ELIAS ISBER

A COMPILER FOR ALL CONTEXT-FREE GRAMMARS GENERATING JAVA BYTECODE

Mémoire
présenté
à la Faculté des études supérieures
pour l'obtention
du grade de Maître ès Sciences (M.Sc.)

Département d'Informatique
FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL

Juillet 1997

To my family,
To all my friends.

Acknowledgments

My heartfelt appreciation and thanks go to my supervisor, Professor Jean Bergeron, for providing me with the great support throughout the course of my study at Laval. I am grateful for his understanding, patience, and kindness. He was always there to listen and to help. THANKS a lot Jean.

Thanks also to the other members of my thesis committee: Mourad Debbabi, Nadia Tawbi. Besides serving on my thesis reading committee and providing feedback. Mourad Debbabi also provided advice and direction during my first year at Laval. His weekly course meetings throughout the semester were interesting and educational. I appreciate Mourad's strong support to my work.

I appreciate the support and patience of my new colleagues at Laval University for commenting on my thesis and also providing me with great ideas to test my software, especially M. Mejri. He is the one who will never say no, the one who has an answer for almost every question. I would like to say a special thanks to all LSFM members who made a friendly environment to work together at the laboratory. Special thanks are also due to A. Faour, I. Mounkaila, and I. Yahmadi.

Finally, I thank my family for their continuous encouragement and for making this education possible.

Résumé

La pluplart des générateurs de compilateur(YACC, VATS, etc...) utilisent une forme restrictive de grammaires de type 2 (LL(1), LR(1), de préséance. etc...) afin d'obtenir de la performance. Cela oblige les utilisateurs à modifier leur grammaire naturelle afin de se plier aux contraintes des générateurs. L'algorithme d'Earley permet d'analyser une grammaire de type 2 quelconque sans/ou avec production vide. Un générateur basé sur cet algorithme permettrait d'accepter une grammaire sans modification. Le but du projet est de développer un analyseur lexico-syntaxique qui accepte une description des symboles ainsi que de la grammaire formelle et qui génère du "bytecode" au sens de Java afin de le rendre transportable. La possibilité d'inclure des productions vides est considérée. Une méthode de spécification des opérateurs d'un langage en terme de "bytecode" est développée.

Québec, Juillet 1997

Elias ISBER
Étudiant

Jean BERGERON
Directeur de recherche

Abstract

Most compiler generators(YACC, VATS, etc...) use a restrictive form of type 2 grammars (LL(1), LR(1), Precedence, etc...) in order to obtain performance. This obliges the users to modify their normal grammar in order to conform to the constraints of the generator. Earley's algorithm allows to analyze any type 2 grammar with/without ϵ-production. A compiler generator based on this algorithm may accept any grammar without modification. The goal of the project is to implement a compiler that accepts any description of symbols as well as any formal grammar, and generates Java protable bytecodes. The possibility to include ϵ-production is considered. A method of specification of language operators in terms of bytecodes is also developped.

Quebec, July 1997

Elias ISBER
Student

Jean BERGERON
Supervisor

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

In the last decades, context-free grammars have been widely used to describe the syntax of most programming languages. Many parsing techniques for context-free grammars were introduced, thus playing an important role in the implementation of compilers and interpreters. However, most of the parsing algorithms used nowadays in compilers are limited to a small set of context-free grammars, namely the unambiguous grammars. If ambiguous grammars are allowed, there is no guarantee of correctness. Further, since lookaheads are fixed, only deterministic grammars can be handled. On the other hand, the broader the class of grammars a parsing technique can handle, the better a compiler generator could be obtained. The mostly known parsing techniques are LL(1), LR(1), LALR(1), SLR(1), and precedence methods [5, 1, 2]. LL(1) has the advantage in all areas except generality. Neither LL(1) nor LALR(1) can handle all unambiguous grammars. However, all LL(1) grammars are LR(1) and virtually all are also LALR(1). LL(1) parsers are fairly strict about grammar forms, forbidding left recursion and productions that share a common prefix, which is a real problem not allowing a considered number of grammars to be used. LALR(1) accepts a better subclass of grammars. It can detect errors in a more accurate way. It is linear in operation. However, the parse table can grow rapidly in dimension using lookahead techniques thus making it so difficult to generate it manually. The precedence techniques are conceptually simpler than the LR techniques. But they have many drawbacks, primarily the restricted class of grammars they can parse. Moreover, precedence grammars are much more difficult to construct than LR grammars. Occasionally, a general parser was needed for any context-free grammar defined. A very efficient parsing algorithm was introduced by Jay Earley(1970). This algorithm is a general context-free parser. It is based on the top-down parsing method. It deals with any context-free and ϵ-free rule format without requiring conversions to Chomsky form, as is often assumed. Unlike LR(k) parsing, Earley's algorithm never needs or uses lookahead. Rather, whenever there is any doubt about to do a reduction, all possibilities are followed, in parallel. Further, no states or automata need to be computed in advance. Rather sets of configuration are manipulated as the parse proceeds and that is why the algorithm is considered as interpretive.

Hence, this algorithm is so appealing because it runs with best-known complexity on a number of special classes of grammars. But, it is sometimes more expensive in terms of time. In particular, it may requires cubic time and quadratic space compared to other algorithms working in linear time. However, this non-linear performance usually occurs when Earley's algorithm performs much more work that is beyond the capability of other unsatisfying parsers [4].Consequently, we see that Earley's algorithm can be as competitive with other parsers and it has not been implemented for a compiler generator nor tested till our days. Few questions arise here.

1. With the powerful computer architectures we have reached nowadays, why don't we take advantage of this algorithm by trying to implement it?

2. Why don't we try to integrate it with a real compiler that works for all types of context-free grammars?

Another problem arises with our currently existing compilers. Most of today's compilers generate bytecodes, a lower level format obtained from the compiled source code. In fact, these bytecodes are usually designed to be interpreted on the specific machine on which they were compiled. After a compiler proves to be successful, a demand often arises to move its bytecodes to new machines, or to new versions of existing machines and with a certain model of security. Hence, a design of a new bytecode that provides separation between source language and target machine concerns anticipates future needs. Finally, a new vision was observed by a team of the SUN Microsystems company, that is, to develop a new programming language which generates portable bytecodes that can be shared between different computer architectures and further interpreted according to the specific machine. The resulting language was the Java language. Here another question can be asked in this study. With the flexibility and portability of Java bytecodes introduced in the Java virtual machine recently, why don't we add this new vision to this compiler, based on Earley's algorithm, to generate transportable code?

The compiler described in this paper provides solutions to all the above questions in well structured modules, taking a flavor of LEX [8] and YACC facilities [9, 3]. To remind, LEX is a lexical analyzer generator widely available under the UNIX operating system. From a description of the tokens grammar, it generates transitions and output tables for a finite state machine(FSM) and a driving program in C language. LEX uses regular expressions to describe the syntax of each token. YACC is an LALR(1) parser generator. It can use scanners generated by LEX and it generates a parser not an entire compiler. However, our work is viewed as a compiler for any CFG and generating java bytecode that is transportable to any machine. We leave the programmer interested in using our compiler to implement his own interpreter according to his machine architecture.

1.2 Goals of this Memoir

In light of the problem just described, the target of this memoir is to develop a general compiler based on Earley's algorithm, and generating java bytecode, that is, to produce a new application capable of compiling a wide number of programming languages according to their CF grammars. The input to the system is three text files. One file contains the context-free grammar for the specific language as well as the java bytecode to be generated by each production. This file will be used by Earley's parser. Another input file includes a description of the tokens belonging to the language grammar as well as the terminal types to be used. This file will be used by the lexical analyzer which is also based on Earley's algorithm. A third file contains the source code to be compiled by the resulting application. The output will be a file of generated java bytecodes, enabling the user to transport his bytecode on any machine and execute it by the corresponding interpreter.

Importantly, we will in this work concentrate on the use of Earley's algorithm in both the lexical analyzer and the parser. Combining these two modules together with building the parse tree from the rules obtained by the parser is an essential task for our compiler. Moreover, a treatment of a small part of the semantic analysis as well as code generation will take place. In fact, we will see that recovery from misdeclared identifiers in our compiler based on Earley's algorithm is not an easy job to perform. That's why we will have less to say about the other part of semantic actions and interpretation. Consequently, we will leave these latter parts to those interested in further implementation of the compiler, thus having a more adequate application.

1.3 Overview of the Memoir

The groundwork for developing a system along the lines sketched above, is an analysis and a design for the whole compilation process as well as a thorough study of linking together the different parts of the compiler. Writing a compiler is not an easy job to perform. However, it can be realized in a modular form knowing that it's usually structured in modules. One contribution of the memoir is thus in identifying the appropriate parts of the compiler, and finding a way to implement each of these parts. Figure 1.1 shows the sequence of modules used in the implementation of the whole application. Four fundamental parts on which we are based to build our compiler are as follows.

- Lexical analysis
- Syntax analysis
- The symbol table and error detection
- Code generation

Before describing the task of each of these components, it is worth mentioning the method on which most of our work is based. To this end, we will in chapter 2

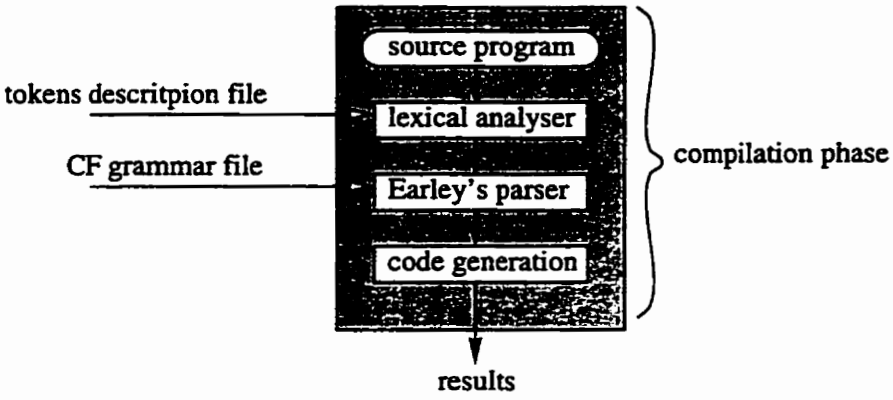


Figure 1.1: The compiler design.

describe Earley's algorithm on which are based both the scanner and the parser. In later chapters we will then point out how exactly this algorithm relates to each of the just mentioned parts of our compiler.The subsequent task is to describe the method of implementation for each part of the compiler. To accomplish this, we have to define first the function of each component.

The lexical analyzer(Scanner) inspects the source code, character by character, eliminates superfluous characters, recognizes the tokens, constructs the lexemes from these characters and passes them to the parser. Since our lexical analyzer is based on Earley's algorithm, it is considered to be a general scanner. Thus, it requires an input file for the specification of tokens. Hence, the input file used by our scanner contains a description of the syntax of each token. It is used to distinguish all the tokens to be obtained from the source code. It consists of three sections:

1. *The Token Definitions section* allows us to predefine the reserved words in the language as well as the specification of terminal symbols passed from the lexical analyzer to the parser. Usually, each language has its own specific reserved words. So we have to specify them in this file in order to distinguish them from the non-terminals when parsing the tokens. Each line in this section consists of a name(token,terminal) on the left and a subset of the reserved words on the right. We need also to define the identifiers because when passing them to the parser, we have to distinguish them in order to pass their corresponding lexical values. As an example, the definition section for the BASIC language includes the following:

```
%%token  FN SIN COS TAN ATN EXP ABS LOG SQR INT RND
%%token  DIM LET DATA RESTORE GO TO ON GOSUB RETURN
%%token  THEN FOR NEXT STEP IF REM STOP END READ PRINT
%%token  DEF FNEND
%%token  + - * / =
%%token  | < > . , ; $  ? ( )
%%terminal fraction decimal lettre message nombre_signe  rem
```

```
%%terminal variable_simple entier nombre_exponentiel
```

2. *The Type definitions section* used to specify the type of each terminal(float, integer, double,...). In fact, our compiler doesn't perform semantic error detections. Recovery from an undeclared variable name is a so difficult task to do when using Earley's algorithm. Since each language has its specific syntax for type declarations, it is not that easy to indicate when a name is misdeclared or not. For instance, in C language, when declaring a variable we have the syntax `type variablename` while in PASCAL language we have `variablename: type`. However, our compiler requires to know the type of each terminal; the reason being is when inserting a variable name in the symbol table, we have to know its type. Thus, we used this section as a trick so that when a variable is assigned a terminal value, by knowing its type(integer for example), we can declare that variable of the type integer. An example of the type definitions section, for the BASIC language is the following:

```
%%type fraction           :D
%%type decimal            :F
%%type entier             :I
%%type nombre_exponentiel :F
```

Each terminal is defined on the left and its type on the right. For instance, the identifier `fraction` has the type `D` i.e. double while `decimal` has the type `F` i.e. float. We will see that this part plays a big role in the java code generation because java bytecode instructions are so specific i.e. each type has its own instruction.

3. *The Rules section* is where we define the rules describing the syntax of each token. Each rule has a left_hand side and a right_hand side. These latters are divided to terminal and non_terminal symbols. Rules are separated by `$$` in the file. An example of this section, showing the lexical definition of an integer, is the following:

```
start      := entier $$
entier     := chiffre1_n $$
chiffre1_n := chiffre chiffre1_n  $$
chiffre1_n := chiffre $$
chiffre    := 0 $$
chiffre    := 1 $$
chiffre    := 2 $$
chiffre    := 3 $$
```

The parser performs a syntax analysis of the source code. It is based on the tokens obtained from the lexical analyser. The parsing procedure consists on verifying if the program is syntaxically correct, i.e. if the sequence of symbols in the source

code does respect the grammar rules of production. Since our parser may be based on any context-free grammar, there must be an input file associated to it. This input file consists of two sections. The first section contains the CF grammar describing the syntax used to verify the correctness of the input strings. This section is divided into three fundamental parts separated by $$ sign.

- The first part contains the rules of production of the currently used grammar.
- The second part contains a number indicating the priority of each rule of production. This number is used for the treatment of precedence operators. The fact that Earley's algorithm does not treat the problem of priority between operators. It is left for the implementor to provide the solution. And, this is the purpose of using the indicated number.
- The third and last part is used in the code generation. Its contains the java bytecodes. Each rule of production may have a corresponding subset of intsructions, each of which has a precise specification of exactly what each bytecode does to the virtual machine. In general, most compilers have their bytecodes embedded in their implementation. However, in compliance with our general compiler, we chose to put this part in front of each rule of production, to facilitate the work for the user.

The second section includes the specification of variables, arrays, functions, function parameters, and so on... . Each language grammar usually has a specific syntax to distinguish identifiers from each others, for exemple a `variable` from an `array`. The purpose of this distinction is to enable us to store each identifier in its appropriate place in the symbol table knowing that this latter is structured to fit each type of identifier. The easiest way to handle this problem in our general compiler is to inroduce this section so that we can easily verify the type of identifiers before storing them in the symbol table.An example of this specification is the following.

```
%%variable    variable                : variable_simple
%%variable    -                       : variable_simple
%%function    def                     : lettre
%%function1   entete                  : lettre
%%array       dim_tableau             : lettre
%%array_len   dim_tableau             : entier
%%array_len   c_entier0_1             : entier
%%fun_par     def                     : variable_simple
%%fun_par     c_variable_simple0_n    : variable_simple
%%fun_par1    entete                  : variable_simple
%%fun_par1    c_variable_simple0_n    : variable_simple
%%
```

The first name written with the sign %% in the left is a definition of the identifier type to be detected during analysis(variable,array,...). Then comes the two non-terminal

symbols before and after the colon. During the parsing process, each of these are compared with left_hand and right_hand sides to detect the type of identifiers

A compiler needs to collect and use information about the names appearing in the source program. This information is entered into a data structure called the ***the symbol table***. Unlike other compilers designed to fill their symbol tables during the lexical analysis phase, we will treat this problem after the syntax analysis process. After parsing all the tokens with Earley's algoithm, we will obtain a series of rules from which the parse tree is constructed. We will see how we can fill our symbol table when traversing this parse tree. Another important phase is worth mentioning together with this section. A compiler must perform many semantic checks on a source program as well as it must detect ***lexical and syntax errors*** during analysis. Many of these are usually done in conjunction with the syntax analysis except the lexical error detection. However, we don't address all these problems in our general compiler. The reason is that this generality prevents from treating almost a big part of semantic checking, specifically type checking. We will talk about the lexical and syntax error detection that can be treated when using Earley's algorithm.

The code generator is the final phase of our compilation process. It is that part which deals with code production. The produced code may be a machine code, an intermediate code or even another programming language. It is the component that realises effectively the translation. Accurate code generation is difficult, and not too much can be said without dealing with the details of particular machines. However, this subject was studied with careful code-generation method which can easily produce java bytecodes that can run on any particular machine after interpretation.

1.4 Organization of the Memoir

Chapter 2 briefly reviews the workings of an Earley parser. It describes the method on which is based the algorithm. It finally shows an example of processing a sentence by the parser.
Chapter 3 is the beginning of the compilation phase; it introduces the topic of lexical analysis and describes the method that will be used in implementing our scanner.
Chapter 4 provides a brief decription of the syntax analysis process in our compiler. It also shows the method we introduced to obtain our parsing tree to be traversed in the code generation.
Chapter 5 discusses the method of building out our symbol table and filling it with identifiers. Its shows the difference in its implementation between our system and other compilers. Following this description, the overall detection of lexical and syntax errors procedure is specified in detail.
Chapter 6 takes the parse tree generated in the parser and shows its use in the code generation process. It also gives a brief descritpion of the java virtual machine on which part of the code generation is based.
Chapter 7 summarizes the work, compares it to other works, states the contributions

made by the memoir, and points to some modifications for future research.
Appendix shows an example of the BASIC language grammar file and its tokens decription file, and the complete program source code of the compiler.

Typeface conventions Although the distinctions are not always straightforward to make, the memoir uses different typefaces to seperate entities belonging to different realms of representation. *Brackets* are used for some examples. *Italics* are reserved for emphasis. Some proper names of systems, grammars or languages appear in UPPER CASE, and excerpts of actual program code or representations in `typewriter`.

Chapter 2

Earley Parsing

2.1 Introduction

As we indicated in section 1.3, each compiler consists of many modules. The two important modules are the lexical analyzer and the parser. These modules are usually based on a certain method of implementation. We describe in this chapter the Earley's method used in the implementation of these two modules in our compiler. An *Earley* parser is essentially a generator that builds left-most derivations of strings, using a given set of context-free rules of production. The word *parse* comes from the fact that the generator outputs all possible derivations that are consistent with the input string up to a certain point. As more and more the input grows, the set of legal derivations can either expand as new choices are introduced, or shrink as a result of the resolution of ambiguities. Thus, in describing the parser, it is appropriate and convenient to use some kind of terminology.
For each position in the input, the parser keeps a set of *items* describing all the possible derivations. This set of items together form a *state* and the set of states is sometimes called the Earley's *chart*. An item is of the form

$$A \rightarrow \alpha.\beta, i$$

where the symbol A is a nonterminal of the grammar, α, β are strings of nonterminals and/or terminals, the dot is a pointer to the next symbol to be analysed, and i is an index representing a state number. The items corresponding to each state are derived from productions in the grammar.

2.2 The Parser

2.2.1 Informal Explanation

Before giving a formal description of the algorithm, we start by an informal explanation of the method. We take as an example the above item. It is derived from a corresponding production

$$A \rightarrow \alpha\beta$$

with the following semantics:

- We have an input string $x_1 ... x_n$ to be scanned from left to right looking ahead at a fixed position i. The set of items describing the input processing at position i is called state i. Note that there is one more state than the number of input symbols. State 0 holds the starting items before any input is processed and state $n + 1$ contains the final items after all the input has been processed.
- Nonterminal A in the above item was expanded starting at position k in the input, i.e. A generates a substring starting at position k.
- The expansion of A proceeded using the production $A \rightarrow \alpha\beta$, and has expanded the right_hand side (RHS) $\alpha\beta$ up to the position indicated by the dot. The dot refers to the current position i.

An item with the dot to the right of the entire RHS is called a *complete item*, since it indicates that the left_hand side(LHS) nonterminal has been totally expanded.

2.2.2 The Recognizer

An optional feature is discarded from our description of Earley's parsing, the *lookahead string*. This feature may probably be used during parsing, in order to process LR(k) grammars deterministically, thus obtaining the same complexity as LR(k) parsers by reducing the number of items when possible. Since we are not interested in this feature in our implementation, we have omitted it from the following explanation.

We define three main operations of the parser that will be executed on the current state or the set of items and the current input symbol. These operations thus add new items to this state and may create a new state of items. The initial state is supposed to start at index 0 and the initial item is of the form

$$S \rightarrow .\alpha, 0$$

where 0 is the predication pointer.

The three types of transitions operate as follows.

prediction For each item

$$A \rightarrow \alpha.B\beta, j$$

where B is a nonterminal in the RHS, we predict $B \rightarrow .\gamma, i$. The prediction pointer is i corresponding to the index of the current state i. An item produced by prediction is called a *predicted item*. Each prediction corresponds to an expansion of a nonterminal in a leftmost derivation.

Scanning For each item

$$A \rightarrow \alpha.X\beta, j$$

where X is a terminal anywhere in the RHS that matches the next input symbol, then add to the state $i + 1$, $A \rightarrow \alpha X.\beta, j$. We move the dot one position forward, i.e. the

symbol has been already read. We call the new produced item a *scanned item.*

Completion For each item

$$A \rightarrow \gamma., j$$

then we know this production was predicted at state j. Search in state j for each item that has A to the right of the dot, i.e. for items of the form $B \rightarrow \alpha.A\beta, k$. Add the item $B \rightarrow \alpha A.\beta, k$ to the current state. An item produced by completion is called a *completed item.* Each completion corresponds to the end of a nonterminal expansion starting at a prediction step.
The three operations just described are performed repetitively for each input symbol and its corresponding state of items, i.e., until no more new items are generated. Although both prediction and completion feed themselves, there is only a finite number of items that can possibly be produced. Therefore, recursive prediction and completion have to terminate eventually, and the parser can proceed to the next input symbol by scanning. After processing the last input symbol, the parser verifies if the item

$$S \rightarrow \alpha., 0$$

is found in the last state produced. If so, this means that the algorithm was terminated successfully. If not, the parse was aborted due to an unmatching input symbol by the set of items in a given state i. As a simple example we will consider the ambiguous grammar

$$\begin{aligned} S &\rightarrow .A \\ A &\rightarrow .A + A \\ A &\rightarrow .Id \end{aligned}$$

Figure 2.1 shows the parsing procedure of the input $Id + Id + Id$. Because $S \rightarrow A., 0$ is found in the last state, it means that the input string was successfully parsed. The parse can generate two possible trees since $A \rightarrow A + A., 0$ corresponds to a parse of $(Id + Id) + Id$ while $A \rightarrow A + A., 2$ corresponds to parsing $Id + (Id + Id)$.

2.3 Conclusion

It is easy to see that Earley parser operations are *relevant*, in the sense that each set of transitions(predictions, scanning steps, completions) corresponds to a possible derivation. It is also true that a parser that performs these operations repetitively is *complete*, i.e., it finds all possible derivations. Formal proofs of these properties are given in Aho & Ullman(1972) [1]. The parse trees for input strings can be reconstructed from the set of states. We will illustrate this in chapter 4.

Earley's parser can deal with any type of context-free rule format, even with null or ϵ-productions, i.e., those that replace a nonterminal with the empty string. Such productions do however require special attention, and make the algorithm more complicated than its description in the previous section. In conclusion, Earley's algorithm runs with the same complexity of other parsers as well as it may surpasses some best

id	+	*id*	+	*id*	
$S \rightarrow .A$,0	$A \rightarrow id.$,0	$A \rightarrow A + .A$,0	$A \rightarrow id.$,2	$A \rightarrow A + .A$,2	$A \rightarrow id.$,4
$A \rightarrow .A + A$,0	$S \rightarrow A.$,0	$A \rightarrow .A + A$,2	$A \rightarrow A + A.$,0	$A \rightarrow A + .A$,0	$A \rightarrow A + A.$,2
$A \rightarrow .id$,0	$A \rightarrow A. + A$,0	$A \rightarrow .id$,2	$A \rightarrow A. + A$,2	$A \rightarrow .A + A$,4	$A \rightarrow A + A.$,0
			$A \rightarrow A. + A$,0	$A \rightarrow .id$,4	$A \rightarrow A. + A$,4
			$S \rightarrow A.$,0		$A \rightarrow A. + A$,2
					$S \rightarrow A.$,0
					$A \rightarrow A. + A$,0
State 0	1	2	3	4	5

Figure 2.1: An Example of Earley's Algorithm

results in terms of time. It does this with one single algorithm with no specific class of grammars to it, i.e., it does not require the grammar in any special form. It is true it might sometimes require more time but this case will be when it performs operations beyond the capability of other parsers.

Chapter 3

Building the Lexical Analyser

3.1 Introduction

In the first chapter of Introduction, we saw that the lexical analyser scans the source program, character by character, eliminates superfluous characters, recognizes the tokens and constructs the lexemes from these tokens and passes them to the parser. A token is a generic term that indicates a set of lexemes having the same grammatical meaning. For instance, in the instruction $V = V_1 + V_2$, V,V_1, and V_2 are three lexemes having the token *identifier*. Concerning the elimination of superfluous characters, these ones come from two different sources, namely comments and characters used to separate two entities: space, tabulation... In order to manage easily all identifiers in the program, a lexical analyser(scanner) converts all read letters to small or capital ones depending on the requirement of the programming language syntax. The lexical analyser, we are implementing in our compiler, is supposed to fit for most types of programming languages. Thus, we have to find a general method to deal with every description of tokens. For this purpose, this lexical analyser takes as a part of the input a file containing the tokens description specified by the user. Based on this description, it generates tokens together with their corresponding lexemes, and passes them to the parser for the next analysis stage.

In this chapter, we discuss the data structures needed to support lexical analysis, the method of constructing our scanner, and all the necessary functions used in this latter. We see in brief the task of the lexical analyser by resolving around three issues which will be discussed in turn:

- How do we represent tokens described in the input file, for appropriate use and efficiency in terms of time during lexical analysis?
- Why and how Earley's algorithm is used in this scanner while other compilers use finite state machines to recognize their languages?
- How are tokens generated when using Earley's algorithm and passed to the parser?

3.2 Regular Grammars Versus Regular Expressions: Background

In programming languages the concept of a regular grammar is generally sufficient to represent common lexical units(tokens). Instead of working directly with a regular grammar, most compilers implementors use the concept of regular expressions, seen as a more convenient and intuitive way to represent regular languages. In fact, a regular expression E defines a regular language $L(E)$. Given an alphabet(finite set of symbols) T, a regular expression is one of the following:

- a symbol of the alphabet set T
- the empty string ϵ
- the empty set ϕ
- obtained by applying a finite number of times one of the following operations:
  - concatenation e_1e_2
  - union $e_1|e_2$
  - Kleene closure e^*

  where e, e_1, and e_2 are regular expressions.

Once they have regular expressions, implementors can construct equivalent finite automata. LEX, a very known lexical analyser generator, uses this concept. Regular expressions used in LEX are composed of normal characters and meta-characters having a special meaning(., $, [,], +, *, ...). The procedure that LEX follows to construct an automaton is performed in three steps:

- it constructs a non-deterministic finite state automaton from the regular expression.
- it transforms the non-deterministic finite state automaton obtained to an equivalent deterministic finite automaton.
- it minimizes the number of states in the automaton finally obtained.

Building the tables to drive a FSM to do lexical analysis for a complete computer language would be extremely tedious. In constrast to this method using regular expressions and finite automata, our lexical analyser based on Earley's algorithm, uses only regular grammars. It doesn't need to confuse with regular expressions and automata. This concept of using a regular grammar is more efficient theoretically and practically speaking. It is more easy to manipulate it for representing regular languages. In fact, a grammar defines the set of rules of production in a general form. What distinguishes this most general form, is that with a regular grammar, Earley's analyser can choose

an execution between many others which is easier than using states with finite automata. Many transitions are thus possible for the same character as well as the null production. Transitions for words of length 0 or more are accepted. Earley's analyser works similarly to the *greedy* algoritm. To clarify, in this context, if the programmer has specified ambiguous rules of production describing the tokens, it is the rule that accepts the longest input string which is preferred. In the case where many rules are possible for the same input, this analyser takes into consideration the possibility for generating three tokens that are taken away. This is a special case treated in our lexical analyser in contrary to most others using regular expressions and taking away only the first token to be obtained during analysis.

3.3 Principles of Functionning of our Lexical Analyser

3.3.1 Top-down Description of the Scanner

We have seen that our lexical analyser based on Earley's algorithm does work directly with regular grammars. There is no need to build finite automata that represent regular expressions. But how can it recognize tokens when scanning the input in this case? In fact, this analyser takes the grammar describing the tokens and stores it in a structured list of rules. These rules are the base of the scanning procedure. They replace finite automata used in implementing other lexical analysers. To be more precise, we need a subset of rules for each token to be recognized. This set of rules is a description of the token syntax, and conceptually we need to run them in parallel on the same input. As an example, the set of rules:

```
entier      := chiffre1_n $$
chiffre1_n := chiffre chiffre1_n  $$
chiffre1_n := chiffre $$
chiffre     := 0 $$
chiffre     := 1 $$
chiffre     := 2 $$
   ⋮            ⋮
chiffre     := 9 $$
```

represents the syntax of an integer token called `entier`. As a convention, we suppose to have at the beginning of the rules list, the names of all types of tokens expected to appear in scanning the input program, to be preceded by the `start` clause. For instance, in the BASIC language, the specification of tokens includes the following starting rules:

```
ss    := start $$
start :=  fraction $$
start :=  decimal $$
start :=  lettre $$
```

```
start := entier $$
start := message $$
start := nombre_signe $$
start := nombre_exponentiel $$
start := rem $$
start := variable_simple $$
start := char_special $$
```

After storing the grammar rules in the list structure, the lexical analyser is called by the parser each time to get the next token. When starting, the analyser passes by(ignores) all the separators. Then, based on the current character(letter, digit,etc), it starts by predicting one rule or more that matches the input. It keeps on realising many transitions until no more input can be read. It eventually verifies if the starting rule `ss := start` was found in the final state of rules. If so, it means that it has recognized a token. The analyser then compares the obtained token with the reserved words stored in the table. In case these latters are not matched, the token is compared with the different types of token terminals coming from the starting rules and if it was found, it is passed to the parser with its corresponding lexeme. If it is not the case, we realize that there is a lexical error.

3.3.2 Structural Representation of Data

Before examining the scanner and each of its submodules in more detail, it is necessary to describe how the necessary data can be structured. The first data structure to be discussed is the representation of the tokens. Although the tokens are not represented by a complex data structure, their use is so important that they are discussed here. When a token is encountered, some representation of that token must be returned to the parser. The easiest and simplest way to do this is to have a unique number for each token. Unfortunately, this representation makes the code for the compiler difficult to understand, since the numerical representation for each token must be remembered. To overcome this difficulty, our implementation includes a global array called `t_reserved_words[nb_reserved_words]` of string elements. This array is divided in three parts. The elements of the first part of the array store all the reserved words specified in the tokens definition section of the grammar input file described in section 1.3. Each reserved word will have the number of the cell it holds as its numerical representation. This number is incremented sequentially by the variable `last_reserved_word` which is first initialized to 0. The second part of the array is used to store all the terminals also specified in the tokens definitions section of the input file. Actually, these terminals are the tokens passed with their corresponding lexemes to the parser. We need also an internal representation for these tokens. Each terminal symbol takes as numerical representation the number of the cell holding it. This number is incremented sequentially by the global variable `last_terminal` initialized to the last value given to the variable `last_reserved_word`. The third and last part of the array is reserved for the parser use. It includes all nonterminal symbols of the language grammar used for parsing the tokens. we will see this part in more detail when illustrating the parser's

full implementation.

The next important data structure to be discussed is the globally declared table of identifier types `t_id_types[10]`. As mentioned earlier, our compiler doesn't treat language grammars like C, Pascal, ..., that have an obligation to declare their types of identifiers(variables,arrays,etc) in the source program. However, our implementation is valid only for those grammars whose identifiers take the type of their assigned values. As the types of identifiers vary from one language to another, their specification described in the types definition section of the input file, must be stored in a table called `t_id_types[10]` of structures. Each of these latters hold the name of the identifier in a variable called `symbol` as well as its type in a variable called `type`. The type specification structure looks like the following:

```
typedef struct type_specification
{
   int symbol;
   int type;
} *type_specification_ptr;
```

To illustrate with BASIC, in the instruction `A = 1` the variable `A` turns automatically to be of type integer because it was assigned an integer value. Actually, while scanning the source program, this value of `1` is nothing but a lexeme of the token `entier`. To declare the variable `A`, we look up in the table `t_id_types` for the token `entier` and gives its corresponding type, integer in this case, to the variable `A`. We will see in more detail, in the chapter of the code generation, the importance of this table. The reason is because java bytecode instructions are so specific for each type of identifier.

After the illustration of the two global data structures used in the scanner and other parts of the compiler, we proceed to enter the *lexer* class. This class contains all the functions that build in our scanner. A very important data structure is the pointer variable `InitRules`. This variable is declared as private for the class. It is a pointer to a linked list of structures holding the initial rules that describe the tokens syntax. It plays an important role in the scanning process. It is used each time there is a new prediction. In fact, all the predicted rules added to a certain state are extracted from the contents of this pointer variable. The structure of the list to which `InitRules` points to, looks like the following:

```
typedef struct RULE
{
  int pred_ptr, priority;
  int left_hs;
  char *semantic_value;
  struct WORDNODE *right_hs;
  struct WORDNODE *right_hs_tail;
  struct WORDNODE  *after_ptr;
  struct WORDNODE *semantic_rule;
  struct RULE *ref_ptr;
  struct RULE *ref_ptr1;
```

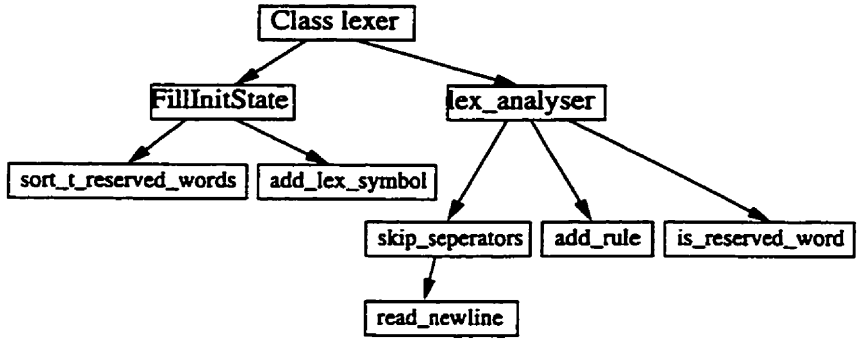


Figure 3.1: Representation of the scanner.

```
  struct RULE *next;
} *RULEPTR;
```

The variable `pred_ptr` holds the prediction pointer. This latter is the number of the state having new predictions(refer to section 2.2). left_hs is a variable holding the left_hand side of a rule. `right_hs` is a linked list of symbols. These symbols are the right_hand side part of a rule of production. `next` is a pointer to the next rule of production. Other variables are to be used in the second part of our compiler, i.e., in the *parser* class. We will discuss these variables in detail in later chapters.

The last and most important data structure in the class *lexer* is the table `state[MAX_STATE]`. This is the table holding the states of Earley's algorithm. An element `state[i]` of the array is a pointer to the list structure `struct RULE` just defined above. In other words, it is a pointer to the list of rules of a state i. In order to search easily for a specific state(by number) when there is a completion(refer to section 2.2), we chose to save the rules sets of each state in this array of states.

3.3.3 Implementation of the *lexer* Class

The class *lexer* is designed to be an interface between the input stream and the parser of the compiler. It consists of two main functions or methods, namely `FillInitState` and `lex_analyser`. These two methods may call other subordinate procedures that will be discussed in detail in this section. Figure 3.1 demonstrates the calling sequence of the lexer class and the subordinate functions for each method. In the diagram, a method invokes those functions at a lower level with which it is connected. For example, the `lex_analyser` invokes the `skip_seperators` function, and the `skip_seperators` function invokes the `read_newline` function.

The `FillInitState` method is the one that performs the numerical representation of the tokens as well as it stores the grammar, specifying the description of tokens, in the list of rules. It works in the following manner. First, the two functions `initialize_t_reserved_words` and `initialize_t_id_types` are called. These two routines initialize the tables of reserved words and identifier types. Then, the file containing the specification of tokens and their grammar is opened. `FillInitState` starts

by reading the tokens definition section in the opened file. It looks for symbols defined as reserved words and/or terminals. When the `%token` clause first appears, it means that the tokens following it are reserved words. They are stored in their proper place in the table `t_reserved_words` described in section 3.3.2. If the `%terminal` clause appears, the terminal symbols following it are the tokens that will be passed to the parser with their corresponding lexemes. They are stored in the second part of the array `t_reserved_words`. Then it comes the types definition section. When the `%type` clause appears, it means that `FillInitState` stores the information about terminal symbols and their types in the table `t_id_types`. This information is used to declare identifiers in the code generation phase(refer to section 3.3.2). The final clause expected to appear is the `%assign_op` clause. The symbol following this latter is stored in the global variable `assign_op`. It is used to identify the assignment operator of the language. Note that this latter may vary from one language to another and this is the purpose of specifying it in the description file.

After storing the reserved words in the array, they have to be sorted in alphabetical order. The function `sort_t_reserved_words` is called to perform this job. The reason to sort these tokens is to facilitate the search when looking for the nature of an obtained token. In other words, each time a token is obtained during the lexical analysis, we have to verify if it is a reserved word or a terminal. Thus, when these symbols are sorted, it is easier for the analyser to look them up in the table. The function `sort_t_reserved_words` looks like the following.

```
void lexer::sort_t_reserved_words() { int
i,j; char *temp;

  for(i=0;i < last_reserved_word;i++)
    for(j=i; j < last_reserved_word;j++)
      {
        if (strcmp(t_reserved_words[j],t_reserved_words[i]) < 0)
             {
               temp = t_reserved_words[i];
               t_reserved_words[i] = t_reserved_words[j];
               t_reserved_words[j] = temp;
             }
      }
}
```

The third important task of the function `FillInitState` is to store the rules of production describing the tokens in the variable `InitRules`. This variable is a pointer to a list of structures(see section 3.3.2). Of course, when talking about a rule of production describing a token, it has symbols defined as terminals, non_terminals, characters, digits,.... These symbols also need to be assigned a numerical representation. The reason is, when using Earley's algorithm in the implementation, it is preferred to do comparisons by numbers and not by symbols for better performance in terms of time. For this reason, we store all these symbols of the grammar rules in a table called `lex_symbols`. The first

256 entries of this table correspond to the ASCII characters that might be used when scanning the input stream. The remaining part of the array is left for non_terminals of the grammar rules. When reading a rule from the file, each of its symbols is given an entry in the array `lex_symbols`. It is the number of this entry that is stored when filling the list of rules pointed to by `InitRules`. In other words, the symbols stored in the variables `InitRules->left_hs` and `InitRules->right_hs->word`, are nothing but the numbers corresponding to those symbols stored in the array `lex_symbols`.

After filling the linked list of the description rules pointed to by `InitRules`, we proceed to start with the scanning procedure. It is the function `lex_analyser` that performs the job. The main data structure on which `lex_analyser` is based, is the table `state[MAX_STATE]` described in section 3.3.2. To remind, this latter is the table holding all the states of the rules sets corresponding to Earley's algorithm. Each entry in the table points to the structure `states` that looks like:

```
typedef struct states
{
  RULEPTR RFirst,RLast;
} STATE;
```

`RFirst` is a pointer to a set of rules. It points to the first rule in each state having a rules set. `RLast` is a pointer to the last rule of a state. We kept this latter in order to add rules at the end of each state easily, each time there is a new transition(prediction, scanning step, completion).

The function `lex_analyser` starts by initializing the table of states. When scanning the input stream, each time obtaining a token the table entries have to be reinitialized to `NULL` to avoid confusion. The, we have to initialize the first entry in the table `state` to the first rule of production. This rule is supposed to be

```
ss := start $$
```

and is the starting rule that leads to other transitions. Moreover, this rule comes from `InitRules` first initialized in the function `FillInitState`. The source code of initialization part looks like the following:

```
for (j=0;j<MAX_STATE;j++)
  {
    state[j].RFirst = state[j].RLast = NULL;
  }

state[0].RFirst = new struct RULE;
state[0].RFirst->pred_ptr = 0;
state[0].RFirst->left_hs = InitRules.RFirst->left_hs;
state[0].RFirst->right_hs = InitRules.RFirst->right_hs;
state[0].RFirst->after_ptr = InitRules.RFirst->after_ptr;
state[0].RFirst->next = NULL;
state[0].RLast = state[0].RFirst;
```

The scanner may invoke any one of four subordinate methods. These are the `skip_separators`, `read_newline`, `add_rule`, and `is_reserved_word` functions. Each of these methods is discussed in turn. You may refer to the full implementation of the source code written in appendix B.

The `skip_separators` function is called each time before scanning processes the next token. In fact, before reading the relevant characters of the input, the scanner gets rid of all the separators like blanks, tabs, newlines, ...

```
void lexer::skip_seperators()
{
  while (((ch[i] == ' ') || (ch[i] == 9)
        || (ch[i] == 0) || (ch[i] == '\n'))
 && (eof != NULL))
    {
      if ((ch[i] == ' ') || (ch[i] == 9))  // blank or tab
        {
          ++i;
        }
      else
        {
          read_newline();
        }
    }
}
```

A prime consideration in lexical analysis is worth mentioning here, to put an effort into handling input/output(I/O) effectively. Because lexical analysis reads input a character at a time, it is essential to find the least-time consuming manner to buffer that input. In our implementation, we use the routine `fgets` that reads and stores an entire line of input in the global string variable `ch[LINE_LENGTH]`. There is a further efficiency gain using this routine instead of `getchar()` that might take more time to execute. When storing the line input into the array, characters are read sequentially by incrementing the variable index `i` corresponding to each entry of the array. If `ch[i]` was a blank or a tab, index `i` is incremented by 1. Else, if the character read was a newline, a call to `read_newline` is made to buffer a new line of input.

During the scanning process, the function `add_rule` is used to add a new rule to a state depending on the type of transition to be made. The new rule is passed as a parameter `new_rule` to the function, and the current state to get the new added rule is passed as a pointer address, or a reference because its new value has to be returned to the function `lex_analyser`. If the current state is empty `RFirst` and `RLast` point to the new rule. If it is not the case, this latter is added to the end of the rules set of this current state. Note that a rule is added to state if and only if it was not found in that state. For this purpose, a function `found` is used to search in the working state.

This function returns an integer value of 1 if the rule was found or else a value of 0 if the rule does not appear in the state.

```
void lexer::add_rule(STATE *state,RULEPTR new_rule) {
    if ( state->RFirst == NULL)
       {
         state->RFirst = new_rule;
         state->RLast = new_rule;
       }

  else {
    state->RLast->next = new_rule;
    state->RLast = new_rule;
  }
}
```

When we detect an identifier, we have to check if it is a reserved word or not. This has a primordial importance because, in case the identifier is a reserved word, the lexical unit is different for each of the other reserved words and no lexeme is associated. However, when the lexical unit is an identifier, the corresponding lexeme is the character string making up that identifier. The function `is_reserved_word` simply searches the table of identifiers for the obtained token. A binary search is done here for better performance in terms of time. The function returns a value of 1 if the identifier was a reserved word or a value of 0 of it was not the case.

```
int lexer::is_reserved_word(char *name)
{
  int i,j,k;

  j = last_reserved_word - 1;
  i = 0;
  do {
    k = (i+j)/2;
    if (strcmp(name,t_reserved_words[k]) > 0)
      i = k+1;
    else
      j = k-1;
  } while ((strcmp(name,t_reserved_words[k])) && (i <= j));
  token = k;
  return (!strcmp(name,t_reserved_words[k]));
}
```

3.4 Conclusion

The lexical analyser is often a convenient place to carry out many administrative chores. We saw how our scanner can skip separators, read input streams and detects tokens. Once in the appropriate detection, the scanner proceeds to process as much input as necessary in order to obtain the next tokens. A most common chore that was not discussed here is stripping out comments. Comments are often not formally defined in the language grammar. However, this is not the case in our lexical analyser. As this latter has to be general for all grammars, we have to specify the rules treating a comment in the grammar. In addition to stripping out comments as they are read in, lexical analysers are usually used to fill symbol tables and treat the type identifiers as well as the length of strings. Unfortunately, these tasks are left for the parsing process in our compiler. The reason will be discussed in later chapters.

Chapter 4

The Parser

4.1 Introduction

In the course of introduction, we saw that the purpose of the parser is to decide whether or not a given source program follows the syntax rules of the specified context-free grammar. This process of validating the syntactic correctness of a program is accomplished by reading the input stream and deciding whether the next input symbol can legally follow what has already been read. The parser obtains tokens from the input stream by calling the scanner. For certain tokens, it will also expect additional information to be available in a variable provided for this purpose. This happens whenever the input token is a number, a literal, or an identifier as was discussed in the description of the scanner(see Section 3.3).

The possibility for a syntax analyzer to examine the statements of a language is to start from a specific symbol. The idea is to replace, as going along, all non-terminal symbols with terminals, by applying the rules of production of the grammar in a judicious manner. The analysis of each statement is generally done from left to right performing left-hand derivations. Many different approaches to this problem exist as well as many based algorithms were known. Existing algorithms to parse only subsets of type 2 grammars, like LL(1), LR(1),..., were used to implement different compilers. As we are interested to build up a general compiler, the most efficient Earley's algorithm that parses any type 2 grammar is used in our case. In chapter 2, we have discussed in detail all the steps of this algorithm. However, this does not really represent a complete description of the algorithm until we describe in detail how all these operations are implemented on a machine. In this chapter, we concentrate on the full implementation of Earley's recognizer. Moreover, we discuss how we can alter the recognizer into a parser so that it can build a parse tree as it does the recognition process. After the implementation, a representation of the parse tree for the input must be considered. The sequence of productions used in all the derivations is assembled into a linked list structure to form the parse tree. We describe in detail all the steps made for the generation of the parse tree. In section 4.2 we look at the full description of the parser describing the data structures and all necessary functions used. In section 4.3 we look at the parse tree construction method used in the implementation.

4.2 Syntax Analysis Method

4.2.1 Top-down Description of the Parser

The parser determines whether or not an input program follows the syntax rules of a certain language. These rules of production of the language grammar are usually embedded in the implementation of most parsers. The reason is that each parser was implemented according to a specific language. However, this is not the case with our compiler. Since this latter is designed to accept most language grammars, the grammar specification must be written in a file that is read inside the implementation to make use of the productions in parsing each token. The `FillInitState` function is made up for this purpose. It stores those productions contained in the file, in a structured list for later use by the algorithm.

When storing the rules of production in the list structure, they must have a numerical representation. A rule is usually composed of a certain number of terminals and/or non-terminals. Each of these latters must be assigned a number that is stored in the list of rules instead of storing them as character strings. The reason is to increment the performance of Earley's algorithm when performing comparisons. In other words, when there is a new transition(prediction, scanning step, completion) to be made during analysis, many comparisons are performed to add new rules to a certain state. It is better to compare symbols represented as numbers and not as strings. The `add_non_terminal` routine is the one that gives the symbol of each production a numerical representation.

The parsing function `Parser` includes the full implementation of Earley's algorithm. This latter is supposed to have some modifications added to its transition steps. Actually, Earley's algorithm does not consider priority between operators in arithmetic expressions. A method should be introduced in the implementation to solve the problem. Another problem arises in the implementation is to store lexemes coming from different tokens for later use in the code generation. Our compiler does not generate code immediately when parsing the tokens. So, we have to keep all corresponding lexemes in the parse tree for later use. To clarify, those lexemes are used when traversing the tree to build the symbol table because this latter is left out to the code generation phase. This problem will be discussed in more detail in the next chapter. The function `Parser` can call two subordinate routines: `found`, `add_rule` as illustrated in the calling sequence of figure 4.1. The purpose of these routines are now discussed. Details about each routine will be described in detail later in this section.

The `found` routine checks if a new rule to be added to a certain state is already found or not.

The `add_rule` routine is used to add a new rule to a certain state. This will be done when a new transition occurs. You can refer to section 2.2 for more detail concerning the description of Earley's algorithm.

In the last portion of this analysis phase, after realizing that the input is syntactically well formed, it comes the construction and the representation of the parse tree. There are two basic types of representation we shall consider here, implicit and explicit. The sequence of productions used in some derivations is an example of an implicit rep-

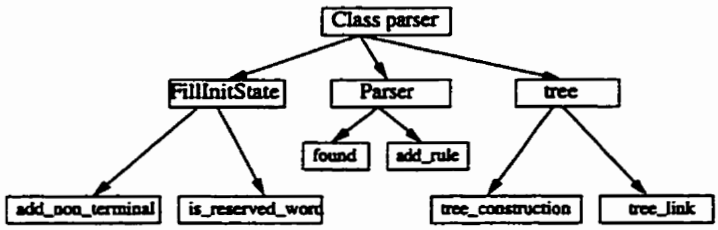


Figure 4.1: The parser sequence.

resentation. A linked list structure for the parse tree is an explicit representation. It is the function `tree` which performs this task of extracting and assembling together the derivations of the parse tree. This function can invoke two procedures, namely, `tree_construction` and `tree_link`.

The `tree_construction` routine constructs the linked list of productions corresponding to the right parse tree. This list will be extracted from the set of states generated by Earley's algorithm by modifying this latter to facilitate the task.

The function `tree_link` turns the list of productions generated by `tree_construction` into a tree like representing the complete derivations obtained from parsing the input.

4.2.2 Implementation of the *parser* Class

In this section, we consider in detail the implementation of each of the main functions of the parser, and their sequence previously defined. We start by the function `FillInitState` for storing the rules of production existing in the grammar specification file. There is no big difference between this function and the one described in the scanner. However, this one was designed to store the productions for syntax analysis while that one for the scanner was to save productions for the lexical analysis.

Opening the file grammar, we start by reading all productions. Each production has a left-hand side and a right-hand side. All productions are separated from each other, by the symbol `$$`. They are saved in the variable `InitRules` of type structure `struct STATE`. This structure was described in section 3.3.3. It holds two pointers `RFirst` ans `RLast` to the first and last rule. The variable `InitRules` is declared as private in the class *parser* and is used in the function `Parser`. The structure that holds the information about each rule is called struct RULE and it was described in the lexical analyser(see section 3.3.2). To remind `left_hs` is a variable that holds the left-hand side symbol of a rule. `pred_ptr` is a prediction pointer to the rule. It is first initialized to 0. `priority` is a number indicating the priority of each rule. It must be included and read from the grammar file. This variable is used to treat the priority between rules specifying arithmetic expressions. `right_hs` is a pointer to the right-hand side of a rule. It points to a list structure of the form

```
struct WORDNODE
```

```
{
  int word;
  char *value;
  char *value1;
  char *instruction;
  char *label;
  struct WORDNODE  *next;
  struct WORDNODE  *before;
  struct RULE *desc;
};
```

where `word` is a variable holding a symbol of the right-hand side of the rule. `value` holds the name of the lexeme, if any, corresponding to each token. `next` is a pointer to the next symbol. `before` is a pointer to the previous symbol. This pointer is reserved for later use to construct the parse tree from obtained derivations. `desc` is the descendant rule of each non-terminal. This variable is also kept for the construction of the parse tree. Other variables of the structure are used in the code generation part of the compiler. We will see in detail the purpose of each of these variables.

Some remaining variables are described in the structure `struct RULE` like `after_ptr`. This latter is a pointer indicating the next symbol of the right-hand side of a rule to be checked during analysis. `right_hs_tail` points to the tail of the right-hand side of a rule. `semantic_rule` is a pointer to the semantic part of rule. It is left for the code generation phase. Two important pointer variables are `ref_ptr` and `ref_ptr1`. These variables play a big role in extracting derivations of the right parse tree. As we mentioned that Earley's algorithm generates all derivations trees when there is ambiguity in the grammar language. So these two pointers are added in order to help in choosing one parse tree of all the possible derivations.

The `FillInitState` function is given here in a more detailed description. Given the language grammar in a file, we open this latter with a linked stream variable called `in`. We start reading the file input, symbol after another, with the variable `temp`. This latter is of type character string. Reading is performed in a loop until the end of the file is reached. For each rule, we allocate memory space to the structure `struct RULE`. This is done with the temporary pointer variable `tempRule`. In the loop, the first symbol read each time is supposed to be the left-hand side of the rule. We store it in `tempRule->left_hs` as an integer and not as a string. For this purpose, a call to the function `add_non_terminal` is made each time to give the symbol its numerical representation. The right-hand side of the rule is read in an inner loop that ends up once reading the `$$` sign. Its symbols are stored in the list structure `struct WORDNODE` pointed to by `tempWords`. This temporary variable is allocated space for each symbol of the right-hand side before storage and it is finally linked to the pointer variable `tempRule->right_hs`. Note that `tempRule->right_hs` is a doubly linked list, i.e., each of its nodes has two pointers to the next and previous one. The reason will be more explained in the parse tree construction. Once the rule is read and stored in `tempRule`. This latter is linked to `InitRules.RFirst` and/or `InitRules.RLast` depending whether it is the first or the last rule to be stored. The source code of

the FillInitState function look like the following. Note that a part of the code, concerning the semantic phase, was deleted here for better understanding.

```
void parser::FillInitState(void){

int index; // index for array
char temp[50];
struct WORDNODE  *tempWords,*tempLastWords;
struct RULE   *tempRule;
ifstream in("basic_gr");   // language grammar file

   InitRules.RFirst=NULL;
   InitRules.RLast=NULL;
   last_non_terminal = last_reserved_word;
   while (in >> temp)
    {
      tempRule = new struct RULE;
      tempRule->pred_ptr = 0;
      add_non_terminal(temp);
      tempRule->left_hs = code_value;
      tempRule->right_hs=NULL;
      tempRule->right_hs_tail=NULL;
      tempRule->after_ptr=NULL;
      tempRule->semantic_rule = NULL;
      tempRule->ref_ptr=NULL;
      tempRule->ref_ptr1=NULL;
      tempRule->next=NULL;
      in >> temp;
      while(strcmp(temp,"$$"))
        {
          tempWords = new struct WORDNODE;
          add_non_terminal(temp);
          tempWords->word =code_value;
          tempWords->value = NULL;
          tempWords->next=NULL;
          tempWords->desc = NULL;
          if(tempRule->right_hs==NULL)
            {
              tempWords->before = NULL;
              tempRule->right_hs=tempWords;
              tempLastWords=tempWords;
              tempRule->after_ptr=tempWords;
            }
          else
            {
```

```
                tempWords->before = tempLastWords;
                tempLastWords->next=tempWords;
                tempLastWords  =tempWords;
              }

            in >> temp;
          }

        if(tempRule->right_hs != NULL)
          tempRule->right_hs_tail = tempLastWords;
                  tempWords->next=NULL;
            if(tempRule->semantic_rule==NULL)
              {
                tempRule->semantic_rule=tempWords;
                tempLastWords=tempWords;
              }
            else
              {
                tempLastWords->next=tempWords;
                tempLastWords  =tempWords;
              }
            in >> temp;
          }

        in >> temp;
        tempRule->priority = atoi(temp);

        if(InitRules.RFirst==NULL)
          {
            InitRules.RFirst=tempRule;
            InitRules.RLast=tempRule;
          }
        else
          {
            InitRules.RLast->next=tempRule;
            InitRules.RLast=tempRule;
          }
      }
}
```

The `add_non_terminal` routine is used to give each symbol of a rule a numerical representation. We have explained how we represent symbols(reserved words, terminals, non-terminals) of the grammar by storing them in the table `t_reserved_words` (see section 3.3.2). The third part of the table `t_reserved_words` was left for non-terminals. This routine checks if the symbol read is a reserved word in the language. It looks up

in the first part of the table t_reserved_words. The function is_reserved_word is called to perform the job. If the search was successful, this latter function returns the numerical representation of the symbol in a variable called code_value, declared as private for the whole class parser. It is this value code_value that is assigned to tempRule_left_hs in the function FillInitState. If the symbol was not a reserved word, it is checked in the second part of the table t_reserved_words. It it was found, it means that it is a terminal. Its value is also passed to FillInitState through the variable code_value. If the symbol read was not matched, we realize that it is a non-terminal. Thus, we search if it was already saved in the third part of t_reserved_words. If it already exists, we can extract its numerical value being the index of the entry in which it was saved. It it was not the case, we add it to a new entry in the table.

```
void parser::add_non_terminal(char *symbol)
{
  int found = 0;

  if (!is_reserved_word(symbol))
    {
      for(code_value = last_non_terminal;
        code_value < nb_reserved_words;code_value++)
        {
          if (t_reserved_words[code_value] == NULL)
            break;

          if (!strcmp(t_reserved_words[code_value],symbol))
            {
              found = 1;
              break;
            }
        }

      if (!found)
        {
          t_reserved_words[code_value]=(char*)malloc(strlen(symbol)+1);
          strcpy(t_reserved_words[code_value],symbol);
        }
    }
}
```

When the grammar rules are saved in the variable InitRules, the parsing procedure starts by analyzing the tokens. It is the function Parser, implementing Earley's algorithm, that performs the job. Basically, the parser algorithm works as follows. The table entries pointing to each rules set(state) are first initialized to NULL. Each entry

in the table `state[MAX_STATE]` is a structure of type `struct STATE` having `RFirst` and `RLast` as pointers to the first and last rule in a state. Then, the first entry of the table `state[0]` is given the starting rule that leads to other transitions. This rule is supposed to be the one first stored in the variable `InitRules.RFirst`. After initialization, the parser starts to read tokens sequentially in a loop that ends up at the end of the source program. The variable `symbol` saves the value of each token obtained by a call to `lex_analyser` in the class `lexer`. `value` is the variable that holds the value of the lexeme in case the token obtained was an identifier. Note that the function `lex_analyser` has two arguments `tokens` and `value`. The table `tokens[2]` holds the names of the tokens obtained by calling the scanner. There is a possibility to have more than one token for the same identifier. For example, the number 2 might be one of the two tokens `entier` or `decimal`. These two tokens are stored in the table `tokens`. The variable `value` contains the lexeme to the corresponding tokens, 2 in this case.

By obtaining the token each time in the loop, we perform the Earley parse. The temporary variable `tempRule` points to the first rule in a state. This rule is nothing but the value of `state[count].RFirst` where `count` is an index first initialized to 0, and incremented sequentially in the loop to point each time to the next state of rules. The analyzer proceeds as follows. While there is a rule not already read in the state, we perform any one of the three possible operations(completion, scanning step, prediction), depending on the symbol held after the prediction pointer. The value of each new rule to be added to a certain state, is saved in the pointer variable `tempRule1`. It is the function `add_rule` that adds `tempRule1` to that state after verifying if the new rule to be added does not already exists. The verification is done by a call to the function `found`.

In addition to the contents of the algorithm, we have made some modifications to the scanning step like saving the obtained lexemes corresponding to the rules containing terminals. When a scanning step occurs, if the current token is an identifier(terminal), we have to add its lexeme to the new rule added to the next state. For this purpose, we look it up in the right-hand side of the new rule, and we add the lexeme in the appropriate place. Note that saving lexemes with derivations is so important for later use in the symbol table because this latter was left out to the code generation phase.

A second important feature newly occurs in the algorithm, to find a way for getting the right parse tree from all possible derivations. Each time we perform a completer operation adding a new rule to a state, say $E \rightarrow \alpha D.\beta$, we construct a pointer `tempRule->ref_ptr1` from this new rule to that rule $D \rightarrow \gamma.$ which caused us to do the operation. This indicates that D was parsed as γ. Moreover, we save in a pointer `tempRule->ref_ptr` the address of the rule from which $E \rightarrow \alpha D.\beta$ was obtained, $E \rightarrow \alpha.D\beta$ in this case. This latter pointer is also used in the two other operations(prediction, scanning step), each time to keep a pointer to the rule that caused the new transition.

An additional important modification is the priority treatment between productions. Each rule in the state is supposed to have a priority number `tempRule->priority`. To remind, this number was indicated by the user and obtained from the grammar description file. In the analysis, suppose that we have the two rules $E \rightarrow E+E$ and $E \rightarrow E+E$. Each time a completer operation occurs, say $E \rightarrow E+E.$, 0,

adding a new rule to a state, we search if this rule is already found in the state. If the search was successful, it means that a previous completer operation, say $E \rightarrow E*E.,0$, has caused the addition of this rule, and a pointer from this latter was constructed to the old completed production. If the new completer has a higher priority number, we alter the reference of the added rule to point to this new completer. In this way, we can assure that the rule $E \rightarrow E+E$ will appear in a higher level in the parse tree and $E \rightarrow E*E$ will be evaluated before the execution of $E \rightarrow E+E$. You can see in detail the source code of the function `Parser` in the appendix .

4.3 The Parse Tree

The analysis step that follows, is to build the right parse tree from all possible derivations in the set of states . In this section, we discuss the problem of the tree construction and linking, by explaining all the necessary related functions.

Once the input is successfully read and no errors have occurred in the analysis, we must reach the terminating state. This latter is expected to have the initial rule, i.e. the rule first saved in `InitRules.RFirst`, from which the analysis has started. In the function `tree`, the initial rule is first saved to a temporary variable called `tempRule`. A call to the routine `found` is made to check if this initial rule, i.e. `tempRule`, is found in the terminating state. If so, the analysis is thus successful and this allows the function `tree_construction` to be called to extract the right derivations of the tree.

The `tree_construction` routine is a recursive function. It has three main arguments, namely `Trule`, `tempRef`, and `rule`. Starting from the initial rule found in the last state and first saved in `Trule`, the task of this function is to visit all referenced rules in the states set. To remind, from each rule in the set of states , we kept a pointer to the rule that caused its addition to a certain state. In each function call, `Trule` is supposed to have the new derivation referenced by the rule before. A verification is made to check whether the content of `Trule` is a completer or not. If so, the derivation is therefore a rule of the parse tree. We have to save it in the linked list of the tree derivations pointed to by `rule`.

When the linked list structure for the tree derivations is obtained, we have to link together all these rules in a tree structure form. For this purpose, the pointer variable `desc` in the structure `struct WORDNODE` was introduced so that each non-terminal of a certain derivation can point to its descendant rule. It is the recursive function `tree_link` that constructs from the obtained list of derivations this parse tree. It starts with the first rule in the list pointed to by `rule` and the last symbol on the right-hand side of this rule pointed to by `right_hs_tail`. This symbol is checked in turn to see whether it is a non-terminal or not. If it was a non-terminal, the next rule to follow in the list is supposed to be its descendant rule. Thus, this non-terminal is linked to its descendant rule with the pointer variable `desc`. The current rule structure in the list must now be altered to point to the rule that follows the one turned out to be a descendant of the non-terminal. We proceed in this way with the descendant rule of this non-terminal until no more non-terminals are found. Then, we go back to the symbol found next to last non-terminal to connect it to its descendants and so on. To

illustrate with an example, suppose that we have the following set of rules obtained from `tree_construction` in this form

Rule 1: programme → enonce0_n terminal
Rule 2: terminal → numero_ligne end
Rule 3: end → END
Rule 4: numero_ligne → entier
Rule 5: enonce0_n → enonce enonce0_n
Rule 6: enonce → numero_ligne donnee
⋮

In order to build our tree, we have to alter this linked list in a tree structure form. We start by a the first call to `tree_link` with the first rule, i.e *Rule 1*, and by the last symbol of this rule, *terminal* in this case. We see that *terminal* is a non-terminal symbol and it must have the next rule in the list, say *Rule 2* as its descendant. We link *terminal* to *Rule 2*. Now *Rule 1* must point to *Rule 3* as its next rule. A recursive call is made with the descendant rule of *terminal*, i.e. *Rule 2*. We check again for the symbol *end* and since it is a non-terminal, *Rule 3* must be linked as its descendant rule. Now *Rule 1* must point to *Rule 4*. Again a recursive call is made with the descendant rule of *end*, i.e. *Rule 3*. Since the symbol *END* is a terminal, we realize that no other descendants are found in this way. We go back to the left symbol of *end* in *Rule 2*, i.e. *numero_ligne*. It is a non-terminal so the rule that follows is supposed to be its descendant, say *Rule 4*. We proceed in this way until no more rules are found in the linked list. Thus we obtain the complete parse tree.

```
void parser::tree_link(RULEPTR *rule, struct WORDNODE **right_hs_tail)
{

  if ((*right_hs_tail != NULL) &&  (*rule != NULL))
    {
      if (((*right_hs_tail)->word >= 0)
          && ((*right_hs_tail)->word < last_terminal))
        {
          tree_link(&(*rule),&(*right_hs_tail)->before);
        }
      else
        {
          (*right_hs_tail)->desc = (*rule)->next;
          tree_link(&(*rule)->next,
                                  &(*right_hs_tail)->desc->right_hs_tail);
          (*rule)->next=(*rule)->next->next;
          tree_link(&(*rule),&(*right_hs_tail)->before);
        }
    }
}
```

4.4 Conclusion

This parser will probably be most useful in all general compilers where the full power of context-free grammars is used. In most other compiler systems, the programmer is allowed to express the syntax of his language in something like BNF, and the system uses a parser to analyze subsequent programs in this language. Programming language grammars tend to lie in a restricted subset of context-free grammars which can be processed efficiently. However, some compiler writing systems in fact use general parsers, so this parser may be of use here. In addition, to its efficiency properties, ours has the advantage that it accepts the grammar in the form in which it is written, so that semantic routines can be associated with productions. One question still remains. If the compiler receives a syntactically erroneous program to be compiled, what should be done when the parser encounters an error? This question is discussed in the next chapter.

Chapter 5

Contextual Analysis and Error Handling

5.1 Introduction

We now have a program capable of reading an input file containing any language program and verifying if its statements do comply with the rules of production of the defined language grammar. Unfortunately, we can see that a programming language has some constraints that we can not formalize using the rules of production. One of these constraints, for example, is that we cannot use an undeclared identifier in a certain instruction. Another example occurs in a function call. In this case, effective and formal parameters must correspond to each other by number and by type. More generally, all these constraints are not treated with the rules of production. The purpose of the contextual analysis phase is precisely to proceed in all the verifications of the rules imposed on the language definition but not formalized by the rules of production. We classify these controls to be carried out in many themes.

- Holding identifiers
- Type analysis
- Conformity of formal and effective parameters

In this chapter, we study all the modifications brought to the syntax analyzer to help in error recovery as well as to incorporate the contextual analysis. Note that not all the above themes were treated by implementation. For example, detecting undeclared identifiers during analysis is not an easy job to perform with Earley's parser. Therefore, we discuss some important topics that we already included in our implementation as well as we introduce others that might be added in future development. One of the primary issues discussed is how to hold identifiers in the symbol table for later use in the analysis. In section 5.2, we discuss this topic by describing the format of entries, the method of access, and the place where identifiers are stored in the symbol table. In section 5.3. we introduce the method of types control and the problem concerning the conformity of formal and effective parameters. In section 5.4, we discuss all types of errors that can be detected in our compiler.

5.2 Holding Identifiers

Holding an identifier designates the part(s) of the program text in which that identifier has the same meaning that has been given to it at the time of its declaration. Given a language having the block structure(i.e. the main block and the function block), the notion of holding identifiers is deeply bound to this structure of blocks. We must hold an identifier from the time of its declaration till the end of the block in which it is defined or used. Inside the same block, all identifiers must be unique. We say that they are local to this block. If a program contains nothing but only one block, namely the main block, holding identifiers like constants, variables is not interrupted and remains from the time of their declaration till the end of the program. We say that these identifiers are global.

The programmer can specify new blocks by defining new functions. The same rules described above apply to function blocks. When a function block defines a new identifier and this latter has been previously defined at the main block level, holding that identifier with the same name in the main block is interrupted in all the duration of the current function block.

Figure 5.1: Holding a global identifier I is interrupted by the block declaring an identifier with the same name.

On the other hand, holding a global identifier not redefined inside a function block comprises this function block. In other words, this global identifier is accessible inside this function block. The contrary is not true, i.e. an identifier defined in a function block, is never accessible outside its function block.

In the study of holding identifiers, the analyzer must verify if the use of an identifier is legal or not, i.e. if this identifier has been previously declared or not and if it is unique in its block. Moreover, because the period of holding an identifier lasts from the time of its apparition till the end of the block defining it, we must verify if the program contains recursive functions. Note that our compiler does not verify undeclared variables because it is implemented indeed only for variables declared when assigned a value, e.g., a variable is declared as an integer if it was assigned an integer value.

To realize this work, the analyzer must keep all the symbols, that it meets from the beginning of the text analysis, in an adequate structure called *symbol table.*

The symbol table is the place where the compiler keeps the names of identifiers and their associated information, when met in the source program. The operations performed on the symbol table are addition, deletion, and the search for a symbol. The structure of a symbol table may be too simple or even so complex and it depends

on the nature of the compiler. For instance, it may be implemented as a simple linear array, a treelike data structure, or using hashing techniques. Different forms for the representation of a symbol table are discussed in [1, 13, 14].

The structure of the symbol table we are adapting in our compiler is simple. It is a linear table considered as a stack-based array. The stack structure is adequate in this case because entering a new function block, identifiers which are local to this block are placed at the top of the stack. While leaving the block, these identifiers are popped from the stack.

For purpose of not loosing unnecessary memory space, the table does not contain in fact except pointers to symbols. The symbols have different structures according to their types(constant, variable, array,...). The structures when implemented are nothing but records with variables.

```
typedef struct t_ident_record
{
  char *name;
  int type;
  union id_info
  {
    struct variable
    {
      int v_typ;
      int v_adr;
    } variable;
    struct fun_par
    {
      int p_typ;
      int p_adr;
      int next;
    } fun_paramter;
    struct function
    {
      int param;
      int f_adr;
      char *tag;
    } function;
    struct constant
    {
      float cons;
    } constant;
    struct array
    {
      int el_typ;
      int length;
    } array;
```

```
  } id_info;
} *t_ident_record_ptr;
```

When the symbol table is first initialized, the different pointers of the stack receive their initial values. All the table entries are set to NULL. In fact, the symbol table is divided into three distinct zones where uniquely the part reserved for symbols belonging to function blocks behaves as a stack(figure 5.2).

```
void parser::InitSymbolTable(void)
{
  int i;
  last_identifier = -1;
  last_main_block = -1;
  last_parameter = max_identifiers;
  current_block = MAIN;
  for(i=0;i<max_identifiers;i++)
    t_symbol_table[i] = NULL;
}
```

During the analysis of the source program, global symbols are first analyzed and placed in sequential order according to their apparition. `last_main_block` indicates the address of the last global symbol. In a function block, all defined symbols are placed sequentially in their order of apparition after the symbol whose address is given by `last_main_block`. The last symbol in a function block is indicated by `last_identifier`. After examining a function block, all its symbols are popped. The formal parameters for each function must not be destroyed with the rest of local symbols. In fact, as we will see later, the compiler has to perform a validity control for the conformity of local and formal parameters. For this reason, they are added to the other part of the table. It is trivial that more efficient data structures could have been chosen to implement these categories of symbols. We were not able to use one of those data structures in order to facilitate the work. Modifications are left for interested programmers.

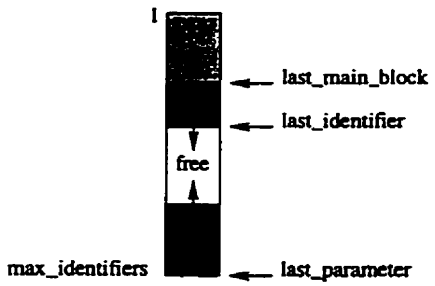


Figure 5.2: The symbol table structure.

In most compilers, when analyzing symbols, it is easy to distinguish identifiers before saving them to the symbol table. The reason is that those compilers were designed for a

specific grammar thus programmed in a way to distinguish all symbols of that grammar. In contrast, since we have a general compiler that fits for most language grammars, we chose to include the specification of identifiers in the grammar file. In this way, we can easily verify the nature of each identifier. As an example, suppose we have the following specification for the BASIC language grammar

```
%%variable    variable              : variable_simple
%%variable    -                     : variable_simple
%%function    def                   : lettre
%%function1   entete                : lettre
%%array       dim_tableau           : lettre
  ⋮              ⋮
```

In the first line of the specification, `%%variable` is the nature of the identifier, `variable` is a symbol of the left-hand side of a rule, and `variable_simple` is a symbol of the right-hand side of a rule. The specification of identifiers is saved in a table called `t_id_specification[12]` where each entry is a record containing the specification of an identifier. The record structure looks like the following:

```
typedef struct Id_specification
{
  int specification,ident;
  int specification1,ident1;
} ID_SPECIFICATION ;
```

where `specification` contains the left-hand symbol of a rule and `ident` contains the right-hand symbol. The variables `specification1` and `ident1` are used when the same identifier may occur in two different rules. Traversing the parse tree, each rule is compared with the entries of the table `t_id_specification`. If the left-hand and right-hand symbols correspond to a certain entry in the table, the identifier is saved in the symbol table according to its specified nature. Note that the index of each entry in the table `t_id_specification` has a specific meaning indicating the nature of the identifier(variable, array, ...).

Introducing an identifier in the symbol table is performed by a call to the function `define`.

```
void parser::define(char ident[],int kind)
{
  int index;

  t_ident_record_ptr id_record;
  if (last_identifier + 1 < last_parameter)
    if (!exist_in_current_block(ident,&index))
      {
```

```
        last_identifier++;
        if (current_block == MAIN)
          last_main_block++;
        id_record = new struct t_ident_record;
        id_record->name = (char*)malloc(strlen(ident)+1);
        strcpy(id_record->name,ident);
        id_record->type = kind;
        if(kind ==VARIABLE)
          {
            current_local_variable = last_local_variable;
            v_type2 = UNKNOWN_TYPE;
            id_record->id_info.variable.v_adr = last_local_variable++;
          }
         t_symbol_table[last_identifier] = id_record;
      }
    else
      {
        if(kind == VARIABLE)
          {
            current_local_variable =
              t_symbol_table[index]->id_info.variable.v_adr;
            v_type2 = t_symbol_table[index]->id_info.variable.v_typ;
          }
      }
}
```

This function first verifies if the table contains enough space, i.e., it determines if the introduction of a new symbol does not collide with the parameters symbols on the other part of the table. If there is enough space in the table, we verify if the symbol introduced does not already exist. If not, the pointers of the table are adjusted and the new symbol is given an entry according to its nature. In the case where the symbol is a variable, the information saved in the symbol table is the type of the variable which is unknown at the beginning since we does not know the type unless a value is assigned to the variable. When the variable occurs in the parse tree for the second time, i.e., it is assigned a value, we modify the type of that variable in its entry of the table. A variable is inserted in the symbol table by a call to the function `define_variable`.

```
void parser::define_variable(char ident[])
{
  define(ident,VARIABLE);
}
```

In the definition of an array, we have to keep the type of the table and the number of elements in this table. For a function identifier, we have to initialize the list of formal parameters. How can we insert a parameter in the table? We must first check if there

is enough space in the table and if the symbol name is unique in the block. Then, it is added to the end of the list of formal parameters of the function. As we have seen previously, in order to be able to validate the number of effective parameters during a function call, the list of formal parameters must remain in contrary to the local symbols of the function which are destroyed at the end of the function analysis.

To manage properly the symbol table, we assume that the analyzer checks if a currently treated identifier is a part of the main block. To do this, a call to the functions `enter_function_block` and `exit_function_block` is done. The first function is executed by entering the function block, thus assuring that symbols between two different blocks are separated. The variable `current_block` indicates the block currently working. By exiting a function block, this variable is altered and all symbols of the block are destroyed and their memory space is freed.

All symbols in the same block must be unique. The function `exist_in_current_block` informs us if an identifier already exists in its block. For this fact, we can deduce if the current identifier is unique or not.

```
int parser::exist_in_current_block(char ident[],int *index)
{
  if (current_block == MAIN)
    return (is_global_symbol(ident,index));
  else
    return (is_local_symbol(ident,index));
}
```

This function is based on two other functions `is_global_symbol` and `is_local_symbol` each of which searches sequentially the set of global symbols, respectively the set of local symbols of the function block. Searching the set of global symbols is not that difficult. However, to determine if a declared identifier in a function block is unique, it is not only sufficient to search the stack of local identifiers, but we have to examine the list of formal parameters.

```
int parser::is_global_symbol(char ident[],int *index)
{
  int found = 0;
  *index = 0;
  while ((*index <= last_main_block) && (! found))
    if (!strcmp(t_symbol_table[*index]->name,ident))
      found = 1;
    else
      (*index)++;

  return found;
}
```

In fact, the identifier designating the function currently analyzed is the last global identifier(identifiers are added sequentially in the symbol table, in their order of apparition in the input program), and consequently can be found at the address `last_main_block`.

We have seen above that for each symbol in the symbol table, an additional information is associated depending on the nature of the symbol analyzed. In the case of a function identifier, such information is the list of formal parameters saved at the second part of the table. The last important function in this context is the function `is_defined_symbol` which verifies if an identifier given as an argument is defined at the time when it was called by the analyzer. This function returns as an argument the address of the identifier in the symbol table.

The integration for an analyzer to verify the unicity of identifiers in the same block, and to verify the existence of an identifier undeclared previously is not a difficult task to perform. We have to be aware not to save an identifier incorrectly, to avoid erroneous recursive calls for the identifier of a function.

5.3 Type Control and Conformity of Effective/Formal Parameters

We saw that our compiler does not treat the problem of undeclared identifiers in the case of type languages. But how if we want to find a solution to this problem? In this section, we discuss some of the problems to be solved in the case of controlling the validity of type identifiers. A compiler should perform a strict control on the performance of types manipulated in each statement of the program. We say that two types are compatible if they are declared with the same type identifier.

The type verifications must be done in many places and are generally more restrictive than the control types compatibility. For instance, in arithmetic expressions, the only type allowed between identifiers is somehow numeric:

- In an assignment instruction, the left and right parts must be of compatible types.
- Operands on the left and the right sides of a binary operator must be numeric.
- The operand of a unary operator must be numeric.
- When accessing an element of a table, we have to be sure that the accessing variable has the same type of the entry in the table.

What really complicates the job in operations is the fact that the compiler does not have to stop when detecting the first error but to proceed in analyzing the rest of the input program. If this latter contains a list of variables of the same type for example, and the type was missing or not well written, how can we stop the compiler from generating a large number of errors and to keep on analyzing the remaining statements in the program? In fact, the solution is to introduce a fictive type that we can give to objects not having the required type. Moreover, the analyzer must perform a contextual control on the nature of identifiers being treated. For example, when

accessing a variable, the identifier must not be a constant, nor a program name, nor a type name. The reader interested in the study of contextual verifications can refer to [12].

The validity control for the correspondence between formal and effective parameters is treated in an ad hoc manner. Many verifications must be done:

- The number of effective parameters must be equal to that of the formal parameters.
- If a formal parameter is to be passed by reference, the effective parameter must have access to this reference.
- If a formal parameter is to be passed by value, the effective parameter may be an expression.
- The types of a formal parameter and an effective parameter must be the same.

To perform these controls, the analyzer sets the list of formal parameters of the current function. To remind, this list is not destroyed at the end of the function block analysis to make these controls possible. During analysis of a function call, we have to pass the symbol table entry for this function identifier, to the function that treats effective parameters. This function `id_function` has access to the list of formal parameters saved in the table. `table_entry` is a function returning a pointer to the block of information associated to a table entry. When accessing this block of information, the analyzer can have access to the address of the first parameter in the list of formal parameters. This facilitates the task of the contextual analysis.

5.4 Error Handling

When a program is submitted to a compiler, there is always a possibility to have errors as is reasonably possible. Thus, with regards to errors, the compiler should print messages to the user describing the nature of the error detected. We can identify four principle types of error which a compiler has to handle.

- *Lexical errors* These are errors in the lexemes making up the program, usually mistyping of keywords, e.g. WHILE entered as HWILE.
- *Syntax errors* These are errors in the overall structure of the program. Common examples are omitting a semicolon between statements or forgetting a closing block bracket.
- *Compile-time semantic errors* Some semantic errors can be discovered before the program is run. The commonest examples of this are type errors, e.g. using a float as a character or trying to call a variable name as a function.
- *Run-time semantic errors* Some errors occur only when the program is executing, e.g. dividing by zero or executing the end of an array in writing.

The compiler must be capable of handling as many of these errors, even if it cannot as a result generate any valid code. After detecting an error, it must recover so that it can parse the rest of the program to detect further errors. Moreover, the compiler must beware of delivering error messages that are a consequence of earlier errors.

Lexical and syntax errors are closely related. In particular, the correction of a lexical error may lead to a syntax error if care is not taken. Since our compiler is designed for most language grammars, we were able to detect lexical and syntax errors in a restricted way. Lexical errors are very simple and mean that the lexical analyzer cannot match the stream of input characters because on of these latters was not found when it was supposed to match a character in the rules of production describing the tokens. The compiler thus returns a message stating that the lexical analyzer could not recognize a token. Unlike other compilers, when there is a non-closing comment or unterminated comment, our compiler does give a message detecting a lexical error, but it does not specify its type. The reason is that the treatment of comments was done by including the rule describing a comment in the file description of tokens. More specification could be added to this file in order to specify the type of lexical errors. This work is left for more interested programmers.

The first action on detecting a syntax error is to report it. Then, we must recover, so that any further errors can be detected, but avoiding cascade of derived errors. There is an approach to discard input tokens until we reach a token that signifies a consistent position in the rules of production. This method is simple and cannot get stuck in a loop since it always consumes input tokens. However, when an input statement of the program has a number of tokens that exceeds that of its corresponding rule of production, it is so difficult to recover from that error with the Earley's parser. So, when only tokens are missing, recovery is possible since we can advance pointers in all the rules of production until we match the next token in a statement.

The two remaining type of errors are the *compile-time* and *runtime semantic* errors. These errors are not treated in our compiler due to some difficulty with Earley's parser. Further study can be done to treat the detection of such types of error. you can refer to [3] for more reading on this issue.

Chapter 6

Code Generation

6.1 Introduction

Having completed the big part of our compiler, namely the syntax and semantic analysis of the source program, we wish to generate the equivalent target program. As we know, this occurs in two stages, first to an intermediate representation, then to the target code. The techniques involved in the generation of the intermediate representation are often very similar to those involved in the generation of the target language. A new technique is used in our compiler: the intermediate representation is included in the grammar description file. We classify it into a tree-based representation. It is closely related to the syntactic structure of the language, and we will cover it in detail in later sections.

In this chapter, we are concerned with the generation of the target machine code from the intermediate representation above. Our aim is to produce as efficient a target program as possible. In later sections, we look at the basic techniques to produce our target program but before, a question can be asked here: For what machine and for which processor we are going to generate this code? In fact, as we have already mentioned in the course of introduction, we do not generate bytecode for a real microprocessor in order not to be lost with specific and technical details of a particular architecture. The compiler we are building generates Java bytecodes for a virtual machine: a stack based abstract machine. Because it is not a really existing machine, how can we execute a program on a personal machine? The answer has already been given in the introduction: it is sufficient to build an interpreter for the Java virtual machine. In this way, we will have the following advantages:

- The compiler is more easily portable from one machine to another. The code generation must not be modified in the interpreter because it is written in a higher level portable language.
- The set of instructions of the Java virtual machine is already defined and created by the Java team. It is not be influenced by the different machines architectures but it is designed so that the virtual machine can be constructed in an easy way.

- The compiler can be easily modified so that it can produce native code for the target machine.

The main disadvantage of this technique is the slow execution of these bytecodes because these latters will be interpreted. This technique is notably used to build the java compiler that produces the Java portable code for an abstract machine[10, 7].

We begin the study by describing the Java virtual machine and the bytecode instructions that we used in our compiler. This is discussed in section 6.2. We progress to enter in detail the description of the code generation method in section 6.3. We then show an example of Java bytecodes generated from a simple program written in the BASIC language.

6.2 The Java Virtual Machine

6.2.1 Description

The Java virtual machine is intended to present an abstract, logical machine whose design is free from the details of any implementation. This virtual machine runs a special set of instructions called bytecodes that are simply a stream of bytes, each of which has precise specification of what exactly each bytecode does to this virtual machine. The java virtual machine can be divided into five fundamental parts:

- A bytecode instruction set
- A set of registers
- A stack
- A garbage-collected heap
- An area for storing functions

Some of these might be implemented by using an interpreter, a native binary compiler, or even a hardware chip but all these abstract components must be supplied in some form in the system.

The Java virtual machine instruction set is optimized to be small and compact. It is designed in a way that it can be moved across the Net with enough security. As mentioned, any language source code is compiled in bytecodes and stored in a file. These bytecodes are a low-level format that cannot be run directly, but must be further interpreted by each computer. Of course, it is exactly this level that gives us the portability of java code.

A bytecode instruction consists of one-byte opcode that serves to identify the instruction involved and zero or more operands, each of which may be more than one byte long, that encode the parameters the opcode requires. Bytecodes interpret data in the run-time memory areas as belonging to a fixed set of types: the primitive types consist of several signed integer types(8-bit `byte`, 16-bit `short`, 32-bit `integer`, `64-bit long`),

one unsigned integer type(16-bit `char`), and two floating-point types(32-bit `float`, 64-bit `double`)). The primitive types are distinguished and managed by the compiler and not in the Java run-time environment. So these types are not tagged in memory. Different bytecodes are designed to handle each of the various primitive types uniquely, and the compiler, based on its knowledge of the actual types, chooses the convenient bytecode corresponding to the type treated. For example, when adding two integers, the compiler generates an `iadd` bytecode; for adding two floats, `fadd` is generated. We will see all these bytecodes in detail later.

The registers of the Java virtual machine are just like the registers inside a real computer. They hold the machine's state, affect its operation, and are updated after each bytecode is executed. The following are the Java registers:

- pc, the program counter, which indicates what bytecode is being executed.
- `optop`, a pointer to the top of the operand stack, which is used to evaluate all arithmetic expressions.
- `frame`, a pointer to the execution environment of the current method or function, which includes an activation record for this call.
- `vars`, a pointer to the first local variable of the currently executing method.

The virtual machine defines these registers to be 32-bits wide.

The Java virtual machine is stack-based. A Java stack frame holds the state for a single function call. Frames for nested function calls are stacked on top of this frame. The stack is used to supply parameters to bytecodes and methods, and to receive results back from them. Each stack frame contains three sets of data: the local variables for the function call, its execution environment and its operand stack. Local variables are stored in an array indexed by the register `vars`. The execution environment in a stack frame helps to maintain the stack itself. It contains a pointer to the previous stack frame, a pointer to the local variables of the function call, and pointers to the stack's current *base* and *top*. The operand stack, a FIFO stack, is used to store the parameters and return values of most bytecode instructions. For example, the `iadd` bytecode expects two integers to be stored on the top of the stack. It pops them, adds them together, and pushes the resulting sum back onto the stack.

The heap is a part of the memory from which newly created objects are allocated. It is often assigned a large, fixed size when the Java run-time system is started, but on systems that support virtual memory, it can grow as needed.

The method area stores the Java bytecodes that implement almost every function or method in the system. It also stores the symbol tables needed for dynamic linking, and any other additional development environments which might want to associate with each function's implementation.

6.2.2 Bytecode Instructions

For each bytecode, there is a brief text that describes its function, and a picture form of the stack, before and after the bytecode has been executed, is shown. This picture

will look like the following:

..., `value1`, `value2` ⇒ ..., `value3`

This means that the bytecode expects two operands, `value1` and `value2`, to be on the top of the stack, pops them both off the stack, operates on them to produce `value3`, and pushes `value3` back onto the top of the stack. You should read each stack from right to left, with the rightmost value being the top of the stack. The ... is read as the rest of the stack below, which is irrelevant to the current bytecode. Bytecodes that do not affect control flow simply move the pc onto the next bytecode that follows in sequence. Those that do affect the pc say so explicitly. Whenever you see `byte1`, `byte2`, and so forth, it refers to the first byte, second byte, and so on, that follows the opcode byte itself. After such a bytecode is executed, the pc automatically advances over these operand bytes to start the next bytecode in sequence. We describe a part of the bytecode instructions used in our compiler. For more detail about java bytecodes, you can refer to [7, 6].

Pushing Constants onto The Stack

`bipush` ... ⇒ ...,`value`

Push one byte signed integer. `byte1` is interpreted as a signed 8-bit value. This value is expanded to an `int` and pushed onto the operand stack.

`sipush` ... ⇒ ...,`value`

Push two-byte signed integer. `byte1` and `byte2` are assembled into a signed 16-bit value. This `value` is expanded to an `int` and pushed onto the operand stack.

`ldc1` ... ⇒ ...,`item`

Push `item` from constant pool. `byte1` is used as an unsigned 8-bit index into the constant pool of the current class. The `item` at that index is resolved and pushed onto the stack.

`iconst_<I>` ... ⇒ ..., `<I>`

Push the `int` `<I>` onto the stack. There are six of these bytecodes, one for each of the integers 0-5: `iconst_0`,`iconst_1`,`iconst_2`,`iconst_3`,`iconst_4`,and `iconst_5`.

`fconst_<F>` ... ⇒ ..., `<F>`

Push the `float` `<F>` onto the stack. There are three of these bytecodes, one for each of the integers 0-2: `fconst_0`, `fconst_1`, and `fconst_2`.

`dconst_<D>` ... ⇒ ..., `<D>-word1, <D>-word2`

Push the `double <D>` onto the stack. There are two of these bytecodes, one for each of the integers 0 and 1: `dconst_0`, and `dconst_1`.

Loading Local Variables onto the Stack

`iload` ... ⇒ ...,`value`

Load `int` from local variable. Local variable `byte1` in the current Java frame must contain an `int`. The value of the variable is pushed onto the operand stack.

`iload_<I>` ... ⇒ ...,`value`

Load `int` from local variable. Local variable `<I>` in the current Java frame must contain an `int`. The value of the variable is pushed onto the operand stack.There are four of these bytecodes, one for each of the integers 0-3: `iload_0`,`iload_1`,`iload_2`, and `iload_3`.

`lload` ... ⇒ ...,`value-word1, value-word2`
`lload_<L>` ... ⇒ ...,`value-word1, value-word2`
`fload` ... ⇒ ...,`value`
`fload_<F>` ... ⇒ ...,`value`
`dload` ... ⇒ ...,`value-word1, value-word2`
`dload_<D>` ... ⇒ ...,`value-word1, value-word2`

Storing Values into Local Variables

`istore` ..., `value` ⇒ ...

Store `int` into local variable. `value` must be an `int`. Local variable `byte1` in the current Java frame is set to `value`.

`istore_<I>` ..., `value` ⇒ ...

Store `int` into local variable. `value` must be an `int`. Local variable `<I>` in the current Java frame is set to `value`. There are four of these bytecodes, one for each of the integers 0-3: `istore_0`,`istore_1`,`istore_2`, and `istore_3`.

`lstore` ..., `value-word1, value-word2`⇒ ...
`lstore_<L>` ..., `value-word1, value-word2`⇒ ...
`fstore` ...,`value` ⇒ ...
`fload_<F>` ...,`value` ⇒ ...
`dstore` ...,`value-word1, value-word2` ⇒ ...
`dstore_<D>` ...,`value-word1, value-word2` ⇒ ...

`iinc` -no change-

Increment local variable by constant. Local variable `byte1` in the current frame must contain an `int`. Its value is incremented by the value `byte2`, where `byte2` is treated as a signed 8-bit value.

Managing Arrays

`newarray` ..., `size` ⇒ `result`

Allocate new array. `size` must be an `int`. It represents the number of elements in the new array. `byte1` is an internal code that indicates the type of array to allocate(boolean, float, char, ...).

`iaload` ..., `array`, `index` ⇒ ..., `value`
`laload` ..., `array`, `index` ⇒ ..., `value-word1`, `value-word2`
`faload` ..., `array`, `index` ⇒ ..., `value`
`daload` ..., `array`, `index` ⇒ ..., `value-word1`, `value-word2`

Load <type> from array. `array` must be an array of <type>s. `index` must be and `int`. The <type> value at position number `index` in `array` is retrieved and pushed onto the top of the stack.

`iastore` ..., `array`, `index`, `value` ⇒ ...
`lastore` ..., `array`, `index`, `value-word1`, `value-word2`⇒ ...
`fastore` ..., `array`, `index`, `value` ⇒ ...
`dastore` ..., `array`, `index`, `value-word1`, `value-word2` ⇒ ...

Store into <type> array. `array` must be an array of <type>s. The <type> value is stored at position `index` in `array`.

Arithemtic Operations

`iadd` ..., `v1`, `v2` ⇒ ..., `result`
`ladd` ..., `v1-word1`, `v1-word2`, `v2-word1`, `v2-word2` ⇒ ..., `result`
`fadd` ..., `v1`, `v2` ⇒ ..., `result`
`dadd` ..., `v1-word1`, `v1-word2`, `v2-word1`, `v2-word2` ⇒ ..., `result`

`v1` and `v2` must be <type>s. The vs are added and replaced on the stack by their <type> sum. <type> is, in turn, `int`, `long`, `float`, and `double`.

`isub` ..., `v1`, `v2` ⇒ ..., `result`
`lsub` ..., `v1-word1`, `v1-word2`, `v2-word1`, `v2-word2` ⇒ ..., `result`
`fsub` ..., `v1`, `v2` ⇒ ..., `result`
`dsub` ..., `v1-word1`, `v1-word2`, `v2-word1`, `v2-word2` ⇒ ..., `result`
`imul` ..., `v1`, `v2` ⇒ ..., `result`

```
lmul    ..., v1-word1, v1-word2, v2-word1, v2-word2 ⇒ ..., result
fmul    ..., v1, v2 ⇒ ..., result
dmul    ..., v1-word1, v1-word2, v2-word1, v2-word2 ⇒ ..., result
idiv    ..., v1, v2 ⇒ ..., result
ldiv    ..., v1-word1, v1-word2, v2-word1, v2-word2 ⇒ ..., result
fdiv    ..., v1, v2 ⇒ ..., result
ddiv    ..., v1-word1, v1-word2, v2-word1, v2-word2 ⇒ ..., result
```

Transfer of Control

For transfer of control we have the following bytecodes:

```
ifeq    ..., value ⇒ ...
ifne    ..., value ⇒ ...
iflt    ..., value ⇒ ...
ifgt    ..., value ⇒ ...
ifle    ..., value ⇒ ...
ifge    ..., value ⇒ ...

ifcmpeq ..., value1,value2 ⇒ ...
ifcmpne ..., value1,value2 ⇒ ...
ifcmplt ..., value1,value2 ⇒ ...
ifcmpgt ..., value1,value2 ⇒ ...
ifcmple ..., value1,value2 ⇒ ...
ifcmpge ..., value1,value2 ⇒ ...
```

When `value <rel> 0` is true in the first set of bytecodes, `value1 <rel> value2` is true in the second set.

6.3 The Code Generator

We have seen earlier that the intermediate representation leading to the code generation is included in the grammar description file. We have also explained that each rule of production has two associated parts, namely the semantic part and the part containing the priority of that rule. It is the semantic part that was left out to this phase of analysis. This part is used to describe the bytecodes and their operands associated to each rule containing a semantic value. The semantic part, if it exists, is included exactly after the rule of production and it is separated by the `$$` sign. It is read in the `FillInitState` function, and stored in the variable pointer `InitRules.RFirst.semantic_rule`. This latter has exactly the same structure `WORDNODE` used to store the right-hand side of a rule of production, but with more variables used in this case within the structure. The semantic part of a rule consists of a number of intervals each of which containing four items. The first item in each interval may be used to specify that there is a label(a

line number in our case) before a bytecode instruction. This label might be referred to for later use in the bytecodes generation. For example, when there is a for loop, we have to indicate the place where it has started in order to go back to the beginning of this loop. So the item here must be a label `L1FOR`. When there is another interval having `EL1FOR` as its first item, we therefore know that either we have to go back to the beginning of that for loop, or it was the end of the loop. If the first item does not exist in a certain interval, it is replaced by a dash. This item when read is saved in the variable `label` of the structure `WORDNODE` pointed to by `semantic_rule`. The second item in the interval specifies the symbol in the rule of production to be treated semantically. The third item, if existing, must be a letter `v` or `s`. When it is a `v`, it means that either the nonterminal of a rule must be assigned its semantic value taken from its descendant rule or else, it might be an interval that has a bytecode instruction and it means that an operand must be associated to that instruction. If the letter was an `s` in the interval, it means that this interval contains a bytecode instruction whose operand is the same as the one for the instruction included in the preceding interval. The fourth item contains the bytecode intruction to be generated. This instruction is saved in the variable `instruction` of the structure `WORDNODE`.

When the parse tree is constructed, it must contain all the productions of the source program compiled, as well as the semantic rules, if existing, attached to each one of these productions. The code generation procedure is performed by traversing the parse tree. It is the function `traverse_tree` that does the job. This function starts visiting the rules from the root of the tree, i.e., from the starting production. It traverses each rule from left to right. Each time there is a non-terminal in the rule, it is called recursively to traverse all its descendants rules. When the recursive calls due to the non-terminals are executed, the function turns back to traverse the next symbols in the activating rule. The code is generated in the following manner: starting with a production, each of its non-terminals is compared in turn with the interval on the head of its semantic rule. If the second item of the semantic rule is the same as the non-terminal, and all the other items are empty, it means that this non-terminal has to be resolved later by a recursive call. By resolving a non-terminal, we mean that its semantic value `semantic_value` declared in the structure `RULE` is taken from its descendant rules. This value is nothing that that of a lexeme obtained when parsing a terminal in the syntax analysis and saved in the variable `value` of the structure `WORDNODE`. Now if the third item in the interval was empty, we know that this non-terminal is to be resolved in its descendant rule. If the letter `v` appeared in the third item, it means that the rule containing that non-terminal is to take the semantic value of its descendant rule. This value may be an operand to be used with a bytecode instruction. When an instruction appears in the fourth item of an interval, it means that there is a new line of code to be generated. Note that for typed instructions like `iadd`, `fadd` ,..., we used `xadd` in the interval instead. The reason is that we don't know the type of an instruction unless in the code generation process. This instruction is resolved at the time of generation when the type of its operand is already known. We can find some intervals having a fourth item but not having a `v` in the third item because some instructions don't have operands associated to them. Instead, we may find a label, for example, for a `goto` interval we can find `EL1FOR` as a third item. This means the operand here is nothing

but a code line number to refer to the beginning of a nesting loop. After an interval in the semantic rule is being processed, we advance the pointer so that the head of this semantic rule points to the next interval. This next interval will be compared to the next symbol of the activating rule.

6.4 Executing our Compiler

For the execution of the program, we are supposed to have each of the three main files in the same directory containing the executable file:

- The tokens description file.
- The language grammar decription file.
- The source program to be compiled.

The result of the execution, i.e., the bytecode generated by the compiler will be held in a file called `result`. The complete source code of the compiler is found in the Appendix. To illustrate our work, we show an example of a program written in the BASIC language with the bytecode instructions generated after the compilation process. Note that the expression `T = 34` gives the two bytecode instructions `bipush 34` and `istore_0`. This means that the value 34 is pushed into the stack at the entry whose index is `0` . This index is saved into the symbol table at the entry holding the variable `T` in order to know to what index `T` corresponds in the stack, whenever met later in the program analysis.

```
2  T = 34
3  T1 = 20.0 / 4
4  T = T1
5  T3 = T + T1 *2 - 1
6  T = T3 + 700
7  DIM A (123)
10 FOR I = 1 TO 5
20 FOR J = 1 TO 5
30 Y = 1
40 NEXT J
50 NEXT I
60 END
```

The bytecode generated for the above program looks like the following.

```
0 bipush 34
1 istore_0
2 lc2w 20.0
3 bipush 4
4 fdiv
```

```
5 fstore_1
6 fload_1
7 fstore_0
8 fload_0
9 fload_1
10 bipush 2
11 fmul
12 fadd
13 bipush 1
14 fsub
15 fstore_2
16 fload_2
17 bipush 700
18 fadd
19 fstore_0
20 bipush 1
21 istore_3
22 iload_3
23 bipush 5
24 if_icmpge 38
25 bipush 1
26 istore_4
27 iload_4
28 bipush 5
29 if_icmpge 35
30 bipush 1
31 istore_5
32 iinc 4
33 iload_4
34 goto 28
35 iinc 3
36 iload_3
37 goto 23
```

Chapter 7

Conclusion

We have presented a general compiler for most context-free grammars that is appealing for its combination of advantages over existing compilers. Earley's control structure lets the compiler run with best-known complexity on a number of grammar subclasses, and no worse than other bottom-up parsers based on CFG and CNF grammars. Moreover, it does this with one single algorithm which does not have specified to it the class of grammars it is operating on and does not require the grammar in any special form.

Unlike other compilers it also generates an extremely portable code based on the Java virtual machine instructions. It translates the source code into bytecodes, a lower-level format that cannot be run directly, but must be further interpreted by each computer. Of course, it is exactly this level of *indirection* that buys us the power, flexibility, and portability of the java code.

The algorithm just discussed in chapter 2 was the basic motivation for developing the compiler presented here. We identified one central cause of previous parsing methods of implementation as a major obstacle for compiler implementors: little attention was given to the type of grammars involved. Instead, most parsing methods used to assume a specific type of a grammar. In addition, we can add a second problematic case, where most compilers used to generate code for a specific machine architecture while we needed some kind of portable code that can be used across the NET in our days. To overcome these limitations, we have proposed here an implementation of a new compiler that puts together many solutions to previously implemented compilers. To accomplish this work, many studies were realized on the above issues; we divide them into the following parts:

- A study of Earley's algorithm that can be used as a scanner and a parser at the same time in the implementation.
- A new method to treat the priority between rules of production, e.g. in the case of arithmetic expressions, priority between operators must be considered.
- A method for constructing the parse tree from the set of states generated by Earley's transitions(scanning step, completion, prediction).
- A study of the java virtual machine instructions used in the code generation process.

On the basis of these issues, different methods of implementation were used in the development of the compiler. We have discussed the most important data structures involved. Of course, these data structures can be modeled in a more efficient way but due to the limitation of the thesis, we chose the simplest data structures to implement our compiler.

Finally, with all the efficiency of Earley's algorithm implemented as a scanner and a parser, there are still many disadvantages making it difficult to have our compiler completely realized. For example, type checking is not an easy job to perform with Earley's algorithm. We have already mentioned that most compilers are constructed according to their specific language grammars. Thus, it is easy to implementors to write their analyzers accordingly to what they see in those grammars. In contrast, in our general compiler the type of grammar is not already known and must be provided by the user. Consequently, further methods may be added to the compiler to help in type error detection. More type specification is needed in the grammar description file to realize this work. Moreover, error detection cannot be 100% accurate. When tokens in a line of code are erroneously found in excess to those of a rule of production, it is impossible to detect an error thus obliged to stop the compilation process. Another problem is the detection of numbers exceeding the range of integers or floats. To solve this problem, more rules must be added to the tokens description file enabling us to verify the correctness of a numeric identifier. We still have the problem of performance. Our compiler will not compete so favorably with other compilers. Unfortunately, our implementation is not so accurate that it might take time to compile a large number of code lines. More analysis can be done in order to obtain a fully implemented compiler.

Appendix

The tokens definition file of the BASIC language is described as follows:

```
%%token  FN SIN COS TAN ATN EXP ABS LOG SQR INT RND
%%token  DIM LET DATA RESTORE GO TO ON GOSUB RETURN
%%token  THEN FOR NEXT STEP IF REM STOP END READ PRINT
%%token  DEF FNEND
%%token  + - * / =
%%token  | < > . , ; $  ? ( )
%%terminal fraction decimal lettre message nombre_signe  rem
%%terminal variable_simple entier nombre_exponentiel

%%type     fraction            :     D
%%type     decimal             :     F
%%type     entier              :     I
%%type     nombre_exponentiel  :     F
%%assign_op =
%%

ss start $$
start fraction $$
start decimal $$
start lettre $$
start entier $$
start message $$
start nombre_signe $$
start nombre_exponentiel $$
start rem $$
start variable_simple $$
start char_special $$

message     " char0_n " $$

entier chiffre1_n $$

nombre_signe   signe0_1 constante $$

nombre     entier $$
nombre     fraction $$
nombre     decimal $$

nombre_exponentiel nombre  exponentiel $$

chiffre1_n chiffre chiffre1_n  $$
chiffre1_n chiffre $$

chiffre0_n chiffre1_n $$
chiffre0_n $$

fraction   . entier $$

decimal    chiffre1_n . chiffre0_n $$

rem    R E M char1_n $$
```

```
char1_n  char char1_n $$
char1_n  char $$

char0_n   char1_n $$
char0_n   $$

char      lettre $$
char      chiffre $$
char      char_special $$

exponentiel E signe0_1 chiffre1_n $$

signe0_1 signe $$
signe0_1 $$

signe + $$
signe - $$

variable_simple   lettre chiffre0_1 $$

chiffre0_1   chiffre $$
chiffre0_1   $$

blanc $$

chiffre 0 $$
chiffre 1 $$
chiffre 2 $$
chiffre 3 $$
chiffre 4 $$
chiffre 5 $$
chiffre 6 $$
chiffre 7 $$
chiffre 8 $$
chiffre 9 $$

char_special   + $$
char_special   - $$
char_special   * $$
char_special   / $$
char_special   = $$
char_special   ) $$
char_special   ( $$
char_special   > $$
char_special   < $$
char_special   . $$
char_special   , $$
char_special   ; $$
char_special   $ $$
char_special   | $$
char_special   ? $$

lettre A lettre1 $$
lettre B lettre1 $$
lettre C lettre1 $$
lettre D lettre1 $$
lettre E lettre1 $$
lettre F lettre1 $$
lettre G lettre1 $$
lettre H lettre1 $$
lettre I lettre1 $$
lettre J lettre1 $$
lettre K lettre1 $$
lettre L lettre1 $$
lettre M lettre1 $$
lettre N lettre1 $$
lettre O lettre1 $$
lettre P lettre1 $$
```

```
lettre Q lettre1 $$
lettre R lettre1 $$
lettre S lettre1 $$
lettre T lettre1 $$
lettre U lettre1 $$
lettre V lettre1 $$
lettre W lettre1 $$
lettre X lettre1 $$
lettre Y lettre1 $$
lettre Z lettre1 $$

lettre1 A lettre1 $$
lettre1 B lettre1 $$
lettre1 C lettre1 $$
lettre1 D lettre1 $$
lettre1 E lettre1 $$
lettre1 F lettre1 $$
lettre1 G lettre1 $$
lettre1 H lettre1 $$
lettre1 I lettre1 $$
lettre1 J lettre1 $$
lettre1 K lettre1 $$
lettre1 L lettre1 $$
lettre1 M lettre1 $$
lettre1 N lettre1 $$
lettre1 O lettre1 $$
lettre1 P lettre1 $$
lettre1 Q lettre1 $$
lettre1 R lettre1 $$
lettre1 S lettre1 $$
lettre1 T lettre1 $$
lettre1 U lettre1 $$
lettre1 V lettre1 $$
lettre1 W lettre1 $$
lettre1 X lettre1 $$
lettre1 Y lettre1 $$
lettre1 Z lettre1 $$
lettre1 $$
```

The grammar descritpion file of the BASIC language looks like the following:

```
%%variable    variable              : variable_simple
%%variable    -                     : variable_simple
%%function    def                   : lettre
%%function1   entete                : lettre
%%array       dim_tableau           : lettre
%%array_len   dim_tableau           : entier
%%array_len   c_entier0_1           : entier
%%fun_par     def                   : variable_simple
%%fun_par     c_variable_simple0_n  : variable_simple
%%fun_par1    entete                : variable_simple
%%fun_par1    c_variable_simple0_n  : variable_simple
%%

programme       enonce0_n terminal $$ $$ 0

enonce0_n       enonce enonce0_n $$ $$ 0

enonce0_n       $$ $$ 0

enonce          numero_ligne donnee1 $$ $$ 0

donnee1         donnee $$ $$ 0

donnee1         $$ $$ 0

donnee          let $$ $$ 0
donnee          read $$ $$ 0
donnee          data $$ $$ 0
donnee          print $$ $$ 0
donnee          goto $$ $$ 0
donnee          on $$ $$ 0
donnee          if $$ $$ 0
donnee          for $$ $$ 0
donnee          next $$ $$ 0
donnee          dim $$ $$ 0
donnee          gosub $$ $$ 0
donnee          return $$ $$ 0
donnee          restore $$ $$ 0
donnee          stop $$ $$ 0
donnee          rem $$ $$ 0
donnee          def $$ $$ 0

terminal        numero_ligne end $$ $$ 0

numero_ligne    entier blanc $$ $$ 0

expression      facteur_multiplicatif $$ $$ 0
expression      signe expression $$ $$ 0
expression      expression - facteur_multiplicatif $$
[ - expression - - ] [ - facteur_multiplicatif - - ]
[ - - v xsub ] $$ 0

expression      expression + facteur_multiplicatif $$
[ - expression - - ] [ - facteur_multiplicatif - - ]
[ - - v xadd ] $$ 0

facteur_multiplicatif    facteur_multiplicatif * facteur_involutif $$
[ - facteur_multiplicatif - - ] [ - facteur_involutif  - - ]
[ - - v xmul ] $$ 0
facteur_multiplicatif    facteur_multiplicatif / facteur_involutif $$
[ - facteur_multiplicatif - - ] [ - facteur_involutif  - - ]
[ - - v xdiv ] $$ 0
facteur_multiplicatif    facteur_involutif $$
[ - facteur_involutif v - ] $$ 0

facteur_involutif    terme $$  $$ 0
```

```
facteur_involutif    terme | terme $$ $$ 0

terme     constante    $$  $$ 0
terme     variable     $$ [ - variable v xload ] $$ 0
terme     reference_fonction $$ $$ 0
terme     ( expression ) $$ $$ 0

variable   variable_simple $$  [ - variable_simple v - ] $$ 0
variable   variable_indicee $$ $$ 0

constante nombre $$ $$ 0
constante nombre_exponentiel $$ $$ 0

nombre     entier $$ [ - entier v bipush ] $$ 0
nombre     fraction $$ [ - fraction v lc2w ] $$ 0
nombre     decimal $$ [ - decimal v lc2w ] $$ 0

variable_indicee   lettre ( expression c_expression0_1 ) $$ $$ 0

c_expression0_1     , expression $$ $$ 0
c_expression0_1     $$ $$ 0

signe + $$ $$ 0
signe - $$ $$ 0

reference_fonction    nom_de_fonction ( expression c_expression0_n ) $$ $$ 0

c_expression0_n     , expression expression0_n $$ $$ 0
c_expression0_n     $$ $$ 0

nom_de_fonction   fonction_librairie $$ $$ 0
nom_de_fonction   fonction_d'utilisateur $$ $$ 0

fonction_d'utilisateur   FN lettre $$ $$ 0

fonction_librairie  SIN $$ $$ 0
fonction_librairie  COS $$ $$ 0
fonction_librairie  TAN $$ $$ 0
fonction_librairie  ATN $$ $$ 0
fonction_librairie  EXP $$ $$ 0
fonction_librairie  ABS $$ $$ 0
fonction_librairie  LOG $$ $$ 0
fonction_librairie  SQR $$ $$ 0
fonction_librairie  INT $$ $$ 0
fonction_librairie  RND $$ $$ 0

dim   DIM dim_tableau c_dim_tableau0_n $$ $$ 0

c_dim_tableau0_n     , dim_tableau c_dim_tableau0_n $$ $$ 0
c_dim_tableau0_n     $$ $$ 0

dim_tableau     lettre ( entier c_entier0_1 ) $$ $$ 0

c_entier0_1     , entier $$ $$ 0
c_entier0_1     $$ $$ 0

let    let0_1 variable = expression $$
[ - expression - - ] [ - variable v xstore ] $$ 0

let0_1    LET $$ $$ 0
let0_1    $$ $$ 0

liste_data    nombre_signe c_nombre_signe0_n $$ $$ 0

c_nombre_signe0_n    , nombre_signe c_nombre_signe0_n $$ $$ 0
c_nombre_signe0_n    $$ $$ 0

data DATA liste_data0_1 $$ $$ 0
```

```
liste_data0_1  liste_data $$ $$ 0
liste_data0_1  $$ $$ 0

restore    RESTORE $$ $$ 0

goto  GO TO numero_ligne $$ $$ 0

on    ON expression GO TO numero_ligne c_numero_ligne1_n $$ $$ 0

c_numero_ligne1_n  , numero_ligne c_numero_ligne1_n $$ $$ 0
c_numero_ligne1_n  , numero_ligne $$ $$ 0

gosub    GOSUB numero_ligne $$ $$ 0

return   RETURN $$ $$ 0

if       IF expression_booleenne THEN numero_ligne $$ $$ 0

expression_booleenne    expression =  expression $$ $$ 0
expression_booleenne    expression >= expression $$ $$ 0
expression_booleenne    expression <= expression $$ $$ 0
expression_booleenne    expression > expression $$ $$ 0
expression_booleenne    expression < expression $$ $$ 0
expression_booleenne    expression <> expression $$ $$ 0

for FOR variable_simple = expression TO expression c_step_exp0_1 $$
[ - expression - - ] [ - variable_simple v xstore ]
[ - - s xload ] [ L1FOR - - - ] [ - expression - - ]
[ - - L2FOR if_icmpge ] $$ 0

c_step_exp0_1  STEP expression  $$ $$ 0
c_step_exp0_1  $$ $$ 0

next  NEXT variable_simple $$
[ - variable_simple v iinc ] [ - - s xload ]
[ - - EL1FOR goto ] [ EL2FOR - - - ] $$ 0

rem   REM char1_n $$ $$ 0

char1_n   char char1_n $$ $$ 0
char1_n   char $$ $$ 0

stop      STOP $$ $$ 0
end       END $$ $$ 0

read READ variable c_variable0_n $$ $$ 0

c_variable0_n    , variable c_variable0_n $$ $$ 0
c_variable0_n    $$ $$ 0

print       PRINT chaine_print ponct0_1 $$ $$ 0

ponct0_1    ponct $$ $$ 0
ponct0_1    $$ $$ 0

ponct       , $$ $$ 0
ponct       ; $$ $$ 0

chaine_print     message $$ $$ 0
chaine_print     expression $$ $$ 0
chaine_print     chaine_print1_n $$ $$ 0

chaine_print1_n chaine_print $$ $$ 0
chaine_print1_n chaine_print chaine_print1_n $$ $$ 0

def  DEF blanc FN lettre ( variable_simple c_variable_simple0_n )
= expression $$ $$ 0
```

```
def def_bloc $$ $$ 0

c_variable_simple0_n   , variable_simple c_variable_simple0_n $$ $$ 0
c_variable_simple0_n   $$ $$ 0

entete  DEF blanc FW lettre ( variable_simple c_variable_simple0_n ) $$ $$ 0

fin_def   FWEND $$ $$ 0

def_bloc   numero_ligne entete enonce0_n numero_ligne fin_def $$ $$ 0

blanc   $$ $$ 0

lettre A $$ $$ 0
lettre B $$ $$ 0
lettre C $$ $$ 0
lettre D $$ $$ 0
lettre E $$ $$ 0
lettre F $$ $$ 0
lettre G $$ $$ 0
lettre H $$ $$ 0
lettre I $$ $$ 0
lettre J $$ $$ 0
lettre K $$ $$ 0
lettre L $$ $$ 0
lettre M $$ $$ 0
lettre N $$ $$ 0
lettre O $$ $$ 0
lettre P $$ $$ 0
lettre Q $$ $$ 0
lettre R $$ $$ 0
lettre S $$ $$ 0
lettre T $$ $$ 0
lettre U $$ $$ 0
lettre V $$ $$ 0
lettre W $$ $$ 0
lettre X $$ $$ 0
lettre Y $$ $$ 0
lettre Z $$ $$ 0

char_special   + $$ $$ 0
char_special   - $$ $$ 0
char_special   * $$ $$ 0
char_special   / $$ $$ 0
char_special   = $$ $$ 0
char_special   ) $$ $$ 0
char_special   ( $$ $$ 0
char_special   > $$ $$ 0
char_special   < $$ $$ 0
char_special   . $$ $$ 0
char_special   , $$ $$ 0
char_special   ; $$ $$ 0
char_special   blanc $$ $$ 0
char_special   $ $$ $$ 0
char_special   | $$ $$ 0
char_special   ? $$ $$ 0
```

This is the complete source program of the compiler written in C++ language [11].

```
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>

#define MAX_STATE 2000            // max number of states
#define LINE_LENGTH 255
#define nb_reserved_words 500     // number of reserved words
#define max_symbols 500           //max number of coded symbols
#define max_identifiers 500       // used for table of identifiers
#define max_labels 200

#define MAIN 1  // used to define current_block
#define FUNC 2  // used to define current_block

#define FLOAT 1
#define INTEGER 2
#define DOUBLE 3
#define UNKNOWN_TYPE 4

#define ARRAY 0
#define ARRAY1 1
#define VARIABLE 2
#define VARIABLE1 3
#define CONSTANT 4
#define CONSTANT1 5
#define FUNCTION 6
#define FUNCTION1 7
#define FUN_PAR 8
#define FUN_PAR1 9
#define ARRAY_LEN 10
#define ARRAY_LEN1 11

// GLOBAL TYPE DEFINITIONS

struct WORDNODE
/* structure containing each word in the rule of production*/
{
  int word;
  char *value;
  char *value1;
  char *instruction;
  char *label;
  struct WORDNODE  *next;
  struct WORDNODE  *before;
  struct RULE *desc;
};

typedef struct RULE /*structure for each rule of production */
{
  int pred_ptr, priority;
  int left_hs;
  char *semantic_value;
  struct WORDNODE *right_hs;
  struct WORDNODE *right_hs_tail;
  struct WORDNODE  *after_ptr;
  struct WORDNODE *semantic_rule;
  struct RULE *ref_ptr;
  struct RULE *ref_ptr1;
  struct RULE *next;
} *RULEPTR;

typedef struct states
{/* each state contains a pointer to the 1st and last
```

```
rule of production */
  RULEPTR RFirst,RLast;
} STATE;

/* The following structure will be used in the function tree() in
   order to save the values of ref_ptr1 used in constructing the syntax
   tree */

typedef struct RefPtr1
{
  RULEPTR tempList;
  struct RefPtr1 *next;
} * REFPTR1P;

typedef struct Id_specification
{ // description of each ident. in the grammar
  int specification,ident;
  int specification1,ident1;
} ID_SPECIFICATION ;

typedef struct t_ident_record
{
  char *name;
  int type;
  union id_info
  {
    struct variable
    {
      int v_typ;
      int v_adr;
    } variable;
    struct fun_par
    {
      int p_typ;
      int p_adr;
      int next;
    } fun_paramter;
    struct function
    {
      int param;
      int f_adr;
      char *tag;
    } function;
    struct constant
    {
      float cons;
    } constant;
    struct array
    {
      int el_typ;
      int length;
    } array;
  } id_info;
} *t_ident_record_ptr;

typedef struct type_specification
{
  int symbol;
  int type;
} *type_specification_ptr;

typedef struct label
{
  fpos_t *pos;
  int value;
} *label_ptr;

typedef struct loop_label
```

```
{
  char *name;
  int value,adr;
} *loop_label_ptr;

//  GLOBAL VARIABLES

int last_reserved_word,line,last_terminal,assign_op;
char *t_reserved_words[nb_reserved_words],*eof;
t_ident_record_ptr t_symbol_table[max_identifiers];
type_specification_ptr t_id_types[10];
FILE *error_f;

/* The following class is the lexer class. It contains all the
   functions needed to build the lexical analayser */

class lexer {

// VARIABLES

STATE state[MAX_STATE], InitRules;
char ch[LINE_LENGTH];
char *t_lex_symbols[max_symbols];
int i,symbol_last_index,token;


// private functions

void add_rule(STATE *state,RULEPTR new_Rule); // to add rule to a state
int found(STATE state,RULEPTR new_rule); // if rules already exists
void skip_seperators();

void initialize_t_reserved_words();
void sort_t_reserved_words();
int add_lex_symbol(char *symbol);
void  initialize_t_id_types();

// public functions

public:

FILE *program;
int FillInitState();
int lex_analyser(int tokens[],char value[]);
void affiche();
int is_reserved_word(char *name);
void read_newline();   // to read a newline from the source code
};

/*  The  Function FillInitState() reads all  the rules  of
productions describing the  tokens of the   language grammar
from the  file and store them  in  the variable InitRules for
later use  in Earley's analyser. It also reads the specification
of reserved words. */

int lexer::FillInitState()
{
  char temp[50];
  struct WORDNODE *tempWords, *tempLastWords;
  struct RULE *tempRule;
  int i = 0;
  ifstream in("basic_sp");  // open file grammar

  if(!in) {
    cout << "Cannot open the grammar file.\n";
    return 1;
    }
```

```
InitRules.RFirst=NULL;
InitRules.RLast=NULL;

initialize_t_reserved_words();
initialize_t_id_types();//initialize table of types specification

last_reserved_word = 0;
in >> temp;

while (strcmp(temp,"%%")) // fill indexed table of reserved words
  {
    if (!strcmp(temp,"%%terminal"))
      {
        in >> temp;
        do {
          t_reserved_words[last_terminal] =
                      (char *)malloc(strlen(temp)+1);
          strcpy(t_reserved_words[last_terminal++],temp);
          in >> temp;
        } while  (strcmp(temp,"%%terminal") && strcmp(temp,"%%") &&
                  strcmp(temp,"%%token") && strcmp(temp,"%%type") &&
                  strcmp(temp,"%%assign_op") );
      }
    if (!strcmp(temp,"%%token"))
      {
        in >> temp;
        do {
          t_reserved_words[last_reserved_word]=
                             (char *)malloc(strlen(temp)+1);
          strcpy(t_reserved_words[last_reserved_word++],temp);
          in >> temp;
        } while  (strcmp(temp,"%%terminal") && strcmp(temp,"%%") &&
                  strcmp(temp,"%%token") && strcmp(temp,"%%type") &&
                  strcmp(temp,"%%assign_op"));
        last_terminal = last_reserved_word;
      }
    if (!strcmp(temp,"%%type"))
      {
        in >> temp;
        int l;
        do {
          t_id_types[i] = new struct type_specification;
          for(l=last_reserved_word;l < last_terminal;l++)
            if (!strcmp(temp,t_reserved_words[l]))
              t_id_types[i]->symbol = l;
          in >> temp;
          in >> temp;
          if (!strcmp(temp,"I"))
            t_id_types[i]->type = INTEGER;
          else   if (!strcmp(temp,"D"))
            t_id_types[i]->type = DOUBLE;
          else   if (!strcmp(temp,"F"))
            t_id_types[i]->type = FLOAT;
          in >> temp;
          ++i;
       }  while  (strcmp(temp,"%%terminal") && strcmp(temp,"%%") &&
                  strcmp(temp,"%%token") && strcmp(temp,"%%type")&&
                  strcmp(temp,"%%assign_op"));
      }
    if (!strcmp(temp,"%%assign_op"))
      {
        sort_t_reserved_words();
        in >> temp;
        if (is_reserved_word(temp))
          assign_op = token;
        in >> temp;
      }
  }
```

```
  for(i=0;i<max_symbols;i++)
    t_lex_symbols[i] = NULL;

  while (in >> temp)
    {
      tempRule= new struct RULE;
      tempRule->pred_ptr = 0;
      if (strlen(temp) == 1) {      // if temp is one character
        tempRule->left_hs = temp[0];
      }
      else
        {
          add_lex_symbol(temp);
          tempRule->left_hs = symbol_last_index;
        }
      tempRule->right_hs=NULL;
      tempRule->next=NULL;
      tempRule->after_ptr=NULL;
      in >> temp;

      while(strcmp(temp,"$$"))
        {
          tempWords= new struct WORDNODE;
          if (strlen(temp) == 1)
            tempWords->word = temp[0];
          else
            {
              add_lex_symbol(temp);
              tempWords->word =symbol_last_index;
            }
          tempWords->next=NULL;
          if(tempRule->right_hs==NULL)
                  {
                    tempRule->right_hs=tempWords;
                    tempLastWords=tempWords;
                    tempRule->after_ptr=tempWords;
                  }
                else
                  {
                    tempLastWords->next=tempWords;
                    tempLastWords  =tempWords;
                  }
          in >> temp;
        }

      if(InitRules.RFirst==NULL)
        {
          InitRules.RFirst=tempRule;
          InitRules.RLast=tempRule;

        }
      else
        {
          InitRules.RLast->next=tempRule;
          InitRules.RLast=tempRule;

        }
    }

  return 0;
}

/* The  function 'lex_analyser' is called each  time by  the parser to
   read the next token  to be parsed. It  returns more than one  token
   for the same identifier if any */
```

```
int lexer::lex_analyser(int tokens[],char symbol[])
{

  RULEPTR tempRule,tempRule1,tempRule2;
//  char symbol[30];
  int end_of_symbol,j,l;
  int count = 0;


  for (j=0;j<MAX_STATE;j++)
    {
      state[j].RFirst = state[j].RLast = NULL;
    }

  state[0].RFirst = new struct RULE;
  state[0].RFirst->pred_ptr = 0;
  state[0].RFirst->left_hs = InitRules.RFirst->left_hs;
  state[0].RFirst->right_hs = InitRules.RFirst->right_hs;
  state[0].RFirst->after_ptr = InitRules.RFirst->after_ptr;
  state[0].RFirst->next = NULL;
  state[0].RLast = state[0].RFirst;

  j = 0;

  skip_seperators();

  if (eof != NULL)
    end_of_symbol = 0;

  while (!end_of_symbol)
    {
      end_of_symbol = 1;
      tempRule = state[count].RFirst;
      symbol[j++] = ch[i++];

      while (tempRule != NULL) {
        if (tempRule->after_ptr != NULL)
          {
            if (tempRule->after_ptr->word == ch[i-1])
              {
                /*if state i in A->a .Xb,j and next input
                symbol is X*/
                end_of_symbol = 0;
                tempRule1 = new struct RULE;
                tempRule1->left_hs = tempRule->left_hs;
                tempRule1->right_hs = tempRule->right_hs;
                tempRule1->pred_ptr = tempRule->pred_ptr;
                tempRule1->after_ptr=tempRule->after_ptr->next;
                tempRule1->next= NULL;
                if (! found(state[count+1],tempRule1))
                                  /*rule not already exists?*/
                  {
                    add_rule(&state[count+1],tempRule1);
                  }
              }
            else
                /*if state i in A-> a.Bb,j then predict B-> .y,i*/
              {
                tempRule2 = InitRules.RFirst;
                while (tempRule2 != NULL)
                  {
                    if (tempRule->after_ptr->word ==
                                                tempRule2->left_hs)
                      {
                        tempRule1 = new struct RULE;
                        tempRule1->left_hs = tempRule2->left_hs;
                        tempRule1->pred_ptr = count;
                        tempRule1->right_hs = tempRule2->right_hs;
```

```
                    tempRule1->after_ptr = tempRule2->after_ptr;
                    tempRule1->next= NULL;
                    if (! found(state[count],tempRule1))
                      {
                        add_rule(&state[count],tempRule1);
                      }
                  }
                tempRule2 = tempRule2->next;
              }
          }
      }
    else
    /* if state i contains A->y.,j then go to state j */
      {
        tempRule2 = state[tempRule->pred_ptr].RFirst;
        while (tempRule2 != NULL)
          {
            if (tempRule2->after_ptr!=NULL)
              {
                if (tempRule2->after_ptr->word == tempRule->left_hs)
                  {
                    tempRule1 = new struct RULE;
                    tempRule1->left_hs=tempRule2->left_hs;
                    tempRule1->pred_ptr=tempRule2->pred_ptr;
                    tempRule1->right_hs=tempRule2->right_hs;
                    tempRule1->after_ptr =
                                    tempRule2->after_ptr->next;
                    tempRule1->next= NULL;
                    if (!found(state[count],tempRule1))
                      {
                        add_rule(&state[count],tempRule1);
                      }
                  }
              }
            tempRule2 = tempRule2->next;
          }
      }

    tempRule = tempRule->next;
  }
  count ++;
  if (end_of_symbol)
    {
      tokens[0]= tokens[1] = 9999;
      tempRule = new struct RULE;
      tempRule->pred_ptr = 0;
      tempRule->left_hs = InitRules.RFirst->left_hs;
      tempRule->right_hs = InitRules.RFirst->right_hs;
      tempRule->after_ptr = NULL;
      if ((found(state[count-1],tempRule)) && ((count-1) != 0))
       /* if the starting rule can be found at the last state->sym found */
        {
          symbol[--j] = '\0';
          if (!is_reserved_word(symbol))
            {
              token = 9999;
              tempRule = state[count-1].RFirst;
              while (tempRule != NULL)
                {
                  if (tempRule->left_hs ==
                            InitRules.RFirst->after_ptr->word)
                    {
                      if (tempRule->pred_ptr == 0)
                        if (tempRule->after_ptr == NULL)
                          {
                            for(l=last_reserved_word;l
                                    < last_terminal;l++)
                              if (!strcmp(
```

```
                                    t_lex_symbols[tempRule->right_hs->word],
                                    t_reserved_words[1]))
                                   {
                                     if (token == 9999)
                                       token = 1;
                                     else
                                       if (tokens[0] == 9999)
                                         tokens[0] = 1;
                                       else
                                         tokens[1] = 1;
                                   }

                                }
                              }
                            tempRule = tempRule->next;
                          }
                      }
                    i--;
                  }
                  else
                    {
                      printf("\n error\n");
                    }
            }
        }

return token;
}

void  lexer::initialize_t_id_types()
{
  int i;
  for (i=0;i<12;i++)
    t_id_types[i] = NULL;
}

/* The function 'add_lex_symbol' saves  every new symbol read
from the file containing     the  rules   of    productions  in
the   table "t_lex_symbols". This is    used    to  give  symbols
a numerical representation  thus facilitate  the  comparisons when
using Earley's analyser*/

int lexer::add_lex_symbol(char *symbol)
{
  int found;
  found = 0;
  for(symbol_last_index=256;
        symbol_last_index<max_symbols;symbol_last_index++)
    {
      if (t_lex_symbols[symbol_last_index] == NULL)
        break;

      if (!strcmp(t_lex_symbols[symbol_last_index],symbol))
        {
          found = 1;
          break;
        }
    }
  if (!found)
    {
      t_lex_symbols[symbol_last_index] =
                        (char*)malloc(strlen(symbol)+1);
      strcpy(t_lex_symbols[symbol_last_index],symbol);
    }
  return 1;
}
```

```
/* The function "initialize_t_reserved_words" initializes the
table of reserved words */

void lexer::initialize_t_reserved_words()
{
  int i;
  for (i=0;i<nb_reserved_words;i++)
    t_reserved_words[i] = NULL;
}

/* The function "sort_t_reserved_words" sorts the symbols
contained in the table of reserved words. */

void lexer::sort_t_reserved_words()
{
  int i,j;
  char *temp;

  for(i=0;i < last_reserved_word;i++)
    for(j=i; j < last_reserved_word;j++)
      {
        if (strcmp(t_reserved_words[j],t_reserved_words[i]) < 0)
          {
            temp = t_reserved_words[i];
            t_reserved_words[i] = t_reserved_words[j];
            t_reserved_words[j] = temp;
          }
      }
}


/*  The function "is_reserved_word" checks  if the  current token
is a reserved word for  the language grammar or  not. it reurns 1
if the search is successful, 0 otherwise.*/

int lexer::is_reserved_word(char *name)
{
  int i,j,k;

  j = last_reserved_word - 1;
  i = 0;


  do {
    k = (i+j)/2;
    if (strcmp(name,t_reserved_words[k]) > 0)
      i = k+1;
    else
      j = k-1;
  } while ((strcmp(name,t_reserved_words[k])) && (i <= j));
  token = k;
  return (!strcmp(name,t_reserved_words[k]));
}

/*The function "read_newline"   reads a new  line  of  input
from the source program.*/

void lexer::read_newline()
{
  eof = fgets(ch,LINE_LENGTH,program);
  if (eof != NULL)
    {
      i = 0;
      line++;
    }
}

/* The function  "skip_seperators"   skips all the seperators
```

```
in the source program(tabs,newline,spacebars,...)*/

void lexer::skip_seperators()
{
  while (((ch[i] == ' ') || (ch[i] == 9) ||
        (ch[i] == 0) || (ch[i] == '\n'))
         && (eof != NULL))
    {
      if ((ch[i] == ' ') || (ch[i] == 9))  // blank or tab
        {
          ++i;
        }
      else
        {
          read_newline();
        }

    }
}

/* The  function "add_rule"  adds a   new rule to   the set
of states pointed toi by "state". This new rule is  generated
when there is a transition(prediction,scanning step,completion.*/


void lexer::add_rule(STATE *state,RULEPTR new_rule) {


  if ( state->RFirst == NULL) {
    state->RFirst = new_rule;
    state->RLast = new_rule;
  }

  else {

    state->RLast->next = new_rule;
    state->RLast = new_rule;
  }
}



/* The function "found" checks if the  new rule to  be entered
to the set of states is already found in the current state of not.*/


int lexer::found(STATE state,RULEPTR new_rule)

{
  RULEPTR Rulefound;

  if (state.RFirst != NULL)
    {
      Rulefound = state.RFirst;
      while (Rulefound != NULL)
        {

          if (Rulefound->left_hs == new_rule->left_hs)
            if (Rulefound->pred_ptr == new_rule->pred_ptr)
              if (Rulefound->right_hs == new_rule->right_hs)
                if (Rulefound->after_ptr == new_rule->after_ptr)
                  return 1;
          Rulefound = Rulefound->next;
        }
    }
  return 0;
}
```

```
/* The class  "parser" contains all the  function responsible for
the parsing procedure, construction the  tree, and generating  the
java bytecode*/

class parser {

  int last_non_terminal,code_value;
  STATE state[MAX_STATE], InitRules;
  RULEPTR Rulefound; /* used in tree_construction function */
  int  count,bytecode_line_nb;
  ID_SPECIFICATION t_id_specification[12];
                      //table of identifiers specification
  int last_identifier,last_main_block,last_parameter,current_block;
  int last_local_variable,current_local_variable,v_type1,v_type2;
  label_ptr t_fwd_labels[max_labels];
  loop_label_ptr t_loop_labels[10];
  int last_label,last_loop_label;
  FILE *out_f;

// PRIVATE FUNCTIONS

void add_non_terminal(char *symbol);
int parser::is_reserved_word(char *name);
void add_rule(STATE *state,RULEPTR new_Rule);//to add rule to a state
int found(STATE state,RULEPTR new_rule);//if rules already exists
void parser::affiche();
void tree_construction(RULEPTR Trule,REFPTR1P tempRef,RULEPTR *rule);
void parser::tree_link(RULEPTR *rule, struct WORDNODE **right_hs_tail);
void parser::traverse_tree(RULEPTR rule,struct WORDNODE *right_hs);

void define_function(char ident[]);
void define_parameter(char ident[]);
void define_variable(char ident[]);
void define_array(char ident[]);
void define_array_length(char ident[]);
void define_constant(char ident[]);
void define(char ident[],int kind);

int  exist_in_current_block(char ident[],int *index);
int is_local_symbol(char ident[],int *index);
int is_global_symbol(char ident[],int *index);

void parser::initialize_t_labels();

// PUBLIC FUNCTIONS

public:

void InitSymbolTable(void);
void FillInitState(void);
void Parser(lexer alg);
void tree();

};

/* The function 'initialize_t_labels' initializes the table of labels */

void parser::initialize_t_labels(void)
{
  int i;
  for (i=0;i<max_labels;i++)
    {
      t_fwd_labels[i] = NULL;
    }
  for (i=0;i<10;i++)
    {
      t_loop_labels[i] = NULL;
```

```
    }
  last_label = last_loop_label = 0;
}

/* The function "InitSymbolTable" initializes the symbol table. */

void parser::InitSymbolTable(void)
{
  int i;
  last_identifier = -1;
  last_main_block = -1;
  last_parameter = max_identifiers;
  current_block = MAIN;
  for(i=0;i<max_identifiers;i++)
    t_symbol_table[i] = NULL;
}


/* the function 'is_global_symbol' verifies of the token obtained
   is a global symbol or not */


int parser::is_global_symbol(char ident[],int *index)
{
  int found = 0;
  *index = 0;
  while ((*index <= last_main_block) && (! found))
    if (!strcmp(t_symbol_table[*index]->name,ident))
      found = 1;
    else
      (*index)++;

  return found;
}

/* The function 'is_local_symbol' verifies if the symbol obtained
   is a local symbol. it searches in symbol table */


int parser::is_local_symbol(char ident[],int *index)
{
  int i = last_identifier;
  int found = 0;

  while ((i > last_main_block) && (! found))
    if (!strcmp(t_symbol_table[i]->name,ident))
      {
        found = 1;
        *index = i;
      }
    else
      --i;
  return found;
}


/* The function 'exist_in_current_block' verifies if the identifier
   is declared as global or local in the block */

int parser::exist_in_current_block(char ident[],int *index)
{
  if (current_block == MAIN)
    return (is_global_symbol(ident,index));
  else
    return (is_local_symbol(ident,index));
}

/* the function 'define' saves a new identifier in the symbol table*/
```

```
void parser::define(char ident[],int kind)
{
  int index;

  t_ident_record_ptr id_record;
  if (last_identifier + 1 < last_parameter)
    if (!exist_in_current_block(ident,&index))
      {
        last_identifier++;
        if (current_block == MAIN)
          last_main_block++;
        id_record = new struct t_ident_record;
        id_record->name = (char*)malloc(strlen(ident)+1);
        strcpy(id_record->name,ident);
        id_record->type = kind;
        if(kind ==VARIABLE)
          {
            current_local_variable = last_local_variable;
            v_type2 = UNKNOWN_TYPE;
            id_record->id_info.variable.v_adr = last_local_variable++;
          }
         t_symbol_table[last_identifier] = id_record;
      }
    else
      {
        if(kind == VARIABLE)
          {
            current_local_variable =
              t_symbol_table[index]->id_info.variable.v_adr;
            v_type2 = t_symbol_table[index]->id_info.variable.v_typ;
          }
      }
}

/* The function 'define_variable' calls another function 'define' to
   define a variable in the symbol table */

void parser::define_variable(char ident[])
{
  define(ident,VARIABLE);
}

/* The function 'define_constant' calls another function 'define' to
   define a constant in the symbol table */
void parser::define_constant(char ident[])
{
  define(ident,CONSTANT);
}

/* The function 'define_function' calls another function 'define' to
   define a function in the symbol table */
void parser::define_function(char ident[])
{
  define(ident,FUNCTION);
}

/* The function 'define_array' calls another function 'define' to
   define a array in the symbol table */
void parser::define_array(char ident[])
{
  define(ident,ARRAY);
}

/* The function 'define_array_length' calls another function 'define'
   to define the length of an array  in the symbol table*/
void parser::define_array_length(char ident[])
{
```

```
  t_symbol_table[last_identifier]->id_info.array.length = atoi(ident);
}

void parser::define_parameter(char ident[])
{
}

/* The Function FillInitState() reads all the rules of productions
   describing the language grammar from the file and store them in the
   variable InitRules for later use in Earley's parser. It reads also
   the intermediate representation of bytecode to be generated */

void parser::FillInitState(void){

int index; // index for array
char temp[50];
struct WORDNODE  *tempWords,*tempLastWords;
struct RULE   *tempRule;
ifstream in("basic_gr");   // language grammar file

InitRules.RFirst=NULL;
InitRules.RLast=NULL;
last_non_terminal = last_reserved_word;

 for(index =0;index<12;index++)
   {
     t_id_specification[index].specification = 9999;
     t_id_specification[index].ident = 9999;
     t_id_specification[index].specification1 = 9999;
     t_id_specification[index].ident1 = 9999;
   }

 in >> temp;
 while (strcmp(temp,"%%"))
   {
     if (!strcmp(temp,"%%array"))
       index = ARRAY;
     else if (!strcmp(temp,"%%array1"))
       index = ARRAY1;
     else if (!strcmp(temp,"%%variable"))
       index = VARIABLE;
     else if (!strcmp(temp,"%%variable1"))
       index = VARIABLE1;
     else if (!strcmp(temp,"%%constant"))
       index = CONSTANT;
     else if (!strcmp(temp,"%%constant1"))
       index = CONSTANT1;
     else if (!strcmp(temp,"%%function"))
       index = FUNCTION;
     else if (!strcmp(temp,"%%function1"))
       index = FUNCTION1;
     else if (!strcmp(temp,"%%fun_par"))
       index = FUN_PAR;
     else if (!strcmp(temp,"%%fun_par1"))
        index = FUN_PAR1;
     else if (!strcmp(temp,"%%array_len"))
       index = ARRAY_LEN;
     else if (!strcmp(temp,"%%array_len1"))
       index = ARRAY_LEN1;

     in >> temp;
     if (strcmp(temp,"-"))
       add_non_terminal(temp);
     else
       code_value = 8888;  // if no id specification exists
     if(t_id_specification[index].specification != 9999)
       t_id_specification[index].specification1 = code_value;
     else
```

```
    t_id_specification[index].specification = code_value;
  in >> temp;
  in >> temp;
  add_non_terminal(temp);
  if(t_id_specification[index].ident != 9999)
    t_id_specification[index].ident1 = code_value;
  else
    t_id_specification[index].ident = code_value;
  in >> temp;
}

while (in >> temp)
 {
   tempRule = new struct RULE;
   tempRule->pred_ptr = 0;
   add_non_terminal(temp);
   tempRule->left_hs = code_value;
   tempRule->right_hs=NULL;
   tempRule->right_hs_tail=NULL;
   tempRule->after_ptr=NULL;
   tempRule->semantic_rule = NULL;
   tempRule->ref_ptr=NULL;
   tempRule->ref_ptr1=NULL;
   tempRule->next=NULL;

   in >> temp;
   while(strcmp(temp,"$$"))
     {
       tempWords = new struct WORDNODE;
       add_non_terminal(temp);
       tempWords->word =code_value;
       tempWords->value = NULL;
       tempWords->next=NULL;
       tempWords->desc = NULL;
       if(tempRule->right_hs==NULL)
         {
           tempWords->before = NULL;
           tempRule->right_hs=tempWords;
           tempLastWords=tempWords;
           tempRule->after_ptr=tempWords;
         }
       else
         {
           tempWords->before = tempLastWords;
           tempLastWords->next=tempWords;
           tempLastWords  =tempWords;
         }

       in >> temp;
     }

   if(tempRule->right_hs != NULL)
     tempRule->right_hs_tail = tempLastWords;

   in >> temp;
   while(strcmp(temp,"$$"))
     {
       in >> temp;  //read label if any
       tempWords= new struct WORDNODE;
       if  (strcmp(temp,"-"))
         {
           tempWords->label =(char *) malloc(strlen(temp)+1);
            strcpy(tempWords->label,temp);
         }
       else
         tempWords->label = NULL;

       in >> temp; // read symbol to be treated
```

```
          if  (strcmp(temp,"-"))
            {
              add_non_terminal(temp);
              tempWords->word = code_value;
            }
          else
            tempWords->word = 9999;

          in >> temp; // read value
          if (strcmp(temp,"-"))
            {
              tempWords->value =(char *) malloc(strlen(temp)+1);
              strcpy(tempWords->value,temp);
            }
          else
            tempWords->value = NULL;
          in >> temp;   // read instruction
            if (strcmp(temp,"-"))
            {
              tempWords->instruction =(char *)malloc(strlen(temp)+1);
              strcpy(tempWords->instruction,temp);
            }
          else
            tempWords->instruction = NULL;
            in >> temp;
          tempWords->next=NULL;
          if(tempRule->semantic_rule==NULL)
            {
              tempRule->semantic_rule=tempWords;
              tempLastWords=tempWords;
            }
          else
            {
              tempLastWords->next=tempWords;
              tempLastWords  =tempWords;
            }
          in >> temp;
        }

      in >> temp;
      tempRule->priority = atoi(temp);

      if(InitRules.RFirst==NULL)
        {
          InitRules.RFirst=tempRule;
          InitRules.RLast=tempRule;
        }
      else
        {
          InitRules.RLast->next=tempRule;
          InitRules.RLast=tempRule;
        }
    }
}

/* this function is the parser. it parses each token obtained from a
   call to 'lex_analyser' */

void parser::Parser(lexer alg)
{

  RULEPTR tempRule,tempRule1,tempRule2;
  struct WORDNODE *m,*temp_right_hs;
  int eof_error,error_occur,symbol;
  int tokens[2],symbol_read;
  char *value;
  error_occur = 1;
  count = 0;
```

```
for (count=0;count<MAX_STATE;count++)
  {
    state[count].RFirst = state[count].RLast = NULL;
  }
count = 0;

state[0].RFirst = new struct RULE;
state[0].RFirst->pred_ptr = 0;
state[0].RFirst->priority = InitRules.RFirst->priority;
state[0].RFirst->left_hs = InitRules.RFirst->left_hs;
state[0].RFirst->right_hs = InitRules.RFirst->right_hs;
state[0].RFirst->right_hs_tail = InitRules.RFirst->right_hs_tail;
state[0].RFirst->after_ptr = InitRules.RFirst->after_ptr;
state[0].RFirst->semantic_rule = InitRules.RFirst->semantic_rule;
state[0].RFirst->ref_ptr = NULL;
state[0].RFirst->ref_ptr1 = NULL;
state[0].RFirst->next = NULL;
state[0].RLast = state[0].RFirst;

line = 0;  // initialize the number of lines in the source program
alg.read_newline(); // read new line of code

eof_error = 0;

while (eof != NULL)
  {
    if (! eof_error)      // if no error get token
      {
        value = (char*) malloc(30);
        symbol = alg.lex_analyser(tokens,value);
      }


   if (eof != NULL)
      eof_error = 1;

    tempRule = state[count].RFirst;

    while (tempRule != NULL) {
      if (tempRule->after_ptr != NULL)
        {
          if ((tempRule->after_ptr->word == symbol) ||
              (tempRule->after_ptr->word == tokens[0]) ||
              (tempRule->after_ptr->word == tokens[1]))
            {
             /* if state i in A->a .Xb,j and next input symbol is*/
              if (tempRule->after_ptr->word == symbol)
                 symbol_read = symbol;
              else if        (tempRule->after_ptr->word == tokens[0])
                symbol_read = tokens[0];
              else if        (tempRule->after_ptr->word == tokens[1])
                symbol_read = tokens[1];
              else
                symbol_read = 9999;

              error_occur = 1;
              eof_error = 0;
              tempRule1 = new struct RULE;
              tempRule1->left_hs = tempRule->left_hs;
              tempRule1->pred_ptr = tempRule->pred_ptr;
              tempRule1->priority = tempRule->priority;
              if ((symbol >= last_reserved_word) &&
                                        (symbol < last_terminal))
                {
              /*if symbol is a terminal passed from lexer to parser*/
                  tempRule1->right_hs = NULL;
                  tempRule1->right_hs_tail = NULL;
```

```
        temp_right_hs = tempRule->right_hs;
        while(temp_right_hs != NULL)
          {
            if (tempRule1->right_hs == NULL)
              {
               m=tempRule1->right_hs =new struct WORDNODE;
               m->before = NULL;
               tempRule1->right_hs->word=
                                    temp_right_hs->word;
                if((temp_right_hs->word>=
                                    last_reserved_word)
                  && (temp_right_hs->word<last_terminal)
                  && (temp_right_hs->word==symbol_read)
                  && (temp_right_hs->value==NULL))
                  {
                    tempRule1->right_hs->value = value;
                  }
                else
                  tempRule1->right_hs->value =
                                    temp_right_hs->value;
              }
            else
             {
              m->next =  new struct WORDNODE;
              m->next->before = m;
              m = m->next;
              m->word = temp_right_hs->word;
              if ((temp_right_hs->word>=last_reserved_word)
                  && (temp_right_hs->word < last_terminal)
                  && (temp_right_hs->word == symbol_read)
                  && (temp_right_hs->value ==NULL))
                 {
                    m->value = value;
                 }
              else
                 m->value = temp_right_hs->value;
             }
            temp_right_hs = temp_right_hs->next;
            m->next = NULL;
            m->desc = NULL;
          }
        if(tempRule1->right_hs !=NULL)
          tempRule1->right_hs_tail = m;
        else
          tempRule1->right_hs_tail = NULL;
      }
    else
      {
        tempRule1->right_hs = tempRule->right_hs;
        tempRule1->right_hs_tail=tempRule->right_hs_tail;
      }
    tempRule1->semantic_rule = tempRule->semantic_rule;
    tempRule1->after_ptr = tempRule->after_ptr->next;
    tempRule1->ref_ptr = tempRule;
    tempRule1->ref_ptr1 = NULL;
    tempRule1->next= NULL;
    if (! found(state[count+1],tempRule1))
                  //does rule already exists
      {
        add_rule(&state[count+1],tempRule1);
      }
  }
else //   if state i in A-> a.Bb,j then predict B-> .y,i
  {
    tempRule2 = InitRules.RFirst;
    while (tempRule2 != NULL)
      {
       if (tempRule->after_ptr->word == tempRule2->left_hs)
```

```
                    {
                     tempRule1 = new struct RULE;
                     tempRule1->left_hs = tempRule2->left_hs;
                     tempRule1->pred_ptr = count;
                     tempRule1->priority = tempRule2->priority;
                     tempRule1->right_hs = tempRule2->right_hs;
                     tempRule1->right_hs_tail=tempRule2->right_hs_tail;
                     tempRule1->semantic_rule=tempRule2->semantic_rule;
                     tempRule1->after_ptr = tempRule2->after_ptr;
                     tempRule1->ref_ptr = tempRule;
                     tempRule1->ref_ptr1 = NULL;
                     tempRule1->next= NULL;
                     if (! found(state[count],tempRule1))
                        {
                          add_rule(&state[count],tempRule1);
                        }
                    }
                   tempRule2 = tempRule2->next;
                  }
              }
            }
          else // if state i contains A->y.,j then go to state j
            {
              tempRule2 = state[tempRule->pred_ptr].RFirst;
              while (tempRule2 != NULL)
                {
                  if (tempRule2->after_ptr!=NULL)
                    {
                      if (tempRule2->after_ptr->word==tempRule->left_hs)
                        {
                          tempRule1 = new struct RULE;
                          tempRule1->left_hs = tempRule2->left_hs;
                          tempRule1->pred_ptr = tempRule2->pred_ptr;
                          tempRule1->priority = tempRule2->priority;
                          tempRule1->right_hs = tempRule2->right_hs;
                          tempRule1->right_hs_tail=tempRule2->right_hs_tail;
                          tempRule1->semantic_rule=tempRule2->semantic_rule;
                          tempRule1->after_ptr=tempRule2->after_ptr->next;
                          tempRule1->ref_ptr = tempRule;
                          tempRule1->ref_ptr1 = tempRule2;
                          tempRule1->next= NULL;
                          if (!found(state[count],tempRule1))
                            {
                              add_rule(&state[count],tempRule1);
                            }
                          else
                            {
                              if ((tempRule->priority >
                                   Rulefound->ref_ptr->priority) &&
                                   ( Rulefound->ref_ptr->priority!=0))
                                {
                                  Rulefound->ref_ptr = tempRule;
                                }
                            }
                        }
                    }
                  tempRule2 = tempRule2->next;
                }
            }
          tempRule = tempRule->next;
        }
       count ++;
      }
}

int parser::is_reserved_word(char *name)
{
  int i,j,k;
```

```
  j = last_reserved_word - 1;
  i = 0;


  do {
    k = (i+j)/2;
    if (strcmp(name,t_reserved_words[k]) > 0)
      i = k+1;
    else
      j = k-1;
  } while ((strcmp(name,t_reserved_words[k])) && (i <= j));
  if (!strcmp(name,t_reserved_words[k]))
    code_value = k;
  return (!strcmp(name,t_reserved_words[k]));
}


/* this function 'add_non_terminal' looks for a symbol in the table of
   reserved word. If it is not found, it adds it to the table */


void parser::add_non_terminal(char *symbol)
{
  int found = 0;

  if (!is_reserved_word(symbol))
    {
      for(code_value = last_non_terminal;code_value<nb_reserved_words;code_value++)
        {
          if (t_reserved_words[code_value] == NULL)
            break;

          if (!strcmp(t_reserved_words[code_value],symbol))
            {
              found = 1;
              break;
            }
        }

      if (!found)
        {
          t_reserved_words[code_value]=(char*)malloc(strlen(symbol)+1);
          strcpy(t_reserved_words[code_value],symbol);
        }
    }
}

void parser::add_rule(STATE *state,RULEPTR new_rule) {

  if ( state->RFirst == NULL) {
    state->RFirst = new_rule;
    state->RLast = new_rule;
  }

  else {

    state->RLast->next = new_rule;
    state->RLast = new_rule;
  }
}

int parser::found(STATE state,RULEPTR new_rule)
{
   if (state.RFirst != NULL)
    {
      Rulefound = state.RFirst;
      while (Rulefound != NULL)
```

```
        {

          if (Rulefound->left_hs == new_rule->left_hs)
            if (Rulefound->pred_ptr == new_rule->pred_ptr)
              if (Rulefound->right_hs == new_rule->right_hs)
                if (Rulefound->after_ptr == new_rule->after_ptr)
                  return 1;
          Rulefound = Rulefound->next;
        }
    }
  return 0;
}


/* this function 'tree_link' links the parse tree obtained as a linked
   list of rules by callin the function 'tree_construction' */

void parser::tree_link(RULEPTR *rule, struct WORDNODE **right_hs_tail)
{

  if ((*right_hs_tail != NULL) &&  (*rule != NULL))
    {
      if (((*right_hs_tail)->word >= 0)
          && ((*right_hs_tail)->word < last_terminal))
        {          //if symbol is terminal
          tree_link(&(*rule),&(*right_hs_tail)->before);
        }
      else
        {
          (*right_hs_tail)->desc = (*rule)->next;
          tree_link(&(*rule)->next,
                        &(*right_hs_tail)->desc->right_hs_tail);
          (*rule)->next=(*rule)->next->next;
          tree_link(&(*rule),&(*right_hs_tail)->before);
        }
    }
}

/* The function 'traverse_tree' traverses the parse tree in order
   to generate the java bytecodes */

void parser::traverse_tree(RULEPTR rule,struct WORDNODE *right_hs)
{
  struct WORDNODE *tempn;
  int i;
  char *value;

  tempn = NULL;
  if (right_hs != NULL)
    {
      printf(" \nRule : %s ----> ",t_reserved_words[rule->left_hs]);
      for (tempn = rule->right_hs;tempn != NULL;tempn=tempn->next)
        {
          printf(" [%s ",t_reserved_words[tempn->word]);
          if (tempn->value != NULL)
            printf(" %s]",tempn->value);
          else
            printf("null]");
        }
      printf("\n");
      printf("[%s ",t_reserved_words[right_hs->word]);
      if (right_hs->value!= NULL)
        printf("%s]\n\n",right_hs->value);
      else
        printf("null]\n\n");

      if((right_hs->word >=last_reserved_word)&&
                          (right_hs->word<last_terminal))
```

```
  {
    for(i=0;t_id_types[i];i++)
      if (t_id_types[i]->symbol == right_hs->word)
        {
          v_type1 = v_type2;
          v_type2 = t_id_types[i]->type;
          break;
        }
  }

for(i=0;i<12;i++)
  {
    if(((rule->left_hs==t_id_specification[i].specification)
      &&(right_hs->word == t_id_specification[i].ident))
       ||
       ((rule->left_hs == t_id_specification[i].specification1)
      &&(right_hs->word == t_id_specification[i].ident1))
       ||
       ((8888 ==  t_id_specification[i].specification) &&
        (right_hs->word == t_id_specification[i].ident))
       ||
       ((8888 == t_id_specification[i].specification1) &&
        (right_hs->word == t_id_specification[i].ident1)))
      {
        switch(i) {
        case 0:
          define_array(right_hs->value);
          //                array_name = ;
          break;
        case 1:
          define_array(right_hs->value);
          //                array_name = ;
          break;
        case 2:
          define_variable(right_hs->value);
          sprintf(right_hs->value,"%d",current_local_variable);
          break;
        case 3:
          define_variable(right_hs->value);
          sprintf(right_hs->value,"%d",current_local_variable);
          break;
        case 4:
          define_constant(right_hs->value);
          break;
        case 5:
          define_constant(right_hs->value);
          break;
        case 6:
          define_function(right_hs->value);
          //                function_name = ;
          break;
        case 7:
          define_function(right_hs->value);
          //                function_name = ;
          break;
        case 8:
          define_parameter(right_hs->value);
          break;
        case 9:
          define_parameter(right_hs->value);
          break;
        case 10:
          define_array_length(right_hs->value);
          break;
        case 11:
          define_array_length(right_hs->value);
          break;
        }
```

```
        break;
      }
  }

if (right_hs->desc != NULL)
  {
    traverse_tree(right_hs->desc,right_hs->desc->right_hs);
  }

if(rule->semantic_rule != NULL)
  {
    if (rule->semantic_rule->label != NULL)
      if (!strcmp(rule->semantic_rule->label,"L1FOR"))
        {
         t_loop_labels[last_loop_label] = new struct loop_label;
         t_loop_labels[last_loop_label]->value = bytecode_line_nb;
         last_loop_label++;
         rule->semantic_rule = rule->semantic_rule->next;
        }
    if (right_hs->word ==  rule->semantic_rule->word)
      {
        if (rule->semantic_rule->value != NULL)
          if(!strcmp(rule->semantic_rule->value,"v"))
            if (rule->semantic_rule->instruction == NULL)
              if (right_hs->value != NULL)
                {
                  rule->semantic_value = right_hs->value;
                }
              else
                rule->semantic_value =
                                right_hs->desc->semantic_value;
            else /* if instruction exists */
             {
              if (right_hs->value != NULL)
                value = right_hs->value;
              else
                value =  right_hs->desc->semantic_value;
              if (rule->semantic_rule->next != NULL)
               if (rule->semantic_rule->next->value != NULL)
                if(!strcmp(rule->semantic_rule->next->value,"s"))
                    {
                      rule->semantic_rule->next->value1=value;
                    }
              if (!strcmp(rule->semantic_rule->instruction,
                  "xload"))
                {
                    if (v_type2 == INTEGER)
                      fprintf(out_f,"%d iload_%s\n",
                                bytecode_line_nb,value);
                    else  if (v_type2 == FLOAT)
                      fprintf(out_f,"%d fload_%s\n",
                                bytecode_line_nb,value);
                    else if (v_type2 == DOUBLE)
                      fprintf(out_f,"%d dload_%s\n",
                                bytecode_line_nb,value);
                 }
              else
                    fprintf(out_f,"%d %s %s\n",bytecode_line_nb,
                       rule->semantic_rule->instruction,value);
              bytecode_line_nb ++;
             }
        rule->semantic_rule = rule->semantic_rule->next;
        while ((rule->semantic_rule != NULL) &&
               (rule->semantic_rule->value !=NULL) &&
               (rule->semantic_rule->instruction != NULL))
          {
           if(!strcmp(rule->semantic_rule->instruction,"xstore"))
              {
```

```
    for(i=0;t_symbol_table[i];i++)
      if (t_symbol_table[i]->id_info.variable.v_adr
            == atoi(rule->semantic_rule->value1))
        {
          t_symbol_table[i]->id_info.variable.v_typ= v_type2;
          break;
        }
    if (v_type2 == INTEGER)
     fprintf(out_f,"%d istore_%s\n",bytecode_line_nb,
            rule->semantic_rule->value1);
    else  if (v_type2 == FLOAT)
     fprintf(out_f,"%d fstore_%s\n",bytecode_line_nb,
            rule->semantic_rule->value1);
    else  if (v_type2 == DOUBLE)
     fprintf(out_f,"%d dstore_%s\n",bytecode_line_nb,
            rule->semantic_rule->value1);
  }
else
 if(!strcmp(rule->semantic_rule->instruction, "xload"))
  {
    if (v_type2 == INTEGER)
     fprintf(out_f,"%d iload_%s\n",bytecode_line_nb,
            rule->semantic_rule->value1);
    else  if (v_type2 == FLOAT)
     fprintf(out_f,"%d fload_%s\n",bytecode_line_nb,
            rule->semantic_rule->value1);
    else if (v_type2 == DOUBLE)
     fprintf(out_f,"%d dload_%s\n",bytecode_line_nb,
            rule->semantic_rule->value1);
  }
else
  if(!strcmp(rule->semantic_rule->instruction,"xmul")
  || !strcmp(rule->semantic_rule->instruction,"xsub")
  || !strcmp(rule->semantic_rule->instruction,"xadd")
  || !strcmp(rule->semantic_rule->instruction,"xdiv")
     )
    {
      if ((v_type2 == INTEGER)&&(v_type1 == INTEGER))
        fprintf(out_f,"%d i%s\n",bytecode_line_nb,
              ++rule->semantic_rule->instruction);
      else  if ((v_type2 == FLOAT)||(v_type1==FLOAT))
        {
          fprintf(out_f,"%d f%s\n",bytecode_line_nb,
              ++rule->semantic_rule->instruction);
          v_type2 = v_type1 = FLOAT;
        }
      else if((v_type2 == DOUBLE)||(v_type1==DOUBLE))
        {
          fprintf(out_f,"%d d%s\n",bytecode_line_nb,
              ++rule->semantic_rule->instruction);
          v_type2 = v_type1 = DOUBLE;
        }
      --rule->semantic_rule->instruction;
    }
  else
   {
    if (!strcmp(rule->semantic_rule->value,"L2FOR"))
     {
      t_loop_labels[last_loop_label] =
          new struct loop_label;
      t_loop_labels[last_loop_label]->adr=last_label;
         fprintf(out_f,"%d %s ",bytecode_line_nb,
              rule->semantic_rule->instruction);
      t_fwd_labels[last_label] = new struct label;
      t_fwd_labels[last_label]->pos =
           (fpos_t*)malloc(sizeof(fpos_t));
      fgetpos(out_f,t_fwd_labels[last_label]->pos);
      fprintf(out_f,"      \n");
```

```
                    last_label++;
                    last_loop_label++;
                   }
                  else if (!strcmp(rule->semantic_rule->value,"EL1FOR"))
                    {
                     rule->semantic_rule->value1 = new char;
                     sprintf(rule->semantic_rule->value1,"%d",
                        t_loop_labels[last_loop_label-2]->value);
                     t_loop_labels[last_loop_label-2]= NULL;
                     fprintf(out_f,"%d %s %s\n",bytecode_line_nb,
                             rule->semantic_rule->instruction,
                             rule->semantic_rule->value1);
                    }
                  else
                    {
                     if (!strcmp(rule->semantic_rule->value,"s"))
                       fprintf(out_f,"%d %s %s\n",bytecode_line_nb,
                           rule->semantic_rule->instruction,value);
                     else
                       fprintf(out_f,"%d %s %s\n",bytecode_line_nb,
                                  rule->semantic_rule->instruction,
                                  rule->semantic_rule->value1);
                      }
                    }
                bytecode_line_nb ++;
                rule->semantic_rule = rule->semantic_rule->next;
              }

            if ((rule->semantic_rule != NULL) &&
                (rule->semantic_rule->label != NULL))
              if (!strcmp(rule->semantic_rule->label,"EL2FOR"))
                {
                 t_fwd_labels[
                    t_loop_labels[last_loop_label-1]->adr]->value
                                              = bytecode_line_nb;
                  t_loop_labels[last_loop_label-1] = NULL;
                  last_loop_label = last_loop_label - 2;
                  rule->semantic_rule = rule->semantic_rule->next;
                }
          }
        else
          if ((right_hs->word != rule->semantic_rule->word) &&
              (rule->semantic_rule->word != 9999))
            {
              tempn = rule->semantic_rule;

              while((tempn!=NULL) && (tempn->word!=right_hs->word))
                {
                  tempn = tempn->next;
                }
              if (tempn != NULL)
                if (tempn->word == right_hs->word)
                  {
                    if (right_hs->desc != NULL)
                      tempn->value1=right_hs->desc->semantic_value;
                    else
                      tempn->value1 = right_hs->value;
                    if (tempn->next != NULL)
                      if (tempn->next->value != NULL)
                        if (!strcmp(tempn->next->value,"s"))
                          tempn->next->value1 = tempn->value1;
                  }
            }
      }
      traverse_tree(rule,right_hs->next);
  }
}
```

```
/* This function calls 'tree_constructio' to build in the parse tree
   and saves it to a linked list of rule. This linked list is then
   transformed to real tree by calling the 'tree_link' function */

void parser::tree()

{
  RULEPTR tempRule,rule;
  REFPTR1P tempRef;
  struct WORDNODE *tempn;
  tempRef = NULL;
  /* fill tempRule with the contents of the initial rule and
     check if temprule is found in the last state generated
     by parsing the input symbols */

  tempRule =  (RULEPTR)malloc(sizeof(struct RULE));
  tempRule->pred_ptr  = 0;
  tempRule->left_hs = InitRules.RFirst->left_hs;
  tempRule->right_hs = InitRules.RFirst->right_hs;
  tempRule->right_hs_tail = InitRules.RFirst->right_hs_tail;
  tempRule->after_ptr = NULL;

  if (found(state[--count],tempRule))
    {
      tree_construction(Rulefound,tempRef,&rule);
      free(Rulefound);
      free(tempRef);
      free(InitRules.RFirst);
      free(InitRules.RLast);
    }

  for(count = 0;state[count].RFirst;count++)
    {
      free(state[count].RFirst);
      free(state[count].RLast);
    }

  tree_link(&rule,&rule->right_hs_tail);
  last_local_variable = 0;
  bytecode_line_nb = 0 ;
  out_f = fopen("result","w");
  initialize_t_labels();
  traverse_tree(rule,rule->right_hs);

  for(count=0;t_fwd_labels[count];count++)
    {
      fsetpos(out_f,t_fwd_labels[count]->pos);
      fprintf(out_f,"%d",t_fwd_labels[count]->value);
    }

  for(count=0;t_symbol_table[count];count++)
    {
      printf("%s %d\n",t_symbol_table[count]->name, t_symbol_table[count]->type);
      if (t_symbol_table[count]->type == ARRAY)
        printf("arr_len: %d\n",t_symbol_table[count]->id_info.array.length);
    }
  }

void parser::tree_construction(RULEPTR Trule,REFPTR1P tempRef,RULEPTR *rule)
{
REFPTR1P tempL;
struct WORDNODE *tempw;

if (Trule->after_ptr == NULL)
  {
    *rule = new struct RULE;
   (*rule)->left_hs = Trule->left_hs;
   (*rule)->semantic_rule = Trule->semantic_rule;
```

```
    if (Trule->right_hs != NULL)
      {
        tempw= new struct WORDNODE;
        (*rule)->right_hs = tempw;
        tempw->before = NULL;
        while(Trule->right_hs != NULL)
          {
            tempw->word = Trule->right_hs->word;
            tempw->value = Trule->right_hs->value;
            tempw->desc = NULL;
            if (Trule->right_hs->next != NULL)
              {
                tempw->next= new struct WORDNODE;
                tempw->next->before = tempw;
                tempw = tempw->next;
              }
            Trule->right_hs = Trule->right_hs->next;
          }
        tempw->next = NULL;
        (*rule)->right_hs_tail = tempw;
      }
    else
      (*rule)->right_hs_tail = NULL;

    rule = &(*rule)->next;
  }

if (tempRef == NULL)
  {
    tempRef = (REFPTR1P)malloc(sizeof(struct RefPtr1));
    tempRef->tempList = Trule->ref_ptr1;
    tempRef->next = NULL;
    tree_construction(Trule->ref_ptr,tempRef,rule);
    free(Trule);
  }
else
  {
    if (Trule->ref_ptr > tempRef->tempList)
      {
        if (Trule->ref_ptr1 != NULL)
          {
            tempL = (REFPTR1P)malloc(sizeof(struct RefPtr1));
            tempL->tempList = Trule->ref_ptr1;
            tempL->next = tempRef;
            tree_construction(Trule->ref_ptr,tempL,rule);
            free(Trule);
          }
        else
          {
            tree_construction(Trule->ref_ptr,tempRef,rule);
            free(Trule);
          }
      }
    else
      {
        if (tempRef->tempList->ref_ptr != NULL)
          {
            tree_construction(tempRef->tempList,tempRef->next,rule);
            free(Trule);
          }
        else
          *rule = NULL;

      }
  }
}
```

```
void main(int argc,char *argv[])
{
  lexer alg;
  parser alg1;
  alg.program = fopen(argv[1],"r");
  error_f = fopen("error","w");
  alg.FillInitState();
  alg1.InitSymbolTable();
  alg1.FillInitState();
  alg1.Parser(alg);
  alg1.tree();
}
```

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice Hall, New Jersey, 1972. London Mathematical Society Student Texts 1.(pbk).

[3] J.P. Bennet. *Introduction to Compiling Techniques.* McGraw-Hill International Series in Software Engineering, 1990.

[4] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102, 1970.

[5] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler.* Benjamin/Cummings, California, 1989.

[6] Gosling J. *Java Intermediate Bytecodes.* ACM SIGPLAN Workshop on Intermediate Representations, California, 1995.

[7] C. Perkins L. Lemay. *Teach Yourself Java in 21 Days.* SAMS, California USA, 1996.

[8] M. E. Lesk. Lex - a lexical analyzer generator. Science technical report 39, AT&T Bell Laboratories, 1975.

[9] J. R. Levine and T. Mason. *Lex & Yacc.* O'Reilly & Associates, Sebastopol, Canada, 1992.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, California USA, 1997.

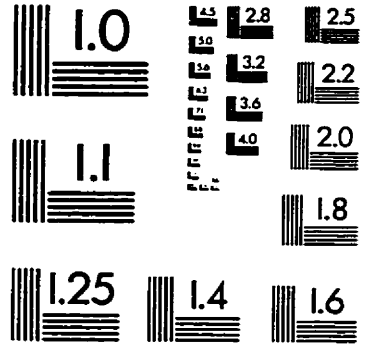
[11] H. Schildt. *C++: The Complete Reference.* McGraw Hill, California, USA, 1995.

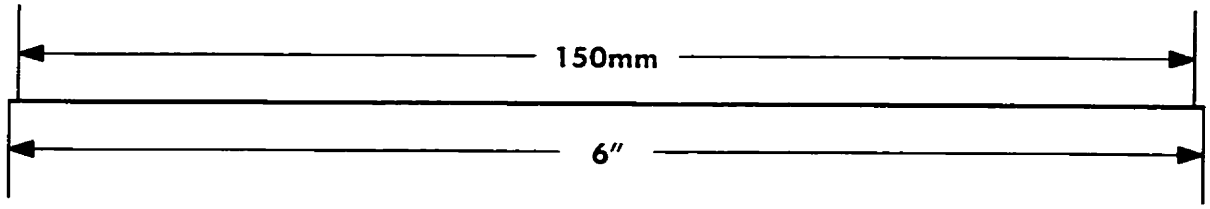
[12] N. Silverio. *Realiser un compilateur.* Eyrolles, Paris, 1994.

[13] J.P. Tremblay and P. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, revised edition, 1985.

[14] R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley, California, USA, 1995.

TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989