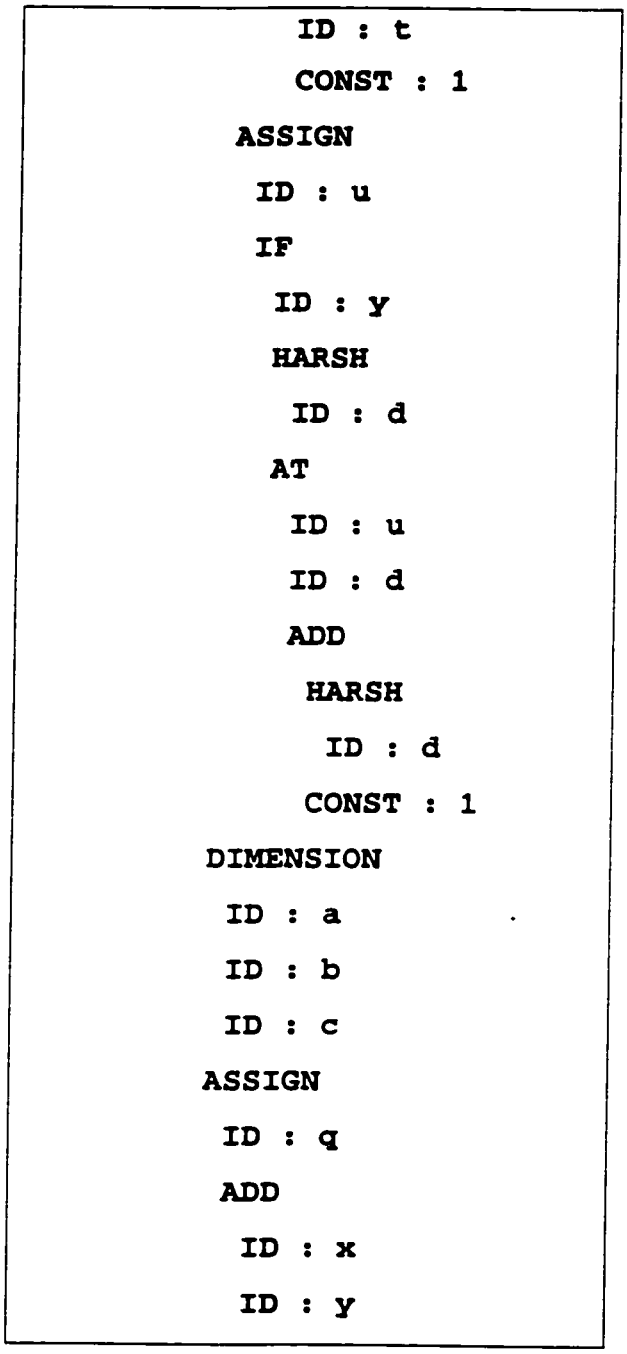
[a] original SIPL program

```
START
  WHERE
   WVR
    ID : y
    ID : d
    ID : x
   DIMENSION
    ID : a
    ID : b
    ID : c
   ASSIGN
    ID : q
```

```
ADD
 ID : x
 ID : y
```

[b] generated SIPL AST

```
START
 WHERE
  WHERE
   AT
    ID : x
    ID : d
    ID : t
   ASSIGN
    ID : t
    AT
     IF
      LE
       HARSH
        ID : d
       CONST : 0
      ID : u
      AT
       ID : u
       ID : d
       MIN
        HARSH
         ID : d
        CONST : 1
   ID : d
   ADD
```



[c] translated AST (GIPL)

excerpt 9-2 : Demonstration of the translator

9.3 Cost of Using JavaCC

The cost of using JavaCC includes time and resource costs. The time cost also includes learning time, grammar file preparation time, debugging time, and running time. JavaCC/JJTree is relatively quite easy to learn. It takes a Java programmer about one week to be a skilled user. To prepare a GIPL or SIPL grammar file based on the former version takes about only an hour. Finally, the efficency of the generated parser is acceptable. To parse and translate an intermediate Lucid program takes not more than 0.5 second. It is within the common acceptance range.

The resource includes human resource and computer resource. One person is enough to handle all tasks from JavaCC/JJTree learning to integration. The size of JavaCC/JJTree application is 1.72MB and the installation size is 2.21MB. The generated GIPL parser, Indexical SIPL parser and the translator take only 248KB.

9.4 Cost Analysis of Grammar Changes Using JavaCC

Lucid language features evolution. New extended operation set, which is for specific application domain, means that we have to develop new SIPL parser. Actually, here in this thesis, indexical Lucid parser is developed upon GIPL parser by adding those indexical operations into the GIPL grammar file, the indexical SIPL tree translator as well. The previous chapters have shown that the upgrading (grammar change) is easy and not time-consuming. As a mattter of fact, the upgrade from GIPL to Indexical SIPL parser took only a few hours. In addition, we have developed the Indexical SIPL tree translator. This translator can be a template and facilitate the development of other SIPL tree translators.

10. Future Work

So far, we have to write the grammar file manually, although parsers are generated by JavaCC/JJTree. Also, we cannot directly apply the generated parser into GIPSY system. We have to customize part of some source code (java files). For example, we have to do a little change to SimpleNode.java, which is generated by JavaCC application, because the basic SimpleNode attributes and methods cannot fully support abstract syntax tree building.

10.1 Automated Parser Generation

Grammar change is the main feature of Lucid language. If we can, in the future, develop a JavaCC/JJTree grammar file generator, it can definitely boost the invention of new SIPLs. The generator asks user to input the SIPL syntax and generates he grammar file automatically.

10.2 Automated Abstract Syntax Tree Generation

At present, since there is disharmony between Lucid syntax and the demanded structure of abstract syntax tree, we have to add tree building action and parsing action into grammar file to make it up manually. Compared to preparing the syntax part of the grammar file, adding tree building action is much more error-prone. The best solution is to develop an automatic tree building action generator. User inputs the specification of the demanded tree structure to the generator, and then the generator insert the proper tree building actions into Lucid grammar file.

10.3 Automated Abstract Syntax Tree Translation

Finally, for the moment, we have to program the tree translator for each SIPL parser, although we have made a template (Indexical SIPL Lucid translator). However, we can speculate that it is definitely feasible to develop a translator generator, which takes the definition of the extended operation set and outputs the translator for that SIPL.

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] WebGain | Products : JavaCC, http://www.webgain.com/products/java_cc/, 2002.

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

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

[6] Thomas Niemann, A Compact Guide To Lex & Yacc.

[7] Catalog of Compiler Construction Tools, http://catalog.compilertools.net,

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

Appendix I : Lexer/Parser generators evaluation table

ACCENT

1. **Function(s):** Parser generator.
2. **Free download?** Yes. From http://accent.compilertools.net/distribution.html.
3. **Target code:** C.
4. **Grammar:** Does not rely on specific subclasses of context-free grammars. Allows Extended-Backus-Naur-Form and ambiguous grammars (but if there is more than one way to parse the input, the grammar must specify which to select.) It can process the entire class of context-free grammars.
5. **Platform(s):** Unix/Linux.
6. **Developer(s):** German National Research Center for Information Technology. And distributed under GNU GPL.
7. **Comments:** Generally cooperates with Yacc.

AFLEX

1. **Function(s):** Lexer generator.
2. **Free download?** Yes. From http://www1.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html.
3. **Target code:** Ada.
4. **Grammar:** N/A
5. **Platform(s):** Unix.
6. **Developer(s):** Arcadia Project at the University of California.
7. **Comments:** Generally cooperates with Ayacc.

AYACC

1. **Function(s):** Parser generator.
2. **Free download?** Yes. From http://www1.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html.
3. **Target code:** Ada.
4. **Grammar:** LALR.
5. **Platform(s):** Unix.
6. **Developer(s):** Arcadia Project at the University of California.
7. **Comments:** Generally cooperates with Aflex.

ANTLR

1. **Function(s):** Provides a framework for constructing recognizers, compilers, and translators.
2. **Free download?** Yes. Version 2.7.1 from http://www.jguru.com/download/view.jsp?EID=201927
3. **Target code:** Java & C++.
4. **Grammar:** LL(K).
5. **Platform(s):** Unix/Linux, Microsoft 95 or NT.
6. **Developer(s):** jGuru.
7. **Comments:** Very hard to learn and use.

DEPOT4

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** Java.
4. **Grammar:** LL.
5. **Platform(s):** Unix, Windows.
6. **Developer(s):** Jürgen Lampe.
7. **Comments:** For non-expert.

GOBO EIFFEL LEX

1. **Function(s):** Lexer generator.
2. **Free download?** Yes.
3. **Target code:** EIFFEL.
4. **Grammar:** N/A.
5. **Platform(s):** Unix.
6. **Developer(s):**.
7. **Comments:** Cooperate with GOBO EIFFEL Yacc.

GOBO EIFFEL LEX

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** Eiffel.
4. **Grammar:** LALR.
5. **Platform(s):** Unix.
6. **Developer(s):** Eric Bezault.
7. **Comments:** Cooperate with GOBO EIFFEL Lex

HAPPY

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** Haskell.
4. **Grammar:** ELL(1).
5. **Platform(s):** Unix.
6. **Developer(s):** Andy Gill and Simon Marlow.
7. **Comments:** Lexer prepared by the user.

HOLUB

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** C.
4. **Grammar:** ELL(1).
5. **Platform(s):** Windows.
6. **Developer(s):** Allen I. Holub.
7. **Comments:** For teaching purpose.

LALRGEN

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** C.
4. **Grammar:** LALR(1).
5. **Platform(s):** Windows.
6. **Developer(s):** Richard J. LeBlanc.
7. **Comments:** Cooperate with SCANGEN.

LEX

1. **Function(s):** Lexer generator.
2. **Free download?** Yes. While generally it's bundled with Unix system.
3. **Target code:** C.
4. **Grammar:** N/A.
5. **Platform(s):** Unix.
6. **Developer(s):** There are many versions developed by different organizations.
7. **Comments:** Generally cooperates with Yacc.

LISA

1. **Function(s):** Lexer/Parser generator.
2. **Free download?** No.
3. **Target code:** C++.
4. **Grammar:** LL(1).
5. **Platform(s):** Unix/Linux, Windows.
6. **Developer(s):** Marjan Mernik, Nikolaj Korbar, Viljem Zumer.
7. **Comments:** Supports experimentation in automatic language implementation.

LLGEN

1. **Function(s):** Parser generator.
2. **Free download?** No.
3. **Target code:** C.
4. **Grammar:** ELL(1).
5. **Platform(s):** Unix.
6. **Developer(s):** Ceriel Jacobs.
7. **Comments:** Document not available.

MKS LEX

1. **Function(s):** Lexer generator.
2. **Free download?** No.
3. **Target code:** C.
4. **Grammar:** N/A.
5. **Platform(s):** Unix/Linux, Windows.
6. **Developer(s):** MKS.
7. **Comments:** Cooperate with MKS Yacc.

MKS YACC

1. **Function(s):** Parser generator.
2. **Free download?** No.
3. **Target code:** C.
4. **Grammar:** LALR.
5. **Platform(s):** Unix/Linux, Windows.
6. **Developer(s):** MKS.
7. **Comments:** Cooperate with MKS Lex.

MUSKOX

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** C++.
4. **Grammar:** LR(K).
5. **Platform(s):** Linux, Windows.
6. **Developer(s):** Mastersys.
7. **Comments:** N/A.

PCYACC

1. **Function(s):** Language Development Environment.
2. **Free download?** No.
3. **Target code:** C/C++, VB, Java, Delphi.
4. **Grammar:** N/A.
5. **Platform(s):** Unix/Linux, Windows.
6. **Developer(s):** Abraxas Software.
7. **Comments:** Can build Assemblers, Compilers, Interpreters, Browsers, Page Description Languages, Language Translators, Syntax Directed Editors, Language Validators, Natural Language Processors, Expert System Shells, and Query languages.

PRECC

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** C.
4. **Grammar:** LALR.
5. **Platform(s):** Unix/Linux, Windows.
6. **Developer(s):** P.T. Breuer and J.P. Bowen.
7. **Comments:** Cooperate with Lex or Flex.

PROGRAMMAR

1. **Function(s):** Parser generator.
2. **Free download?** No.
3. **Target code:** VB, VC, Delphi.
4. **Grammar:** N/A.
5. **Platform(s):** Windows.
6. **Developer(s):** Norken Technologies.
7. **Comments:** Integrated development environment.

RDP

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** C.
4. **Grammar:** LL(1).
5. **Platform(s):** Unix/Linux.
6. **Developer(s):** Adrian Johnstone.
7. **Comments:** N/A.

RE2C

1. **Function(s):** Lexer generator.
2. **Free download?** Yes.
3. **Target code:** C.
4. **Grammar:** N/A.
5. **Platform(s):** Linux.
6. **Developer(s):** Peter Bumbulis and Brian Young.
7. **Comments:** N/A.

SCANGEN

1. **Function(s):** Lexer generator.
2. **Free download?** Yes.
3. **Target code:** C.
4. **Grammar:** N/A.
5. **Platform(s):** Windows.
6. **Developer(s):** Richard J. LeBlanc.
7. **Comments:** Cooperate with LLGEN or LALR(GEN).

TP LEX

1. **Function(s):** Lexer generator.
2. **Free download?** Yes.
3. **Target code:** Turbo Pascal.
4. **Grammar:** N/A.
5. **Platform(s):** Windows.
6. **Developer(s):** Albert Graef (ag@muwiinfa.geschichte.uni-mainz.de).
7. **Comments:** Lex in Turbo Pascal, cooperate with TP YACC.

TP YACC

1. **Function(s):** Parser generator.
2. **Free download?** Yes.
3. **Target code:** Turbo pascal.
4. **Grammar:** LALR.
5. **Platform(s):** Windows.
6. **Developer(s):** Albert Graef (ag@muwiinfa.geschichte.uni-mainz.de).
7. **Comments:** Yacc in Turbo Pascal, cooperate with TP LEX.

VISUALPARSE++

1. **Function(s):** Visual development environment for writing lexers and parsers.
2. **Free download?** No.
3. **Target code:** C/C++, Java, Delphi, VB.
4. **Grammar:** N/A.
5. **Platform(s):** Unix, Windows and embedded OS.
6. **Developer(s):** Sand Stone.
7. **Comments:** Visual interface.

YACC

1. **Function(s):** Parser generator.
2. **Free download?** Yes. Generally it's bundled with Unix system.
3. **Target code:** C.
4. **Grammar:** LALR.
5. **Platform(s):** Unix.
6. **Developer(s):** There are many versions developed by different organizations.
7. **Comments:** Generally cooperates with Lex.

YACC++

1. **Function(s):** Integrated Lexer/Parser generator.
2. **Free download?** No.
3. **Target code:** C++.
4. **Grammar:** LR(1).
5. **Platform(s):** Unix, Windows.
6. **Developer(s):** Compiler resources.
7. **Comments:** N/A.

Appendix II : Original GIPL grammar file

```
// GIPL Lucid grammar file, file name: Lucid8.jj, which excludes tree-building actions
// This is a stand-alone GIPL Lucid parser grammar file. After compile, the generated
java files
// can compose a Lucid parser, which takes Lucid program from command line.

options {                                   // JavaCC options
  LOOKAHEAD = 2;                            // set look ahead to 2, that is back track is 2
  FORCE_LA_CHECK = true;
}

// Beginning of Compilation part

PARSER_BEGIN(Lucid8)

public class Lucid8 {
  static int errorCount=0;

  public static void main(String args[]) {
    Lucid8 parser;

    if (args.length == 0) {
      System.out.println("Lucid8 Parser Version 0.0:  Reading from standard input . .
.");
      parser = new Lucid8(System.in);
    } else if (args.length == 1) {
      System.out.println("Lucid8 Parser Version 0.0:  Reading from file " + args[0] +
" . . .");
      try {
        parser = new Lucid8(new java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
        System.out.println("Lucid8 Parser Version 0.0:  File " + args[0] + " not
found.");
        return;
      }
    } else {
      System.out.println("Lucid8 Parser Version 0.0:  Usage is one of:");
      System.out.println("         java Lucid < inputfile");
      System.out.println("OR");
      System.out.println("         java Lucid inputfile");
      return;
    }
    try {
      parser.StartP(); /* StartP is the start state */
      System.out.println("Lucid8 Parser Version 0.0:  "+errorCount+"  errors.");
    } catch (ParseException e) {    }
  }
}

PARSER_END(Lucid8)

// End of compilation part


// Lexical specification

SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
```

```
SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < DIMENSION: "dimension" >
| < ELSE: "else" >
| < END: "end" >
| < FI: "fi" >
| < IF: "if" >
| < THEN: "then" >
| < WHERE: "where" >

}

TOKEN : /* OPERATORS */
{
  < ASSIGN: "=" >  /*   ??????   */
  < GT: ">" >
| < LT: "<" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < OR: "or" >
  < AND: "and" >
  < NOT: "not" >
  < PLUS: "+" >
  < MINUS: "-" >
  < STAR: "*" >
  < SLASH: "/" >
  < REM: "%" >
  < ISEOD: "iseod" >  /* unop */
  < AT: "@" >
  < WHEN: "#" >
}

TOKEN : /* LITERALS */
{
  < CONST:
        <INTEGER_LITERAL>
      | <FLOATING_POINT_LITERAL>
  >
|
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
```

```
|
  < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
|
  < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

TOKEN : /* IDENTIFIERS */
{
  < ID: <LETTER> (<LETTER>|<DIGIT>)* >

  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
```

```
  >
}  /* Suppose they are unicode */

TOKEN : /* SEPARATORS */
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >  /* Peter suggests omit ';'*/
| < COMMA: "," >
}

TOKEN :  /* ILLEGAL CHARACTERS */
{
  < ILLEGALCHAR: ["~", "%", "^", "`"] >
}

// End of lexical specifications


// Common token action

TOKEN_MGR_DECLS :
{
  static int count=0;

  static void CommonTokenAction(Token t) {
          System.out.println(t.image);
  }
}

// Syntaxical specifications, including the error report and error recovery

JAVACODE void errorCounting()  // count errors during parsing
{
   errorCount++;
}

void Start2() :  { }
{
 try {
   E()  <EOF>
 } catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
 }
}

void E() : { }
{
  try {

     "if" E() "then" E() "else" E() "fi" E1()
    |  "#" "." E()  E1()
    | Term() E1()
    | Sign() Term() E1()
  }   catch (ParseException e) {
          errorCounting();
         System.out.println(e.toString());
  }
}

void E1() : { }
{
  try {

    [   AddOp() Term() E1()
```

```
    | RelOp() E() E1()
    | "@" "." E() E() E1()
    | "where" Q() "end" E1()
    | "[" E() ("," E())* "]" "(" E() ("," E())* ")" E1()
  )
  }  catch (ParseException e) {
        errorCounting();
        System.out.println(e.toString());
  }
}
void Term() : { }
{
  Factor() Term1()
}
void Term1() : { }
{
   ( MultOp() Term() Term1() )
}
void Factor() : { }
{
    <ID>
  | <CONST>
  | "(" E() ")"
  | "not" "(" E() ")"
}
void Q() : { }
{
  try {
     "dimension" <ID> ("," <ID>)* ";" (Q())*
   | <ID> "=" E() ";" (Q())*
   | <ID> "[" <ID> ("," <ID>)* "]" "(" <ID> ("," <ID>)* ")" "=" E() ";" (Q())*
  }  catch (ParseException e) {
        errorCounting();
        System.out.println(e.toString());
        Token t;
        do {
            t=getNextToken();
        } while (t.kind!=SEMICOLON);
     }
}
void Sign() : { }
{
     "+"
   | "-"
}
void AddOp() : { }
{
     "+"
   | "-"
   | "or"
}
void MultOp() : { }
{
     "*"
   | "/"
   | "%"
   | "and"
}
void RelOp() : { }
{
     "<"
   | ">"
   | ">="
   | "<="
   | "=="
   | "!="
}
```

Appendix III : GIPL grammar file with tree-building actions

```
// This is GIPL Lucid grammar file, including tree-building actions. File name is
Lucid.jjt, so, we
// have to treat this file with JJTree before using JavaCC.
// Generated java files are no longer compose a stand-alone parser, in fact, they are
just part of
// GIPL/SIPL/translator module.

/************** OPTION **********************/
options {
  NODE_DEFAULT_VOID = true;
  LOOKAHEAD = 2;
  FORCE_LA_CHECK = true;
}

/************** COMPILATION PART *************/

PARSER_BEGIN(Parser)

public class Parser {
  static int errorCount=0;

  public void startParse(Parser parser, Facet facet) {
    try {
     SimpleNode n = parser.StartP(); /* StartP is the start state */
     facet.treepass(n);
     facet.display("Lucid Parser Version 1.0:  "+errorCount+"  errors.");
    } catch (ParseException e) {
        facet.display(e.getMessage());
    }
  }
}
PARSER_END(Parser)

/***************** LEXICAL SPECIFICATIONS ****************/

// Note, this part is skipped, because it is exactly the same as the lexical part of
Appendix I.
// Please refer to the same part of Appendix I if interested.


/************** COMMON TOKEN ACTION  *************/

TOKEN_MGR_DECLS :
{
  static int count=0;
  static void CommonTokenAction(Token t) {
      System.out.println(t.image);
  }
}

/***************** SYNTACTICAL SPECIFICATIONS  ****************/

JAVACODE void errorCounting()  // count errors during parsing
{
   errorCount++;
}

JAVACODE SimpleNode harsh(SimpleNode node1, SimpleNode node2)
{
       node1.jjtSetParent(node2);
       node2.jjtAddChild(node1, 0);
       return node2;
}
```

```
JAVACODE SimpleNode sign(SimpleNode node1, SimpleNode node2)
{
        node1.jjtSetParent(node2);
        node2.jjtAddChild(node1, 0);
        return node2;
}

JAVACODE SimpleNode addOp(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        node2.jjtAddChild(node1, 1);
        return node2;
}

JAVACODE SimpleNode multOp(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        node2.jjtAddChild(node1, 1);
        return node2;
}

JAVACODE SimpleNode relOp(SimpleNode node1, SimpleNode node2, SimpleNode node3) //
right, relop, left
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        node2.jjtAddChild(node1, 1);
        return node2;
}

JAVACODE SimpleNode at(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4) // right2 ,1, at, left (node1-4)
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node4.jjtSetParent(node3);
        node3.jjtAddChild(node4, 0);
        node3.jjtAddChild(node2, 1);
        node3.jjtAddChild(node1, 2);
        return node3;
}

JAVACODE SimpleNode where(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        if (node1.children!=null) {
                int i=node1.children.length;
                SimpleNode interNode;
                for (int j=0; j<i; j++) {
                        interNode=(SimpleNode) node1.children[j];
                        interNode.jjtSetParent(node2);
                        node2.jjtAddChild(interNode, j+1);
                }
        }

        return node2;

}

JAVACODE SimpleNode function(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
```

```
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node3.jjtAddChild(node2, 0);
        node3.jjtAddChild(node1, 1);
        return node3;
}

JAVACODE SimpleNode ifClause(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4)  // this is for sipl, gipl doesn't need this function
{                                  // node1-4: if, 1st, 2nd, 3rd paras
        node2.jjtSetParent(node1);
        node3.jjtSetParent(node1);
        node4.jjtSetParent(node1);
        node1.jjtAddChild(node2, 0);
        node1.jjtAddChild(node3, 1);
        node1.jjtAddChild(node4, 2);
        return node1;
}

JAVACODE SimpleNode assign(SimpleNode node1, SimpleNode node2, SimpleNode node3)  //
left, =, right
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node1, 0);
        node2.jjtAddChild(node3, 1);
        return node2;
}


SimpleNode StartP() #START : {  }
{
 try {
   E()  <EOF>
   { return jjtThis; }
 }  catch (ParseException e) {
        errorCounting();
        System.out.println(e.toString());
 }
}

void E() : {  }
{
  try {
    ( "if" E() "then" E() "else" E() "fi" ) #IF E1()
    | ( "#" #HARSH "." E() )
        {   SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // E
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // #
            SimpleNode node=harsh(midNode1, midNode2);
            jjtree.pushNode(node);
        }
        E1()
    | Term() E1()
    | ( ( "+" #POSI | "-" #NEGE ) Term() )
        {   SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // E
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // sign
            SimpleNode node=sign(midNode1, midNode2);
            jjtree.pushNode(node);
        }
      E1()
  }   catch (ParseException e) {
        errorCounting();
        System.out.println(e.toString());
  }
}

void E1() : {  }
{
```

```
    try {
    [    (( "+" #ADD | "-" #MIN | "or" #OR ) Term() )
           { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
             SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // AddOp
             SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
             SimpleNode node=addOp(midNode1, midNode2, midNode3);
             jjtree.pushNode(node);
           }
         E1()
       | (( "<" #LT | ">" #GT | ">=" #GE | "<=" #LE | "==" #EQ | "!=" #NE ) E() )
           { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
             SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // relOp
             SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
             SimpleNode node=addOp(midNode1, midNode2, midNode3);
             jjtree.pushNode(node);
           }
         E1()
       | ( "@" #AT "." E() E() )
           { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right1
             SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // right2
             SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // AT
             SimpleNode midNode4=(SimpleNode) jjtree.popNode(); // left
             SimpleNode node=at(midNode1, midNode2, midNode3, midNode4);
             jjtree.pushNode(node);
           }
         E1()
       | ("where" #WHERE  Q() #Qlist "end" )
           { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
             SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // where
             SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
             SimpleNode node=where(midNode1, midNode2, midNode3);
             jjtree.pushNode(node);
           }
         E1()
       | ( Tail() )
           { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // paras
             SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
             SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // function name
             SimpleNode node=function(midNode1, midNode2, midNode3);
             jjtree.pushNode(node);
           }
         E1()
    ]
    }  catch (ParseException e) {
          errorCounting();
          System.out.println(e.toString());
    }
}

void Tail() :  { }  // This Tail and the QTail later are not to be considered so far
{
  try {
     "[" ( E() ("," E())* ) #DIM "]" "(" ( E() ("," E())* ) #PARAS ")"
  }  catch (ParseException e) {
        errorCounting();
        System.out.println(e.toString());
  }
}

void Term() : { }
{
  Factor() Term1()
}

void Term1() : { }
{
   [( ( "*" #TIMES | "/" #DIV | "%" #MOD | "and" #AND )  Term() )
```

```
        { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
          SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // multOp
          SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
          SimpleNode node=multOp(midNode1, midNode2, midNode3);
          jjtree.pushNode(node);
        }
   Term1()  ]

}

void Factor() : { }
{
    ID()
  | INTEGER()
  | FLOAT()
  | "(" E() ")"
  | ("not" "(" E() ")") #NOT
}

void Q() : { }
{
  try {
    ( ( "dimension"  ID() ("," ID())* ";" ) #DIMENSION) (Q())*
  | ((ID() "=" E() ";"  ) #ASSIGN) ((Q())* )
  | ((ID() QTail() "=" E() ";" ) #ASSIGN) ((Q())*)
  }  catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
       Token t;
       do {
           t=getNextToken();
       } while (t.kind!=SEMICOLON);
    }
}

void QTail() : { }
{
  try {
   "[" ( ID() ("," ID())* ) #DIM "]" "("  ( ID() ("," ID())* ) #PARAS ")"
                              { Node midNode1=jjtree.popNode(); // paras
                                Node midNode2=jjtree.popNode(); // dim
                                Node midNode3=jjtree.popNode(); // function name
                                midNode1.jjtSetParent(midNode3);
                                midNode2.jjtSetParent(midNode3);
                                midNode3.jjtAddChild(midNode2, 0);
                                midNode3.jjtAddChild(midNode1, 1);
                                jjtree.pushNode(midNode3);
                              }
  }  catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
            Token t;
          do {
          t=getNextToken();
        } while (t.kind!=SEMICOLON);
  }
}

void ID() #ID:    // Note: Because of the scope issue, we have to put #ID at this
position so that jjtThis can work properly
{
  Token t;
}
{
  t = <ID>      // for example, we cannot put #ID over here, or jjtThis will not refer
to Node ID.
  { jjtThis.setImage(t.image); }
}
```

```
void INTEGER() #CONST:
{
  Token t;
}
{
      t = <INTEGER_LITERAL>  { jjtThis.setType(0); jjtThis.setImage(t.image); }
}

void FLOAT() #CONST:
{
  Token t;
}
{
      t = <FLOATING_POINT_LITERAL>  { jjtThis.setType(1); jjtThis.setImage(t.image); }
}
```

Appendix IV : Indexical Lucid SIPL grammar file

```
// This is Indexical SIPL Lucid grammar file, including tree-building actions.
// File name is sipl1.jjt, so, we have to treat this file with JJTree before using
JavaCC.
// Generated java files are no longer compose a stand-alone parser, in fact, they are
part of
// GIPL/SIPL/translator module.

/************** OPTION **********************/
options {
  NODE_DEFAULT_VOID = true;
  LOOKAHEAD = 2;
  FORCE_LA_CHECK = true;
}

/************** COMPILATION PART **************/

PARSER_BEGIN(SIPL)

public class SIPL {
  static int errorCount=0;
  public void startParse(SIPL sipl, Facet facet) {
    try {
     SimpleNode n = sipl.StartP(); /* StartP is the start state */
     facet.treepass(n);
     facet.display("Lucid Parser Version 0.0:  "+errorCount+"  errors.");
    } catch (ParseException e) {
        facet.display(e.getMessage());
    }
  }
}

PARSER_END(SIPL)

/***************** LEXICAL SPECIFICATION ******************/

SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < DIMENSION: "dimension" >
| < ELSE: "else" >
| < END: "end" >
| < FI: "fi" >
| < IF: "if" >
| < THEN: "then" >
| < WHERE: "where" >
}

TOKEN : /* OPERATORS */
{
```

```
  < ASSIGN: "=" >  /*   ??????   */
| < GT: ">" >
| < LT: "<" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < OR: "or" >
| < AND: "and" >
| < NOT: "not" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < REM: "%" >
| < ASA: "asa" >            // extended indexical SIPL operations
| < FBY: "fby" >
| < UPON: "upon" >
  < WVR: "wvr" >
| < FIRST: "first" >
| < NEXT: "next" >
  < PREV: "prev" >
| < ISEOD: "iseod" >   // end of extended indexical SIPL operations
| < AT: "@" >
  < WHEN: "#" >
}

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >

  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >

  < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >

  < #OCTAL_LITERAL: "0" (["0"-"7"])* >

  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >

  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >

  < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >

  < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
```

```
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
        )*
        "\""
    >
}

TOKEN : /* IDENTIFIERS */
{
  < ID: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
    >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
    >
}  /* Suppose they are unicode */

TOKEN : /* SEPARATORS */
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >  /* Peter suggests omit ';'*/
| < COMMA: "," >
}

TOKEN :  /* ILLEGAL CHARACTERS */
{
  < ILLEGALCHAR: ["~", "%", "^", "`"] >
}

/*************** COMMON TOKEN ACTION  ***************/
```

```
TOKEN_MGR_DECLS :
{
  static int count=0;
  static void CommonTokenAction(Token t) {
      System.out.println(t.image);
  }
}



/****************** SYNTACTICAL SPECIFICATIONS ****************/

JAVACODE void errorCounting()  // count errors during parsing
{
   errorCount++;
}

JAVACODE SimpleNode harsh(SimpleNode node1, SimpleNode node2)
{
        node1.jjtSetParent(node2);
        node2.jjtAddChild(node1, 0);
        return node2;
}

JAVACODE SimpleNode sign(SimpleNode node1, SimpleNode node2)
{
        node1.jjtSetParent(node2);
        node2.jjtAddChild(node1, 0);
        return node2;
}

JAVACODE SimpleNode addOp(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        node2.jjtAddChild(node1, 1);
        return node2;
}

JAVACODE SimpleNode multOp(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        node2.jjtAddChild(node1, 1);
        return node2;
}

JAVACODE SimpleNode relOp(SimpleNode node1, SimpleNode node2, SimpleNode node3) //
right, relop, left
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        node2.jjtAddChild(node1, 1);
        return node2;
}

JAVACODE SimpleNode at(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4) // right2 ,1, at, left (node1-4)
{

        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node4.jjtSetParent(node3);
        node3.jjtAddChild(node4, 0);
```

```
        node3.jjtAddChild(node2, 1);
        node3.jjtAddChild(node1, 2);
        return node3;
}

JAVACODE SimpleNode where(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node3, 0);
        if (node1.children!=null) {
                int i=node1.children.length;
                SimpleNode interNode;
                for (int j=0; j<i; j++) {
                        interNode=(SimpleNode) node1.children[j];
                        interNode.jjtSetParent(node2);
                        node2.jjtAddChild(interNode, j+1);
                }
        }
        return node2;
}

JAVACODE SimpleNode function(SimpleNode node1, SimpleNode node2, SimpleNode node3)
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node3.jjtAddChild(node2, 0);
        node3.jjtAddChild(node1, 1);
        return node3;
}

JAVACODE SimpleNode first(SimpleNode node1, SimpleNode node2, SimpleNode node3) // id,
dim, first
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node3.jjtAddChild(node1, 0);
        node3.jjtAddChild(node2, 1);
        return node3;
}

JAVACODE SimpleNode next(SimpleNode node1, SimpleNode node2, SimpleNode node3)   //
node1:id, node2:dim, next
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node3.jjtAddChild(node1, 0);
        node3.jjtAddChild(node2, 1);
        return node3;
}
JAVACODE SimpleNode prev(SimpleNode node1, SimpleNode node2, SimpleNode node3)   //
node1:id, node2:dim
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node3.jjtAddChild(node1, 0);
        node3.jjtAddChild(node2, 1);
        return node3;
}

JAVACODE SimpleNode ifClause(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4)  // this is for sipl, gipl doesn't need this function
{                                  // node1-4: if, 1st, 2nd, 3rd paras
        node2.jjtSetParent(node1);
        node3.jjtSetParent(node1);
        node4.jjtSetParent(node1);
        node1.jjtAddChild(node2, 0);
        node1.jjtAddChild(node3, 1);
        node1.jjtAddChild(node4, 2);
```

```
        return node1;
}

JAVACODE SimpleNode fby(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4)  // node1:id-right, node2:dim, node3:fby, node4:id-left
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node4.jjtSetParent(node3);
        node3.jjtAddChild(node1, 0);
        node3.jjtAddChild(node2, 1);
        node3.jjtAddChild(node4, 2);
        return node3;
}

JAVACODE SimpleNode makeQlist(SimpleNode child, SimpleNode parent)
{
        child.jjtSetParent(parent);
        if (parent.children!=null)
            parent.jjtAddChild(child, parent.children.length);
        else
            parent.jjtAddChild(child, 0);
        return parent;
}

JAVACODE SimpleNode assign(SimpleNode node1, SimpleNode node2, SimpleNode node3)  //
left, =, right
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node1, 0);
        node2.jjtAddChild(node3, 1);
        return node2;
}
JAVACODE SimpleNode wvr(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4)  // node1:id-right, node2:dim, node4:id-left
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node4.jjtSetParent(node3);
        node3.jjtAddChild(node1, 0);
        node3.jjtAddChild(node2, 1);
        node3.jjtAddChild(node4, 2);
        return node3;
}

JAVACODE SimpleNode aaa(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4)  // node1:id-right, node2:dim, node4:id-left
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node4.jjtSetParent(node3);
        node3.jjtAddChild(node1, 0);
        node3.jjtAddChild(node2, 1);
        node3.jjtAddChild(node4, 2);
        return node3;
}

JAVACODE SimpleNode upon(SimpleNode node1, SimpleNode node2, SimpleNode node3,
SimpleNode node4)  // node1:id-right, node2:dim, node4:id-left
{
        node1.jjtSetParent(node3);
        node2.jjtSetParent(node3);
        node4.jjtSetParent(node3);
        node3.jjtAddChild(node1, 0);
        node3.jjtAddChild(node2, 1);
        node3.jjtAddChild(node4, 2);
        return node3;
```

```
}

SimpleNode StartP() #START : { }
{
 try {
   E()  <EOF>
   { return jjtThis; }
 } catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
 }
}

void E() : { }
{
  try {
    ( "if" E() "then" E() "else" E() "fi" ) #IF E1()
    | ( "#" #HARSH "." E() )
       {   SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // E
           SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // #
           SimpleNode node=harsh(midNode1, midNode2);
           jjtree.pushNode(node);
       }
       E1()
    | Term() E1()
    | ( ( "+" #POSI | "-" #NEGE ) Term() )
       {   SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // E
           SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // sign
           SimpleNode node=sign(midNode1, midNode2);
           jjtree.pushNode(node);
       }
      E1()
  } catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
  }
}

void E1() : { }
{
  try {
    (( "+" #ADD | "-" #MIN | "or" #OR ) Term() )
       { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
         SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // AddOp
         SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
         SimpleNode node=addOp(midNode1, midNode2, midNode3);
         jjtree.pushNode(node);
       }
     E1()
   | (( "<" #LT | ">" #GT | ">=" #GE | "<=" #LE | "==" #EQ | "!=" #NE ) E() )
       { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
         SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // relOp
         SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
         SimpleNode node=addOp(midNode1, midNode2, midNode3);
         jjtree.pushNode(node);
       }
     E1()
   | ( "@" #AT "." E() E() )
       { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right1
         SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // right2
         SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // AT
         SimpleNode midNode4=(SimpleNode) jjtree.popNode(); // left
         SimpleNode node=at(midNode1, midNode2, midNode3, midNode4);
         jjtree.pushNode(node);
       }
     E1()
   | ("where" #WHERE  Q() #Qlist "end" )
       { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
```

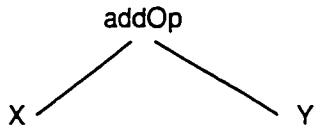
```
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // where
            SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
            SimpleNode node=where(midNode1, midNode2, midNode3);
            jjtree.pushNode(node);
          }
        E1()
      | ( Tail() )
          { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // paras
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
            SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // function name
            SimpleNode node=function(midNode1, midNode2, midNode3);
            jjtree.pushNode(node);
          }
        E1()
      | ( ("fby" #FBY) ( "[" E() ("," E())* "]" ) #DIM "(" E() ")"
        | ( "fby" #FBY ) "." E() E() )
          { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // id-right
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
            SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // fby
            SimpleNode midNode4=(SimpleNode) jjtree.popNode(); // id-left
            SimpleNode node=fby(midNode1, midNode2, midNode3, midNode4);
            jjtree.pushNode(node);
          }
        E1()
      | ( ("wvr" #WVR) ( "[" E() ("," E())* "]" ) #DIM "(" E() ")"
          ( "wvr" #WVR ) "." E() E() )
          { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // id-right
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
            SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // wvr
            SimpleNode midNode4=(SimpleNode) jjtree.popNode(); // id-left
            SimpleNode node=wvr(midNode1, midNode2, midNode3, midNode4);
            jjtree.pushNode(node);
          }
        E1()
      | ( ("aaa" #AAA) ( "[" E() ("," E())* "]" ) #DIM "(" E() ")"
        | ( "aaa" #AAA ) "." E() E() )
          { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // id-right
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
            SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // aaa
            SimpleNode midNode4=(SimpleNode) jjtree.popNode(); // id-left
            SimpleNode node=aaa(midNode1, midNode2, midNode3, midNode4);
            jjtree.pushNode(node);
          }
        E1()
      | ( ("upon" #UPON) ( "[" E() ("," E())* "]" ) #DIM "(" E() ")"
        | ( "upon" #UPON ) "." E() E() )
          { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // id-right
            SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
            SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // upon
            SimpleNode midNode4=(SimpleNode) jjtree.popNode(); // id-left
            SimpleNode node=upon(midNode1, midNode2, midNode3, midNode4);
            jjtree.pushNode(node);
          }
        E1()
  ]
  } catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
  }
}

void Tail() : { } // This Tail and the QTail later are not to be considered so far
{
  try {
     "[" ( E() ("," E())* ) #DIM "]" "(" ( E() ("," E())* ) #PARAS ")"
   } catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
```

```
  }
}

void Term() : { }
{
  Factor() Term1()
}

void Term1() : { }
{
   [( ( "*" #TIMES | "/" #DIV | "%" #MOD | "and" #AND )  Term() )
        { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // right
          SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // multOp
          SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // left
          SimpleNode node=multOp(midNode1, midNode2, midNode3);
          jjtree.pushNode(node);
        }
   Term1()  ]
 }

void Factor() : { }
{
    ID()
  | INTEGER()
  | FLOAT()
  | "(" E() ")"
  | ("not" "(" E() ")") #NOT
  | Unary()
}

void Unary() : { }
{
   ( ("first" #FIRST) ( "[" E() ("," E())* "]" ) #DIM "(" E() ")"
     | ( "first" #FIRST ) "." E() E() )
        { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // id
          SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
          SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // first
          SimpleNode node=first(midNode1, midNode2, midNode3);
          jjtree.pushNode(node);
        }
  | ( ( "next" #NEXT ) ("[" E() ("," E())* "]") #DIM "(" E() ")"
    | ("next" #NEXT ) "." E() E() )

        { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // id
          SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
          SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // next
          SimpleNode node=next(midNode1, midNode2, midNode3);
          jjtree.pushNode(node);
        }
  | ( ( "prev" #PREV ) ( "[" E() ("," E())* "]" ) #DIM "(" E() ")"
    | ("prev" #PREV ) "." E() E() )
        { SimpleNode midNode1=(SimpleNode) jjtree.popNode(); // id
          SimpleNode midNode2=(SimpleNode) jjtree.popNode(); // dim
          SimpleNode midNode3=(SimpleNode) jjtree.popNode(); // prev
          SimpleNode node=prev(midNode1, midNode2, midNode3);
          jjtree.pushNode(node);
        }
}

void Q() : { }
{
  try {
    ( ( "dimension" ID() ("," ID())* ";" ) #DIMENSION) (Q())*
  | ((ID() "=" E() ";"  ) #ASSIGN) ((Q())* )
  | ((ID() QTail() "=" E() ";" ) #ASSIGN) ((Q())*)
  } catch (ParseException e) {
      rrorCounting();
      System.out.println(e.toString());
```

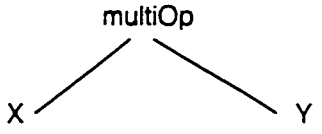
```
        Token t;
        do {
              t=getNextToken();
           } while (t.kind!=SEMICOLON);
  }
}

void QTail() : { }
{
  try {
   "[" ( ID() ("," ID())* ) #DIM "]" "(" ( ID() ("," ID())* ) #PARAS ")"
                                { Node midNode1=jjtree.popNode(); // paras
                                  Node midNode2=jjtree.popNode(); // dim
                                  Node midNode3=jjtree.popNode(); // function name
                                  midNode1.jjtSetParent(midNode3);
                                  midNode2.jjtSetParent(midNode3);
                                  midNode3.jjtAddChild(midNode2, 0);
                                  midNode3.jjtAddChild(midNode1, 1);
                                  jjtree.pushNode(midNode3);
                                }
  } catch (ParseException e) {
       errorCounting();
       System.out.println(e.toString());
       Token t;
        do {
             t=getNextToken();
          } while (t.kind!=SEMICOLON);
  }
}

void ID() #ID:    // Note: Because of the scope issue, we have to put #ID at this
position so that jjtThis can work properly
{
  Token t;
}
{
  t = <ID>        // for example, we cannot put #ID over here, or jjtThis will not refer
to Node ID.
  { jjtThis.setImage(t.image); }
}

void INTEGER() #CONST:
{
  Token t;
}
{
     t = <INTEGER_LITERAL>  { jjtThis.setType(0); jjtThis.setImage(t.image); }
}

void FLOAT() #CONST:
{
  Token t;
}
{
     t = <FLOATING_POINT_LITERAL>  { jjtThis.setType(1); jjtThis.setImage(t.image); }
}
```

Appendix V : Indexical Lucid AST structures

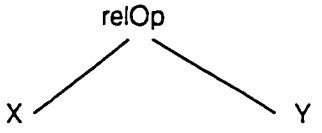
1. X **addOp** Y



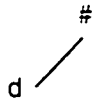
2. X **multiOp** Y



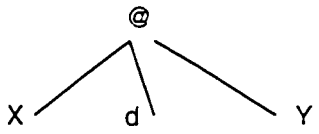
3. X **relOp** Y



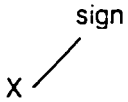
4. **#.** d



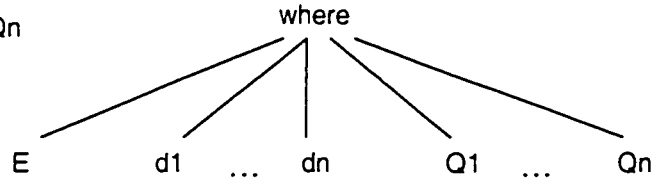
5. X **@.** d Y



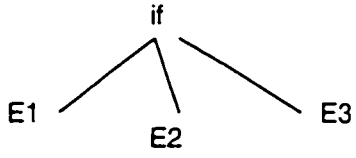
6. **sign** X



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



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



Appendix VI : Indexical Lucid operations definitions

The Indexical SIPL Lucid extended operation set includes the following operations:

Uniary operations: *first*, *next*, and *prev*.
Binary operations: *fby*, *wvr*, *asa* and *upon*.

And the following are their definitions:

```
first.d X = X @.d 0;
next.d X = X @.d (#.d+1);
prev.d X = X @.d (#.d-1);

X fby.d Y = if #.d <= 0 then X else Y @.d (#.d-1);
X wvr.d Y = X @.d T
          where
              T = U fby.d U @.d (T+1);
              U = if Y then #.d else next.d U;
          end;
X asa.d Y = first.d (X wvr.d Y);
X upon.d Y = X @.d W
          where
              W = 0 fby.d if Y then (W+1) else W;
          end;
```

Appendix VII : Source code of customized SimpleNode.java

```
/* Generated By:JJTree: Do not edit this line. SimpleNode.java */

public class SimpleNode implements Node {
  protected Node parent;
  protected Node[] children;
  protected int id;
  protected Parser parser;
  protected SIPL1 sipl1;
  protected String image;  // customized attribute record the image of id, if it was
an id
  protected int type;      // customized attribute record the type of const, if it was
a const

  public SimpleNode() {   // customized constructor
  }

  public SimpleNode(int i) {
    id = i;
  }

  public void jjtOpen() {
  }

  public void jjtClose() {
  }

  public void jjtSetParent(Node n) { parent = n; }
  public Node jjtGetParent() { return parent; }

  public void jjtAddChild(Node n, int i) {
    if (children == null) {
      children = new Node[i + 1];
    } else if (i >= children.length) {
      Node c[] = new Node[i + 1];
      System.arraycopy(children, 0, c, 0, children.length);
      children = c;
    }
    children[i] = n;
  }

  public Node jjtGetChild(int i) {
    return children[i];
  }

  public int jjtGetNumChildren() {
    return (children == null) ? 0 : children.length;
  }

  public String toString() { // customized method
     if (Lucid.parserType==0) {
        if (image != null) return ParserTreeConstants.jjtNodeName[id] + " : " + image;
        else return ParserTreeConstants.jjtNodeName[id];
     }
     {
        if (image != null) return SIPL1TreeConstants.jjtNodeName[id] + " : " + image;
        else return SIPL1TreeConstants.jjtNodeName[id];
     }
  }
  public String toString(String prefix) { return prefix + toString(); }

  public void dump(String prefix) {
    System.out.println(toString(prefix));
    if (children != null) {
      for (int i = 0; i < children.length; ++i) {
```

```
        SimpleNode n = (SimpleNode)children[i];
        if (n != null) {
          n.dump(prefix + " ");
        }
      }
    }
  }

  public void setImage(String s) { // customized method
     image=s;
  }

  public void setType(int t) {  // customized method
     type=t;
  }

 public void copyNode(SimpleNode node) {  // customized method
       parent=node.parent;
       id=node.id;
       parser=node.parser;
       image=node.image;
       type=node.type;

       if (node.children!=null) {
         int i=node.children.length;
           children=new SimpleNode[i];
         for (int j=0; j<i; j++)
           children[j]=node.children[j];
       }
  }
}
```

Appendix VIII : Source code of Translator.java

```
public class Translator {
       public Translator() { }
       void translate(SimpleNode simpleNode) {
         if (simpleNode.children !=null) {
            for (int i = 0; i < simpleNode.children.length; i++) {
                SimpleNode node = (SimpleNode) simpleNode.children[i];
                switch (node.id) {
                  case 21:  // asa
                      SimpleNode n4 = (SimpleNode) simpleNode.children[i];
                      SimpleNode node4 = new SimpleNode();
                      node4    =    asa((SimpleNode)    n4.children[0],    (SimpleNode)
n4.children[1], (SimpleNode) n4.children[2]);
                      simpleNode.jjtAddChild(node4, i);
                      node4.jjtSetParent(simpleNode);
                      translate(node4);
                      break;

                   case 18:  // fby
                      SimpleNode n5 = (SimpleNode) simpleNode.children[i];
                      SimpleNode node5 = new SimpleNode();
                      node5    =    fby((SimpleNode)    n5.children[0],    (SimpleNode)
n5.children[1], (SimpleNode) n5.children[2]);
                      simpleNode.jjtAddChild(node5, i);
                      node5.jjtSetParent(simpleNode);
                      translate(node5);
                      break;

                   case 22:  // upon
                      SimpleNode n6 = (SimpleNode) simpleNode.children[i];
                      SimpleNode node6 = new SimpleNode();
                      node6    =    upon((SimpleNode)    n6.children[0],    (SimpleNode)
n6.children[1], (SimpleNode) n6.children[2]);
                      simpleNode.jjtAddChild(node6, i);
                      node6.jjtSetParent(simpleNode);
                      translate(node6);
                      break;

                   case 20:  // wvr
                      SimpleNode n7 = (SimpleNode) simpleNode.children[i];
                      SimpleNode node7 = new SimpleNode();
                      node7    =    wvr((SimpleNode)    n7.children[0],    (SimpleNode)
n7.children[1], (SimpleNode) n7.children[2]);
                      simpleNode.jjtAddChild(node7, i);
                      node7.jjtSetParent(simpleNode);
                      translate(node7);
                      break;

                   case 29:  // first
                      SimpleNode n1 = (SimpleNode) simpleNode.children[i];
                      SimpleNode node1 = new SimpleNode();
                      node1    =    first((SimpleNode)    n1.children[0],    (SimpleNode)
n1.children[1]);
                      simpleNode.jjtAddChild(node1, i);
                      node1.jjtSetParent(simpleNode);
                      translate(node1);
                      break;

                   case 30:  // next
                      SimpleNode n2 = (SimpleNode) simpleNode.children[i];
                      SimpleNode node2 = new SimpleNode();
                      node2    =    next((SimpleNode)    n2.children[0],    (SimpleNode)
n2.children[1]);
                      simpleNode.jjtAddChild(node2, i);
                      node2.jjtSetParent(simpleNode);
```

```
                    translate(node2);
                    break;

                case 31:  // prev
                    SimpleNode n3 = (SimpleNode) simpleNode.children[1];
                    SimpleNode node3 = new SimpleNode();
                    node3    =    prev((SimpleNode)    n3.children[0],    (SimpleNode)
n3.children[1]);
                    simpleNode.jjtAddChild(node3, 1);
                    node3.jjtSetParent(simpleNode);
                    translate(node3);
                    break;

                default:
                    translate(node);
            }
        }
      }
    }

SimpleNode first(SimpleNode node1, SimpleNode node2) // id, dim
{
    SimpleNode at=new SimpleNode(15);      // create a new 'at' node
    SimpleNode const0=new SimpleNode(35);  // create a new '0' node (integer)
    const0.setImage("0");
    const0.setType(0);                     // set value of '0' node

    node1.jjtSetParent(at);
    node2.jjtSetParent(at);
    const0.jjtSetParent(at);
    at.jjtAddChild(node1, 0);
    at.jjtAddChild(node2, 1);
    at.jjtAddChild(const0, 2);
    return at;
}

SimpleNode next(SimpleNode node1, SimpleNode node2)  // node1:id, node2:dim
{
    SimpleNode sn = new SimpleNode();
    SimpleNode harsh=new SimpleNode(3);    // create a new 'harsh' node
    SimpleNode add=new SimpleNode(6);      // create a new 'add' node
    SimpleNode const1=new SimpleNode(35); // create a new '1' node
    SimpleNode dim1=new SimpleNode();
    dim1.copyNode(node2);        // make a copy of node2 (dimension)
    const1.setImage("1");
    const1.setType(0);                    // set value of '1' node
    SimpleNode at=new SimpleNode(15);      // create a new 'at' node
         try {
        sn = SIPL.at(SIPL.addOp(const1, add, SIPL.harsh(dim1, harsh)), node2, at,
node1);
         } catch (ParseException e) { }
    return sn;
}

SimpleNode prev(SimpleNode node1, SimpleNode node2)  // node1:id, node2:dim
{
    SimpleNode sn = new SimpleNode();
    SimpleNode harsh=new SimpleNode(3);    // create a new 'harsh' node
    SimpleNode min=new SimpleNode(7);      // create a new 'minus' node
    SimpleNode const1=new SimpleNode(35); // create a new '1' node
    SimpleNode dim1=new SimpleNode();
    dim1.copyNode(node2);        // make a copy of node2 (dimension)
    const1.setImage("1");
    const1.setType(0);                    // set value of '1' node
    SimpleNode at=new SimpleNode(15);      // create a new 'at' node
    try {
        sn = SIPL.at(SIPL.addOp(const1, min, SIPL.harsh(dim1, harsh)), node2, at,
node1);
```

```
        } catch (ParseException e) { }
    return sn;
}

SimpleNode ifClause(SimpleNode node1, SimpleNode node2, SimpleNode node3, SimpleNode
node4)  // this is for sipl, gipl doesn't need this function
{                                   // node1-4: if, 1st, 2nd, 3rd paras
        node2.jjtSetParent(node1);
        node3.jjtSetParent(node1);
        node4.jjtSetParent(node1);
        node1.jjtAddChild(node2, 0);
        node1.jjtAddChild(node3, 1);
        node1.jjtAddChild(node4, 2);
        return node1;
}

SimpleNode fby(SimpleNode node1, SimpleNode node2, SimpleNode node3)   // node1:id-
right, node2:dim, node3:id-left
{
        SimpleNode sn = new SimpleNode();
        //      X fby.d Y = if #.d <= 0 then X else Y @.d (#.d-1)
        SimpleNode ifNode=new SimpleNode(2);   // create a new 'if-clause' node
        SimpleNode harsh1=new SimpleNode(3);   // create a new 'harsh' node
        SimpleNode harsh2=new SimpleNode(3);   // create another new 'harsh' node
        SimpleNode at=new SimpleNode(15);   // create a new 'at' node
        SimpleNode min=new SimpleNode(7);     // create a new 'minus' node
        SimpleNode lessEqual=new SimpleNode(12);   // create another new '<=' node
        SimpleNode const0=new SimpleNode(35); // create a new '0' node
        SimpleNode const1=new SimpleNode(35); // create a new '1' node
        SimpleNode dim1=new SimpleNode();
        dim1.copyNode(node2);       // make a copy of node2 (dimension)
        SimpleNode dim2=new SimpleNode();
        dim2.copyNode(node2);       //  make another copy of node2 (dimension)
        const0.setImage("0");
        const0.setType(0);                    // set value of '0' node
        const1.setImage("1");
        const1.setType(0);                    // set value of '1' node
        try {
            sn      =       ifClause(ifNode,       SIPL.relOp(const0,       lessEqual,
SIPL.harsh(node2,harsh1)),  node3,  SIPL.at(SIPL.addOp(const1,  min,  SIPL.harsh(dim2,
harsh2)), dim1, at, node1));
        } catch (ParseException e) { }
    return sn;
}

SimpleNode makeQlist(SimpleNode child, SimpleNode parent)
{
        child.jjtSetParent(parent);
        if (parent.children!=null)
            parent.jjtAddChild(child, parent.children.length);
        else     parent.jjtAddChild(child, 0);
        return parent;
}

SimpleNode assign(SimpleNode node1, SimpleNode node2, SimpleNode node3)  // left, =,
right
{
        node1.jjtSetParent(node2);
        node3.jjtSetParent(node2);
        node2.jjtAddChild(node1, 0);
        node2.jjtAddChild(node3, 1);
        return node2;
}

SimpleNode wvr(SimpleNode node1, SimpleNode node2, SimpleNode node3)   // node1:id-
right, node2:dim, node3:id-left
{
        SimpleNode sn = new SimpleNode();
```

```
        /**************************************
         X wvr.d Y = X @.d T
            where
                T = U fby.d U @.d (T+1);
                U = if Y then #.d else next.d U;
            end
        **************************************/
        SimpleNode where=new SimpleNode(16);   // create a 'where' node
        SimpleNode ifNode=new SimpleNode(2);   // create an 'if-clause' node
        SimpleNode harsh1=new SimpleNode(3);   // create a 'harsh' node
        SimpleNode at1=new SimpleNode(15);
        SimpleNode at2=new SimpleNode(15);   // create 2 'at' nodes
        SimpleNode add=new SimpleNode(6);     // create an 'add' node
        SimpleNode const1=new SimpleNode(35); // create a new '1' node
        const1.setImage("1");
        const1.setType(0);                  // set value of '1' node

        SimpleNode dim1=new SimpleNode();
        dim1.copyNode(node2);
        SimpleNode dim2=new SimpleNode();
        dim2.copyNode(node2);
        SimpleNode dim3=new SimpleNode();
        dim3.copyNode(node2);
        SimpleNode dim4=new SimpleNode();
        dim4.copyNode(node2);        // make 4 copies of node2 (dimension)
        SimpleNode t1=new SimpleNode(34);   // create 3 intermediate id nodes: T
        t1.setImage("t");
        SimpleNode t2=new SimpleNode();
        t2.copyNode(t1);
        SimpleNode t3=new SimpleNode();
        t3.copyNode(t1);
        SimpleNode u1=new SimpleNode(34);   // create 4 intermediate id nodes: U
        u1.setImage("u");
        SimpleNode u2=new SimpleNode();
        u2.copyNode(u1);
        SimpleNode u3=new SimpleNode();
        u3.copyNode(u1);
        SimpleNode u4=new SimpleNode();
        u4.copyNode(u1);
        SimpleNode assign1=new SimpleNode(33);   // create 2 'assign' nodes
        SimpleNode assign2=new SimpleNode(33);
        SimpleNode qlist=new SimpleNode(17);
         try {
                makeQlist(assign(t2,  assign1,  SIPL.at(SIPL.addOp(const1,  add,  t3),
dim2, at2, fby(u2, dim1, u1))), qlist);
                makeQlist(assign(u3,       assign2,       ifClause(ifNode,       node1,
SIPL.harsh(dim3,harsh1), next(u4, dim4))), qlist);
        // need 'QList' reference, to child T and U
               sn = SIPL.where(qlist, where, SIPL.at(t1, node2, at1, node3));
         } catch (ParseException e) { }
        return sn;
}

SimpleNode asa(SimpleNode node1, SimpleNode node2, SimpleNode node3)   // node1:id-
right, node2:dim, node3:id-left
{
        /**************************************
          X aaa.d Y = first.d (X wvr.d Y);
        **************************************/
        SimpleNode dim1=new SimpleNode();
        dim1.copyNode(node2);        // make a copy of node2 (dimension)
        return first(wvr(node1, dim1, node3), node2);
}

SimpleNode upon(SimpleNode node1, SimpleNode node2, SimpleNode node3)   // node1:id-
right, node2:dim, node3:id-left
{
        SimpleNode sn = new SimpleNode();
```

```
        /*****************************************
          X upon.d Y = X @.d W
             where
               W = 0 fby.d if Y then (W+1) else W;
             end;
        *****************************************/
        SimpleNode where=new SimpleNode(16);   // create a 'where' node
        SimpleNode ifNode=new SimpleNode(2);   // create a new 'if' node
        SimpleNode at=new SimpleNode(15);   // create a new 'at' node
        SimpleNode const0=new SimpleNode(35); // create a new '0' node
        SimpleNode const1=new SimpleNode(35); // create a new '1' node
        const0.setImage("0");
        const0.setType(0);                    // set value of '0' node
        const1.setImage("1");
        const1.setType(0);                   // set value of '1' node
        SimpleNode add=new SimpleNode(6);     // create a new 'add' node
        SimpleNode assign=new SimpleNode(33);     // create a new 'assign' node
        SimpleNode dim1=new SimpleNode();
        dim1.copyNode(node2);        // make a copy of node2 (dimension)
        SimpleNode w1=new SimpleNode(34);   // create 4 intermediate id nodes: W
        w1.setImage("w");
        SimpleNode w2=new SimpleNode();
        w2.copyNode(w1);
        SimpleNode w3=new SimpleNode();
        w3.copyNode(w1);
        SimpleNode w4=new SimpleNode();
        w4.copyNode(w1);                      // create 4 intermediate id nodes: W
        SimpleNode qlist=new SimpleNode(17);
        try {
               makeQlist(fby(ifClause(ifNode, node1, SIPL.addOp(const1, add, w2), w3),
dim1, assign(w4, assign, const1)), qlist);
               sn = SIPL.where(qlist, where, SIPL.at(w1, node1, at, node1));
        } catch (ParseException e) { }
        return sn;
    }
}
```

Appendix IX : Source code of the façade interface class

```
public class Facet {   // file name is facet.java
        SimpleNode simpleNode;  // all kind of trees are referred by this node only
        Parser parser;
        SIPL sipl;
        String filename;
        String message;
        Translator translator;

        void parsing(String filename) {
          if (Lucid.parserType == 0) {
            display("Lucid Parser Version 3.0 (GIPL):  Reading from file "+filename+"
. . .");
            try {
                parser = new Parser(new java.io.FileInputStream(filename));
            } catch (java.io.FileNotFoundException e) {
                        display("Lucid Parser Version 3.0 (GIPL):  File " + filename +
" not found.");
                        return;
            }
            parser.startParse(parser, this);
          }
          else {
            display("Lucid Parser Version 3.0 (SIPL):  Reading from file "+filename+"
. . .");
            try {
                       sipl = new SIPL(new java.io.FileInputStream(filename));
            } catch (java.io.FileNotFoundException e) {
                        display("Lucid Parser Version 3.0 (SIPL):  File " + filename +
" not found.");
                        return;
                  }
            sipl.startParse(sipl, this);
               }
            }

        void display(String message) {
          System.out.println(message);
        }

        void treepass(SimpleNode simpleNode) {
          this.simpleNode = simpleNode;
        }

        void showtree() {
          simpleNode.dump(" ");
        }

        void translate() {
          translator = new Translator();
          translator.translate(simpleNode);
        }
}
```