INFORMATION TO USERS

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

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

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

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

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

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

AN OBJECT-ORIENTED PARSER GENERATOR FOR LL(1) GRAMMARS

Hassan Manasfi

A thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

December 1999

Abstract

An Object-Oriented Parser Generator for LL(1) Grammars

Hassan Manasfi

In this thesis we investigate the use of Object-Oriented techniques to build both a parser and a parser generator for an LL(1) attributed grammars.

When top-down parsing is being used, a node of the parse tree is expanded into several other nodes. Thinking in terms of objects, an object in the tree constructs other objects. An object, which is an instance of a class, on the tree corresponds to an instance of the corresponding symbol in a program. This means that each symbol in the language corresponds to a class or an instance of a class. So in order to parse a document and construct a parse tree for it, we begin by constructing the root of that tree by creating an object representing the nonterminal start symbol. The constructor of the root would in turn construct the subtrees of the root.

The attributes of a symbol correspond to data members inside the class representing that symbol. If we use objects to represent nodes of the parse tree (or abstract syntax tree), then these are the natural objects to send messages to. A semantic action is a message sent to an object, and is represented by a member function inside a class.

The parsing approach is explained. The design and implementation of key parts of the parser are shown. Also the definition of the language's grammar which contains regular expressions and offer an easy notation to specify semantic actions is documented.

Many parser generators that claimed to be object-oriented in the past wrapped up a finite state machine, a push-down automaton, or a decision table in classes. Having those components wrapped up in classes does not offer any boost to the comprehensibility of the parser generator, nor does it reflect the relation between the parser generator and its product: the generated parser, which are of the most important objectives of object-oriented programming.

A grammar is nothing but a set of components: rules, terminals, nonterminals, regular expressions, semantic actions. Classes represent very well a grammar's components. By representing every component with a class, the set of objects obtained can cooperate together on the generation of the parser by each producing the code about itself. The set of objects representing the grammar relate directly to the generated parser since there is a clear match between every object and a class of the generated parser. The parser generator benefits from several advantages of OOP.

The design and implementation of the parser generator are explained and discussed. Sample output from the parser generator is provided.

Acknowledgments

"And your Lord has decreed that you worship none but Him. And that you be dutiful to your parents. If one of them or both of them attain old age in your life, say not to them a word of disrespect, nor shout at them but address them in terms of honour. And lower unto them the wing of submission and humility through mercy, and say: 'My Lord! Bestow on them Your Mercy as they did bring me up when I was small.' Your Lord knows best what is in your inner-selves. If you are righteous, then, verily, He is Ever Most Forgiving to those who turn unto Him again and again in obedience, and in repentance." [Qur'aan, 17:23-25]

I know there are not enough words to thank my parents. But still, thank you, I owe you everything. I would like also to thank my brothers and sisters and their families. You complete me and I complete you.

I would like to thank my thesis supervisor, Dr. Peter Grogono, for his valuable advice, and guidance during my work on this thesis. His careful supervision and patience made this work possible. My work with Dr. Grogono was a very pleasant and enriching experience. He is the only teacher I know that all his students agree "Oh! Grogono, he is very nice".

Contents

List of Figures

Chapter 1

Introduction

The study of parsers and parser generators necessitates basic knowldge about the phases of compiler and language definitions.

1.1 Phases of a Compiler

A typical decomposition of the phases of a compiler is shown in Figure 1. Many tools have been invented to aid in the generation of compilers by producing some of the phases automatically. These tools are often referred to collectively as compiler tools. Figure 2 shows us some compiler tools and their connection with the compiler construction phases. For the front end of a compiler these tools are often termed lexical analyzer generator, syntax analyzer generator (or parser generator), and semantic analyzer generator. Some work has been done also on code generator generators.

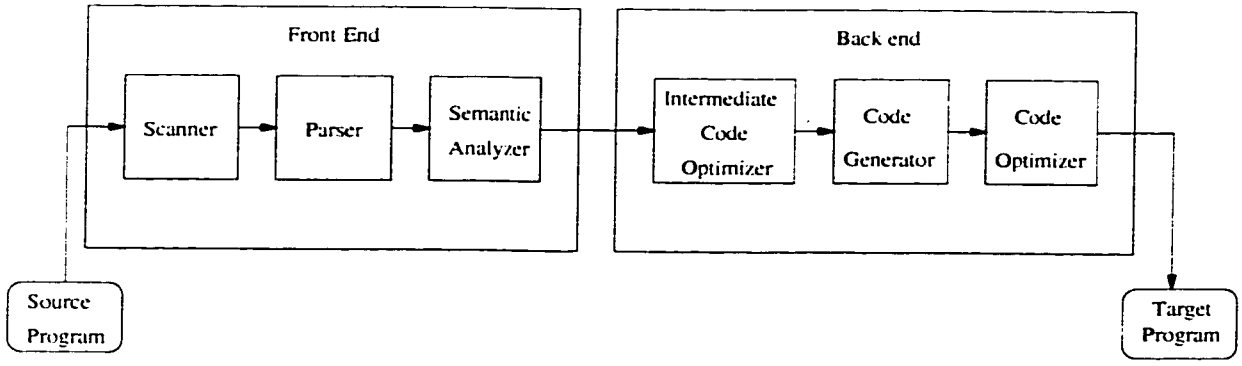


Figure 1: Phases of a compiler

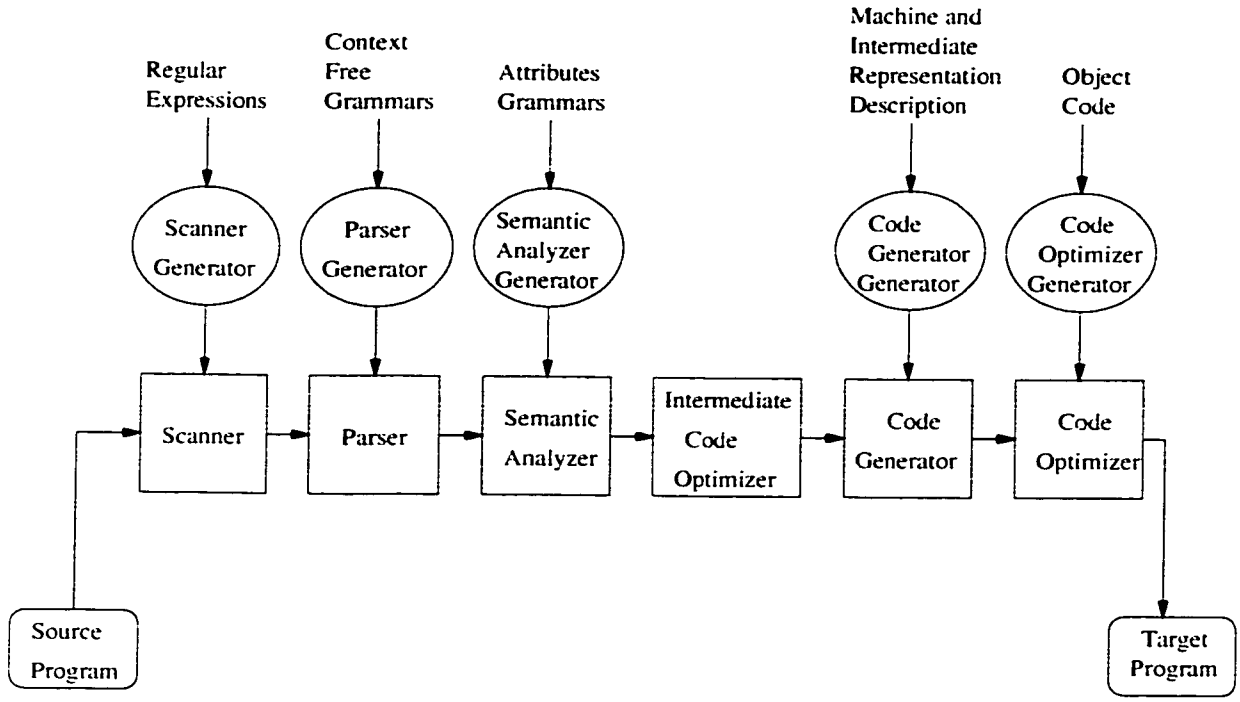


Figure 2: Compiler tools

1.2 Language Definition

The definition of a programming language sets the ground rules on how to interpret programs: What symbols form valid fragments, whether a program is legal, and what a program means in terms of what it achieves.

The complete definition of a programming language consists of two parts: the syntax and the semantics. The syntax defines the set of well-formed programs, and the semantics defines the meaning of each of these programs.

Natural languages such as English are often described by a grammar which groups words into syntactic categories such as subjects, predicates, prepositional phrases, etc. The standard method of defining the syntax of programming languages is the context-free grammar (CFG) that was introduced by Chomsky [Cho56] [Cho59].

The CFG is a structure $G = (N, T, P, S)$, where

1. N is a finite set of nonterminal symbols.
2. T is a finite set of terminal symbols, where $N \cap T = \emptyset$.
3. P is a finite subset of $N \times (N \cup T)^*$
4. S is a symbol of N designated as the start symbol.

An element of P is called a production and is normally written as $X \rightarrow \gamma$ instead of (X, γ), where the arrow, $\rightarrow$, means "is defined as". Figure 3 shows a definition of a CFG for simple arithmetic

expressions.

G = ({E, T, F}, {+, *,), (, number}, {
E ⟶ E + F
E ⟶ T
T ⟶ T * F
T ⟶ F
F ⟶ (E)
F ⟶ number
}, E)

Figure 3: CFG for simple arithmetic expressions.

Productions with the same nonterminal on the left can have their right side grouped with the alternative right sides separated by the symbol |, which we read as " or. "

The **Backus-Naur Form (BNF) notation** was introduced in the Algol 60 report [BBG+60]. This was the first use of BNF for the purpose of defining languages. The BNF notation is a variant of CFG. This form's main properties are the use of angle brackets to enclose non-terminals and of ::= to denote "may produce". In some variants, the rules are terminated by a semicolon. Since the introduction of BNF the theory of context free languages has become well understood and this has led to better language design and better parsing algorithms. In turn, this has led to the development of practical parser generators. A small example of a BNF grammar for simple arithmetic expressions is shown in Figure 4.

< expr >	::=	< term >	| < expr > + < expr >
< term >	::=	< factor >	| < term > * < factor >
< factor >	::=	(< expr >)	| < number > | < ident >
< number >	::=	< digit >	| < number > < digit >
< ident >	::=	< letter >	| < ident > < letter > | < ident > < digit >
< letter >	::=	A | B | C | ··· | Z	
< digit >	::=	0 | 1 | 2 | ··· | 9	

Figure 4: BNF Grammar for simple arithmetic expressions.

Extended BNF Grammars(EBNFG) are a more compact and more readable form of CF grammars. EBNFGs introduce special short-hands for frequently used constructions. Braces, { }, are often used to represent 0 or more occurrences of items on the left-hand side of a rule, and brackets, [ ], to represent optional items. Sometimes [ ]* is used in place of { }. Other notations also allow for the specification of one or more iterations such as { }+. Figure 5 shows how to represent the grammar in figure 4 as an EBNFG.

$$
\begin{array}{lcl}
\langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \ \{ \ + \ \langle \text{term} \rangle \ \} \\
\langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \ \{ \ * \ \langle \text{factor} \rangle \ \} \\
\langle \text{factor} \rangle & ::= & (\ \langle \text{expr} \rangle \ ) \mid \langle \text{number} \rangle \mid \langle \text{ident} \rangle \\
\langle \text{number} \rangle & ::= & \{ \langle \text{digit} \rangle \}+ \\
\langle \text{ident} \rangle & ::= & \langle \text{letter} \rangle \mid \langle \text{ident} \rangle \ \langle \text{letter} \rangle \mid \langle \text{ident} \rangle \ \langle \text{digit} \rangle \\
\langle \text{letter} \rangle & ::= & A \mid B \mid C \mid \cdots \mid Z \\
\langle \text{digit} \rangle & ::= & 0 \mid 1 \mid 2 \mid \cdots \mid 9
\end{array}
$$

Figure 5: EBNF Grammar for simple arithmetic expressions.

1.3 Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. Therefore, the string of tokens has to be examined to determine whether it obeys certain structural conventions explicit in the syntactic definition of the language. The syntactic structure of a given string provides invaluable information for the computation process. For example, the syntactic structure of the expression A + B * C reflects that B and C are first multiplied and then the result is added to A. No other ordering of the operations will produce the desired calculation.

If one were to check whether this English sentence

"The student is in the class"

is grammatically correct, one would draw the parsing diagram shown in Figure 6. Similarly, the output of parsing an expression is a tree which represents the syntactic structure inherent in that expression. Even though the tree may not be actually constructed, it is helpful to think of such a tree as being constructed.

For example, Figure 7 shows us the syntax tree corresponding to parsing the expression

position := initial + rate * 60.

The direction of the arrows in the productions of a CFG implies a method for generating or deriving valid sentences, which is represented by a syntax tree. A parser is an acceptor for the language. While analysing the input string, the parser attempts to find a syntax tree. Therefore, in some sense, parsing reverses the derivation process in that we have an input string and have to "discover" the parse tree for it.

Parsing is one of the best understood branches of computer science. It has been so since the early 70's when Aho and Ullman [AU72], Knuth [Knu71] [Knu74], and many others put various parsing techniques solidly on their theoretical feet. Today, parsing is being used extensively in a number of disciplines: in computer science (for compiler construction, database interfaces, artificial intelligence), in linguistics (for text analysis, machine translation), in document preparation and conversion, to name a few. They are used by many other and still can be used in more disciplines.

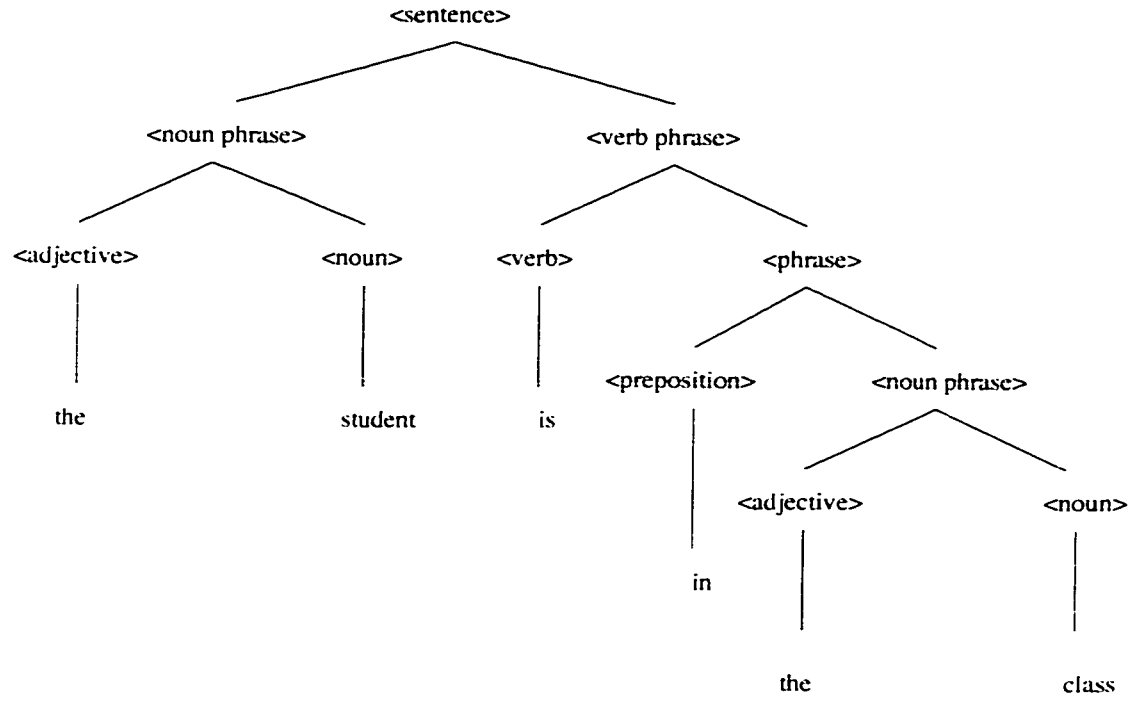


Figure 6: Tree structure for English sentence

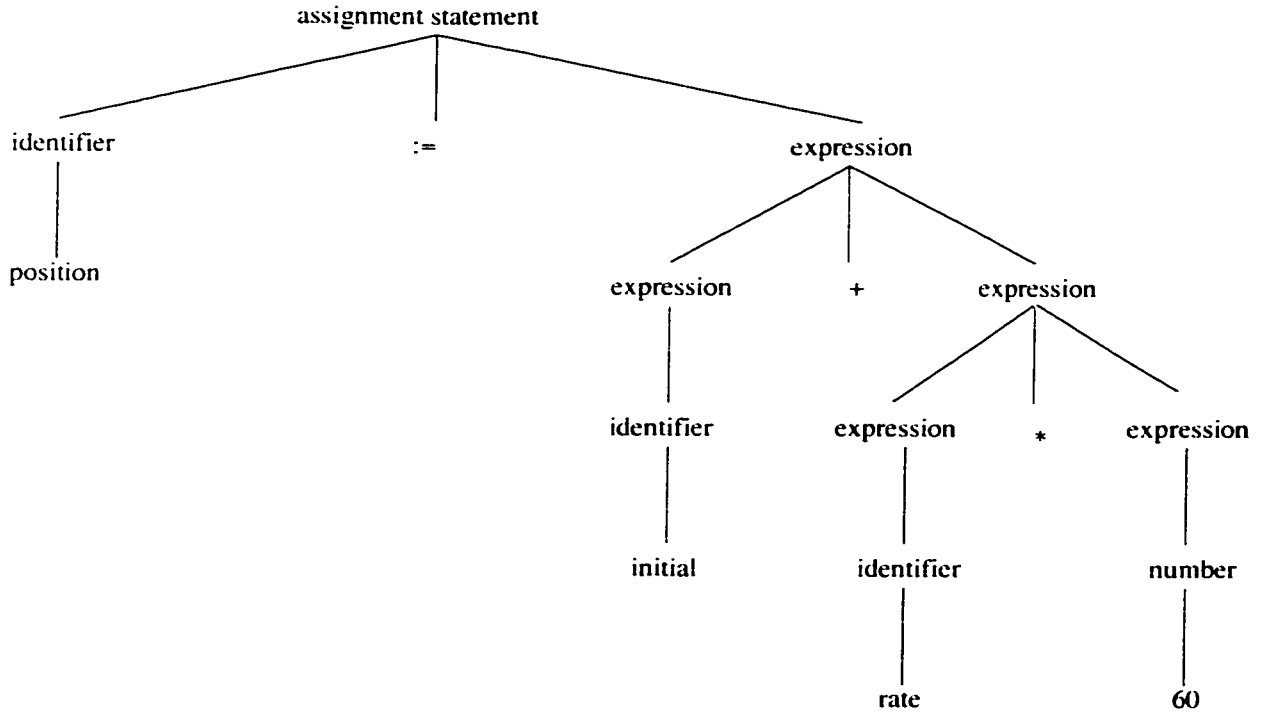


Figure 7: Parse tree for "position := initial + rate * 60"

This widespread use of parsing techniques is due to several reasons. One reason derives from the fact that parsing or the structuring process helps us to process the parsed object further. When we know that a certain segment of a sentence in German is the subject of the sentence, that information helps in translating the sentence.

A second is the completion of missing information that parsers can provide. Given a reasonable grammar of the language, an error-repairing parser can suggest possible words for missing or unknown words.

1.4 Parser Generators

A parser generator is a program which compiles a syntax definition, usually in the form of a grammar, into a parser for the language defined. The parser must accept any sentence of the language, and reject any other input string.

Typically, the output of a parser generator is source code for some programming language (such as C, C++, or Java), which can then be compiled and linked to the client application. Semantic actions defined within the grammar are used to control how the client application processes data elements as they are parsed, by binding specific symbols from the grammar to programmer-written application code. When a given symbol is parsed by the parser, application code corresponding to the semantic action for that symbol gets executed. Semantic actions are usually defined directly within the grammar definition and consist of compilable statements from the target programming language. In many cases, semantic actions are used to construct internal representations of the parsed data for subsequent use by the application.

1.5 What Problems Are We Trying to Solve?

There are already a number of parser generators, such as lex/yacc and their gnu versions flex/bison. They are not written in an object-oriented way and they do not generate an OO parser. In addition, these parser generators have three kinds of problems :

1. Learning how to use these parser generators is not an easy task. Including semantic actions is complicated.

2. Difficulty understanding the parser generator itself and how it is generating the parser.

3. The generated parser is not easy to understand. Standard parser generators, such as Yacc, have awkward features and do not conform to the goals of software engineering such as ease of use, modularity, information hiding, etc.

Our parser generator reads a grammar that is LL(1) and constructs a C++ program from it. The generated parser is a top-down recursive-descent parser.

The idea of representing a node in an annotated parse tree with a record with different fields is an old concept. Our generated parser contains a class for each symbol of the grammar. Every node on the parse tree is an instantiation of the corresponding class.

The class provides a function for parsing the symbol and other functions for processing the symbol. Since in object-oriented programming the idea of using global variables is discouraged, all the data needed to perform the parsing were kept within the classes, hence semantic actions are specified as member functions inside the class. The body of these functions are left to the user of the parser generator to complete.

1.6 Thesis Outline

There are five additional chapters. Chapter 2 gives an overview about parsing. It introduces top-down parsing, bottom-up parsing, and attribute grammars. It also presents a comparison between bottom-up and top-down parsing. Chapter 3 explains the role of a parser generator. It explains how Yacc can be used. It also presents the work of other researchers on object-oriented parsers and parser generators. In chapter 4 we justify our selection for a recursive descent parser. We also present our approach to building an object-oriented parser. Chapter 5 contains the explanation about the design and implementation of the parser generator. Chapter 6 contains the conclusion and future work. The thesis also contains an appendix containing output samples from the parser generator.

Chapter 2

Parsing

2.1 Early Attempts to Automate Parsing

In 1962, Chomsky introduced a pushdown automaton (PDA) [Cho62] that serves as a recognition device for context-free languages. This was the earliest automation of the parsing process. The basic idea of a pushdown automaton is as follows. It consists of:

- An input tape: the tape is divided into squares which contain the symbols of the input string: the tape can move to the left, a square at a time; each symbol can be read only once.
- A storage tape: this potentially infinite tape is called the pushdown stack; only the top symbol of the stack can be read; symbols can be added to or erased from the top of the stack.
- A finite control and a finite set of instructions; the finite control can be in one of finitely many states.

Depending on the input symbol, the state of the finite control and the top symbol on the pushdown stack, the machine performs its actions. Figure 8 shows a pushdown automaton.

Instructions have the form

$$(a, p, b) \rightarrow (q, u)$$

with the interpretation that when in state p, if the input symbol is a and the top pushdown symbol is b, then the next state is q, the input tape moves one position to the left, and symbol b is replaced by the string u. The leftmost symbol of u will be on top of the stack. Notice that if u is the empty word, then symbol b is in fact removed from the pushdown stack. Also, the empty word does not cause the input tape to move.

We say that a string is accepted by a PDA if it causes the automaton to empty its pushdown stack when all the input has been read. On the other hand, a language is said to be accepted by a PDA if it consists only of strings that are accepted by the automaton.

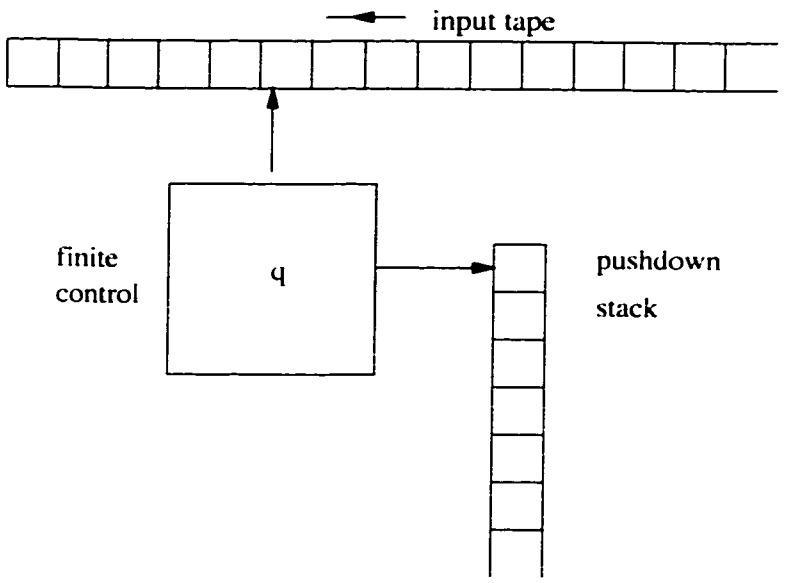


Figure 8: Pushdown automaton

PDAs define the same languages as context-free grammars. There exists methods to convert a context-free grammar to an equivalent PDA, and vice versa. In the early 1960, however, these methods were not clear. This is why PDAs are not popular. Moreover, for some languages, there are no deterministic PDAs that can be given for their grammars. Parsing methods that are based on a non deterministic PDA are too slow and thus not desirable.

Most parsing methods fall into one of two classes, called the top-down and bottom up methods. These terms refer to the order in which nodes in the parse tree are constructed.

The popularity of top-down parsers is due to the fact that efficient-parsers can be constructed more easily by hand using top-down methods. Bottom-up parsing, however, can handle a larger class of grammars, so software tools for generating parsers directly from grammars have tended to use bottom-up methods.

2.2 Top-down Parsers

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in pre-order.

Top-down parsers are often referred to as predictive parsers since at any stage, they try to predict the next lower level of the parse tree. This prediction is done by examining the next token in the input (the input is being scanned from left to right), and the current tree and then choosing the production to try next. Thus the tree is built from the top down trying to construct a leftmost derivation.

Figure 9 shows a step by step leftmost derivation of "$a * (a + a)$" from grammar G1 :

$$
\begin{aligned}
E &\rightarrow E + F \mid T \\
T &\rightarrow T \times F \mid F \\
F &\rightarrow (E) \mid a
\end{aligned}
$$

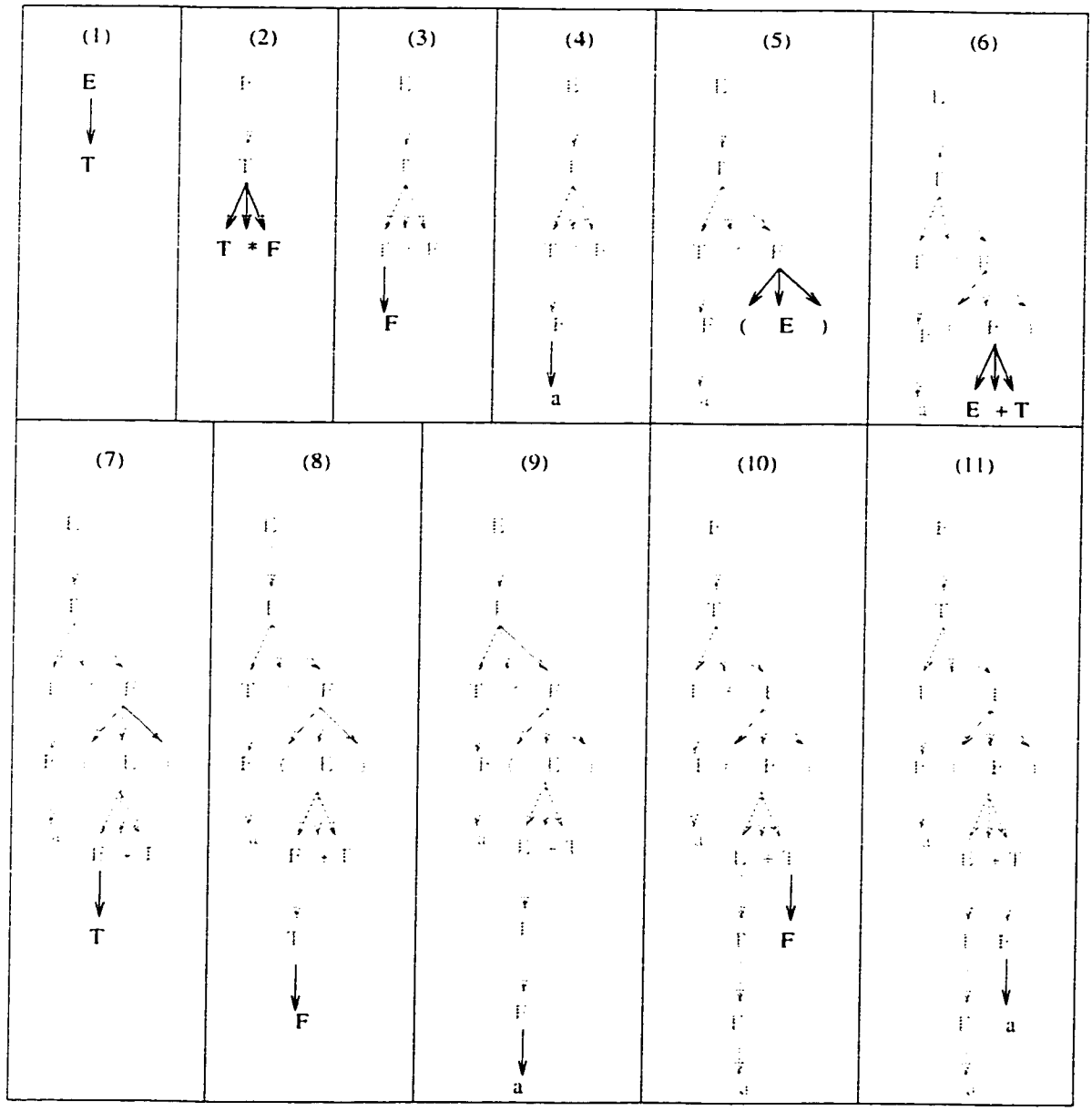


Figure 9: A step by step leftmost derivation of "$a * (a + a)$"

The algorithm below gives a systematic approach for generating leftmost derivations:

1. Begin with the start symbol for the grammar as the root for the parse tree.

2. As long as it is possible, replace the leftmost nonterminal V by u in the current sentential form $x\ V\ y$, where there is a production $V \rightarrow u$, so that we have $x\ u\ y$

In general the selection of a production for a nonterminal may involve trial and error, that is to try another production if the first is found to be unsuitable. A production is unsuitable, if after using the production, we can not complete the tree to match the input string.

2.2.1 Problems of Top-down Parsing

Top-down parsers run mainly into two major problems: left-recursion and backtracking.

Left Recursion

Consider the grammar we used earlier, and suppose we were parsing the expression $a * a * a$. The parse tree for this expression is shown in fig 10.

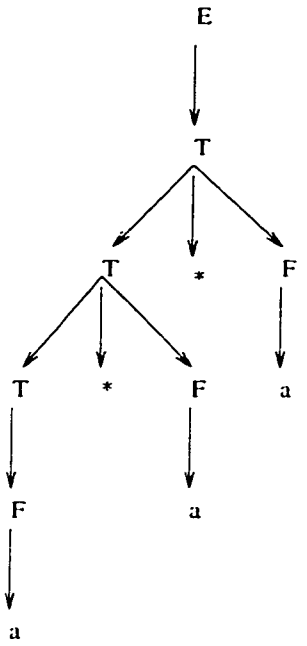


Figure 10: Parse Tree for "$a * a * a$"

It is possible for the top-down parser of this grammar to keep trying to grow the parse tree for ever. The problem arises with left-recursive productions like $T \rightarrow T * F$ in which the leftmost symbol on the right side is the same as the nonterminal on the left side of the production.

Let us suppose the top-down parser for this grammar started expanding the tree exactly as already shown in figure 10, i.e., with productions : $E \rightarrow T$, $T \rightarrow T * F$, and $T \rightarrow T * F$. At this moment, instead of choosing $T \rightarrow F$, the parser tries $T \rightarrow T * F$. The parser has no way to know that this is not the right production. A parser bases its decisions on the grammar and the input token string, which has not changed. The input token string, changes only when a terminal in the right side is matched. Since the production begins with the nonterminal T, no changes to the input take place, thus causing the parser to try growing the tree for ever as shown in figure 11.

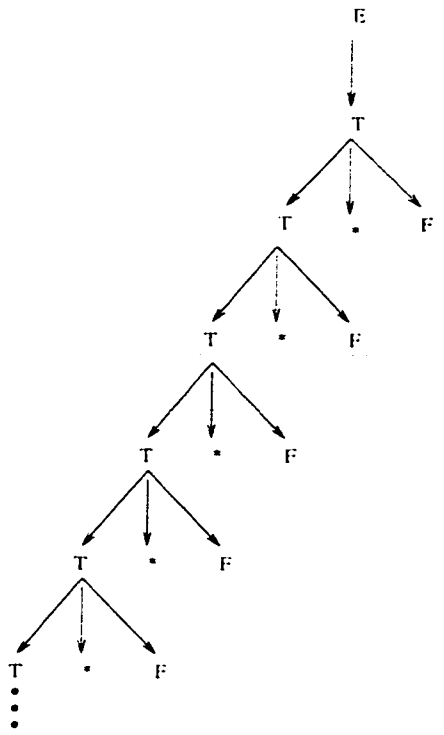


Figure 11: Effect of left-recursive grammars on top-down parsing

In a production of the form $A \rightarrow A\alpha$ left recursion is obvious and therefore called immediate left recursion. There is another form of left recursion, called non-immediate left recursion, that is not as immediately apparent but is equally dangerous. In the latter, productions have the form:

$$A \rightarrow B\ \alpha\ |\ \cdots$$
$$B \rightarrow A\ \beta\ |\ \cdots$$

The problem occurs when A uses $B\alpha$ and B uses $A\beta$ and so forth.

The only solution to both immediate and non-immediate left recursion is to rewrite the grammar in such a way as to eliminate the left recursions. There are systematic algorithms to do this [ASU86].

Backtracking

One way to perform a top-down parse is to begin with the start symbol, examine the incoming input token, and select an applicable production. An applicable production is one that is not ruled out by the fact that it starts with a terminal other than the incoming input token. If the selected production fails to produce a correct tree, the parser has to backtrack and try another applicable production. The parser will keep doing this until a correct parse tree is constructed or all productions are exhausted. For example, consider grammar G2 and the input *cad* :

$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b\ |\ a$$

As shown in figure 12. to construct a parse tree for this string top-down, we create a tree consisting of a single node labelled S. Initially the input symbol is c. We then try to expand tree with the first production for S. Since this production starts with terminal c that matches the current input symbol, the current input symbol becomes a. Next we try to expand the leftmost nonterminal leaf in the tree. A is expanded using the first alternate for A. We now have a match for the second input symbol so we advance the input symbol to d. At this point, the parser generated a complete tree, there are no nonterminals left to expand. But this is a wrong tree. To fix this problem, the parser must go back or backtrack to A and look for another alternative of A. The current input symbol must be returned back to where it was before choosing the erroneous production. The second alternative of A will produce the correct parse tree.

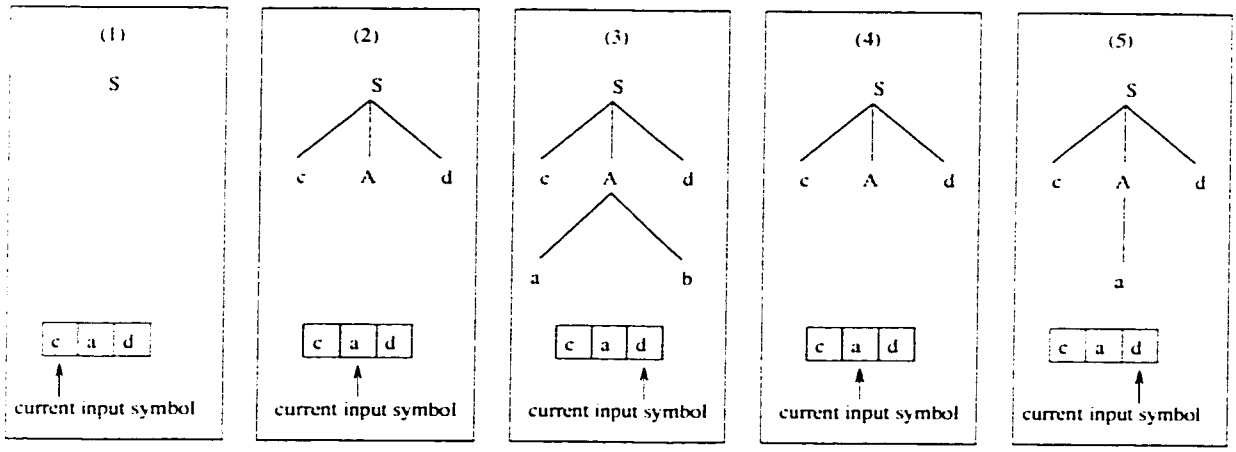


Figure 12: Backtracking in top-down parsing

These problems can all be handled, and there are top-down parsers that take advantage of backtracking. However, backtracking parsers are rarely seen because they are slow in general and particularly slow when the input contains an error. The backtracking parser will have to try all productions before discovering that the program contains an error.

Another problem of backtracking is related to semantics. If we make a sequence of erroneous expansions and subsequently discover a mismatch, we may have to undo the semantic effects of making these erroneous expansions. For example, entries made in the symbol table might have to be removed. Since undoing semantic actions requires a substantial overhead, it is reasonable to consider top-down parsers that avoid backtracking.

Therefore backtracking is not an attractive approach to parsing and is often avoided. In many situations, the grammar can be modified to get rid of backtracking. For example, the problem in our grammar is that two of the productions of A start with a and hence it is impossible to predict which production to choose upon finding an a in the input. The solve this problem we have to eliminate this ambiguity. This can be done only by introducing some changes on the grammar to become grammar G3 :

$$
\begin{aligned}
S &\rightarrow c\ A\ d \\
T &\rightarrow b \mid \epsilon \\
A &\rightarrow a\ T
\end{aligned}
$$

This grammar defines the same language as the earlier, but it allows the parser to generate the tree without backtracking. We have factored out the common prefix a and used another nonterminal to permit the matching of b when it is in the input. This transformation is known as left factoring, and there are systematic algorithms to do it [ASU86].

2.3 LL(1) Parser

2.3.1 Predictive Parsing

As we have already mentioned, top-down parsers are often referred to as predictive parsers since at any stage, they try to predict the next lower level of the parse tree. Predictive parsing is a special form of top-down parsing in which no backtracking is required. To construct a predictive parser, we must know, given the current input symbol a and the nonterminal A to be expanded, which one of the alternatives of A, $\alpha 1, \alpha 2, \alpha 3, \ldots$ is the unique alternative that derives a string beginning with a.

Since left-recursion causes top-down parsers to loop forever, in a predictive parser grammars can not be left recursive. Also, in a predictive parser the grammar has to be left factored, therefore if $S \rightarrow \alpha \mid \beta$, where α and β is some sentential form, then the set of terminals that α can start with has to be disjoint with the set of terminals that β can start with.

Let $FIRST(\alpha)$ be the set of terminals that begin the strings derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in $FIRST(\alpha)$. The following algorithm shows how to calculate $FIRST(\alpha)$

1. If α is a terminal, then $FIRST(\alpha)$ is $\{\alpha\}$.
2. If α is ϵ, then $FIRST(\alpha) = \{\epsilon\}$.
3. If α is a nonterminal and $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$ then $FIRST(\alpha) = \cup_k FIRST(\beta_k)$
4. If $\alpha = X_1 \ldots X_n$ where $X_1, \ldots, X_n$ are all symbols, and $n >= 2$
   then if $X_1 \stackrel{*}{\Rightarrow} \epsilon$
   then $FIRST(\alpha) = (FIRST(X_1) - \{\epsilon\}) \cup FIRST(\beta)$, where $\beta = X_2 X_3 \ldots X_n$
   else $FIRST(\alpha) = FIRST(X_1)$

If the grammar does not contain any ϵ productions, the parsing process would be perfectly predictive. By comparing the forthcoming input token with the $FIRST$ set of each of the alternate of the nonterminal we are trying to expand, the parser can decide which alternate to choose. Obviously, if this token does not belong to any of the $FIRST$ sets, the parser knows there is an error in the input. However when the grammar contains some ϵ productions, the selection of productions might still involve some guessing even with $FIRST$ sets being used.

When grammars have ϵ productions, the parser, trying to expand the nullable nonterminal A, can not tell for sure upon finding that the incoming token is not in $FIRST(A)$ whether to choose

$A \rightarrow \epsilon$ or whether the input is erroneous. To solve this problem, the parser must be able to know the tokens needed to expand what follows A. When the parser has this information, it can decide whether the input token is erroneous or whether $A \rightarrow \epsilon$ is the needed production. We call this information $FOLLOW(A)$.

We define $FOLLOW(A)$, for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is the set of terminals a such that there exists a derivation of the form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ where S is the start symbol, α and β are some sentential forms. Note that there may, at some time during the derivation, have been symbols between A and a, but if so, they derived ϵ, and disappeared.

The following algorithm computes $FOLLOW(A)$

1. if there is a production $A \rightarrow \alpha B \beta$ where B is either a nonterminal or a regular expression
   if $(FIRST(\beta) - \{\epsilon\}) \not\subset FOLLOW(B)$
   $FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) - \{\epsilon\})$

2. if there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where $\beta \stackrel{*}{\Rightarrow} \epsilon$
   if $FOLLOW(A) \not\subset FOLLOW(B)$
   $FOLLOW(B) = FOLLOW(A) \cup FOLLOW(B)$

3. repeat steps 1 and 2 until nothing can be added to any follow set.

2.3.2 Predictive Recursive Descent Parsing

A top-down model syntax analysis in which we execute a set of recursive functions to process the input. A function is associated with each nonterminal of a grammar. The sequence of functions called in processing the input implicitly defines a parse tree for the input. Consider the following grammar, G4:

$$
\begin{array}{lcllll}
E & \rightarrow & T & E2 & & \\
E2 & \rightarrow & + & T & E2 & | \quad \epsilon \\
T & \rightarrow & F & T2 & & \\
T2 & \rightarrow & \times & F & T2 & | \quad \epsilon \\
F & \rightarrow & (& E &) & | \quad id
\end{array}
$$

The corresponding functions for each nonterminal in pseudo-code are shown in figure 13. NextSymbol in figure 13 is the token produced by the lexical analyser. The PRINT statements will print out the reverse of a left derivation.

2.3.3 Table-Driven Predictive Parsing

In predictive parsing, having the current input symbol and the nonterminal to expand is enough to select the production to use next in the derivation. For example, consider the grammar above. One can say that if E is the nonterminal to expand, and the current terminal on the input is id then production $E \rightarrow TE2$ must be used. One can also write down all the possible combinations of

```
PROCEDURE E
BEGIN {E}
   T ; E2
   PRINT(''E found.'')
END {E}
```

```
PROCEDURE T
BEGIN {T}
   F ; T2
   PRINT(''T found.'')
END {T}
```

```
PROCEDURE E2
BEGIN {E2}
  IF next_symbol IS '+' THEN
  BEGIN {IF}
      MATCH('+') ; T ; E2
  END {IF}
  PRINT(''E2 found'')
END {E2}
```

```
PROCEDURE T2
BEGIN {T2}
   IF next_symbol IS '*' THEN
   BEGIN {IF}
      MATCH('*') ; F ; T2
   END {IF}
   PRINT(''T2 found'')
END {T2}
```

```
PROCEDURE MATCH(t : token)
BEGIN {MATCH}
   IF next_symbol IS t THEN
   BEGIN {IF}
      PRINT('' t found'')
      GET_NEXT_SYMBOL
   END {IF}
   ELSE
      ERROR('' t was expected'')
END {MATCH}
```

```
PROCEDURE F
BEGIN {F}
   IF next_symbol IS '(' THEN
   BEGIN {IF}
      MATCH('(') ; E ; MATCH(')')
   END {IF}
   ELSE
   IF next_symbol IS id THEN
      MATCH(id)
   ELSE
      ERROR(''F failed'')
   PRINT(''F found'')
END {F}
```

Figure 13: Pseudo-code for a predictive recursive-descent parser.

(A, a, P) where A is a nonterminal to expand, a is the terminal on the input, and P the production that must be chosen, in the form a table as in figure 14.

Non Terminals	Input Symbols					
	id	+	*	(	)	S
E	E → T E2			E → T E2		
E2		E2 → + T E2			E2 → ϵ	E2 → ϵ
T	T → F T2			T → F T2		
T2		T2 → ϵ	T2 → * F T2		T2 → ϵ	T2 → ϵ
F	F → id			F → (E)		

Figure 14: Predictive parsing table.

Naturally, an empty entry in this table represents the impossibility of finding a production to expand the left-most non terminal with the current input symbol.

The following algorithm can be used to generate such a table :

1. For every production $A \rightarrow \alpha$ in the grammar, do steps 2 and 3.
2. If α can derive a string starting with a then $TABLE[A, a] = A \rightarrow \alpha$
3. If α can derive the empty string, ϵ, then for all b that can follow a string derived from A : $TABLE[A, b] = A \rightarrow \alpha$
4. Set empty entries in the table to error.

In recursive descent parsing, the algorithm relied on the implicit stack defined by the language. For example, consider "Procedure E" in figure 13. In this procedure, "T" is called. "T" in turn calls "F" and "T2" then "T" returns control. The implicit stack plays the role of a memory that tells us where did we leave last. "E2" must be executed next.

Table-driven predictive parsing is "table driven" recursive descent parsing. Instead of recursive procedure calls, a table is consulted for the next action to perform and an explicit stack is used. Figure 15 shows us the model of a recursive predictive parser.

The following algorithm shows us how a parsing table and a stack can be used to perform non recursive predictive parsing.

1. Initially, the stack contains the start symbol S on its top.
2. If the stack top contains a terminal symbol a, then the input symbol must be an a, else ERROR. If the two match, then advance the read head on the input string and pop the terminal symbol from the stack.
3. If the stack top contains a nonterminal symbol A, then examine the input symbol currently under the read head, and consult the table to determine which production is to be applied. If

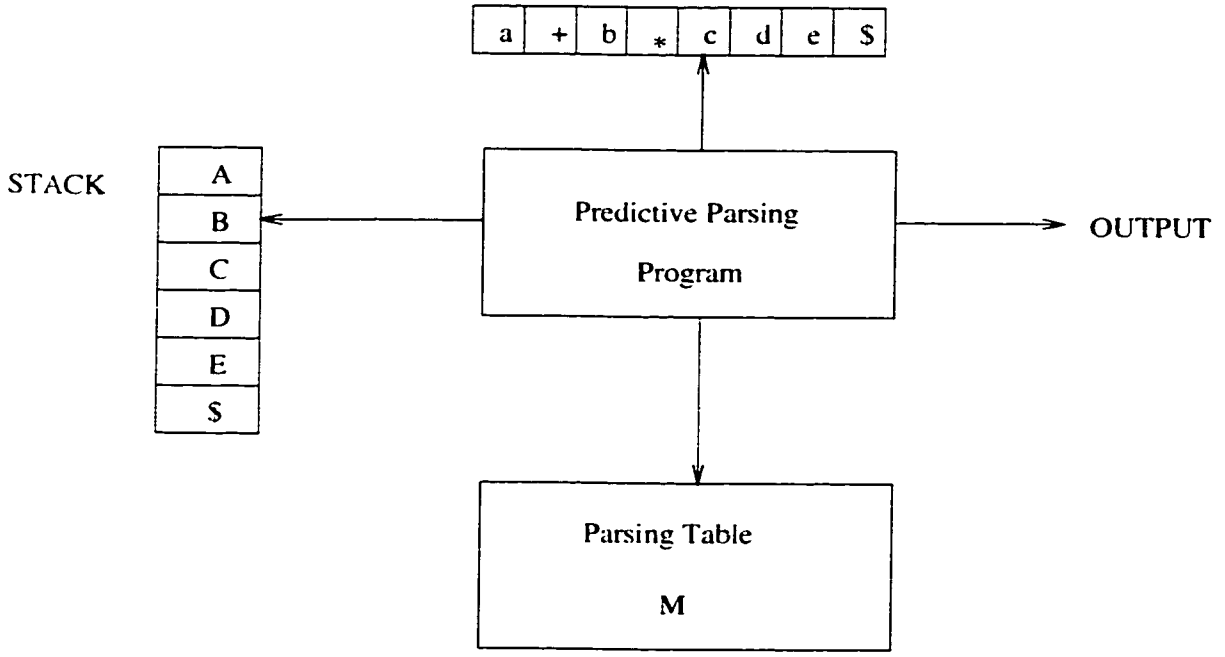


Figure 15: Nonrecursive predictive parser.

the table indicates production $A \rightarrow w$, then remove A from the stack and push the string w onto the stack.

4. Stop when the stack is empty. If stack is empty and all the input is consumed then parsing was successful.

Figure 16 shows us all the moves made by the non recursive predictive parser of grammar G4 on input $id * id + id$.

2.3.4 Characteristics of LL(1) Grammars

Even with the elimination of left-recursion, left factoring, and using $FIRST$ and $FOLLOW$ sets, there are still cases when examining the next token is not sufficient to decide which production to use. For example, when we are trying to expand a nonterminal A, if the current input token is a, $A \stackrel{*}{\Rightarrow} \epsilon$, and $a \in FIRST(A) \cap FOLLOW(A)$, then we can not tell for sure by just having a on the input what is the correct derivation, $A \stackrel{*}{\Rightarrow} \epsilon$ or the other that derives a string beginning with a.

For example consider grammar G5 below:

$$\begin{aligned} A &\rightarrow B \ \ id \\ B &\rightarrow id \ + \ B \ \mid \ \epsilon \end{aligned}$$

This grammar is neither left recursive nor does it need left factoring. The predictive parsing table for this grammar is shown in figure 17.

In this table we notice that, $TABLE[B, id]$ has two entries. There are two potential productions with which to expand B when an id is encountered on the input. We call this a conflict.

STACK	INPUT	PRODUCTION
E	id * id + id	E → T E2
E2 T	id * id + id	T → F T2
E2 T2 F	id * id + id	F → id
E2 T2 id	id * id + id	
E2 T2	* id + id	T2 → * F T2
E2 T2 F *	* id + id	
E2 T2 F	id + id	F → id
E2 T2 id	id + id	
E2 T2	+ id	T2 → ∈
E2	+ id	E2 → + T E2
E2 T +	+ id	
E2 T	id	T → F T2
E2 T2 F	id	F → id
E2 T2 id	id	
E2 T2		T2 → ∈
E2		E2 → ∈

Figure 16: Moves made by predictive parser on input "$id * id + id$".

Non Terminals	Input Symbols	
	id	+
A	A → B id	
B	B → id + B B → ∈	

Figure 17: Predictive parsing table that have two entries in one cell.

A grammar whose parsing table has no multiply-defined entries is said to be LL(1). A LL(1) grammar is one in which we need only to examine the next token to decide which production to use. The first "L" in LL(1) stands for scanning the input left to right and the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing actions decisions.

No ambiguous or left-recursive grammars can be LL(1). It can also be shown that a grammar is LL(1) if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G the following conditions hold:

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- if $\beta \stackrel{*}{\Rightarrow} \epsilon$, then $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$

2.4 LL(k)

Up until now, we have limited the look-ahead to just one symbol, and one might wonder if having a look-ahead of k-symbols instead of one makes the method more powerful. It does, so let us define LL(k) grammars.

G is LL(k) if given a string $wA\alpha$ in $(N \cup T)^*$ and the first k terminal symbols (if they exist) to be derived from $A\alpha$ there is at most one production which can be applied to A to yield a derivation of any terminal string beginning with w followed by those k terminals.

We say a grammar is LL if it is LL(k) for any $k \geq 1$.

More formally, a CFG $G = (N, T, P, S)$ is LL(k), for some fixed integer k, if whenever there are two leftmost derivations

$$\left.\begin{array}{llllll} S & \stackrel{*}{\Rightarrow} & wA\alpha & \Rightarrow & w\beta\alpha & \stackrel{*}{\Rightarrow} & wx \\ \text{and} & & & & & & \\ S & \stackrel{*}{\Rightarrow} & wA\alpha & \Rightarrow & w\gamma\alpha & \stackrel{*}{\Rightarrow} & wy \end{array}\right\} \begin{array}{ll} \text{such that} & FIRST_k(x) = FIRST_k(y), \\ \text{it follows that} & \beta = \gamma. \end{array}$$

We define

$$FIRST_k(\alpha) = \{w \in T^* \mid \text{ either } |w| < k \text{ and } \alpha \stackrel{*}{\Rightarrow} w, \text{ or } |w| = k \text{ and } \alpha \stackrel{*}{\Rightarrow} wx \text{ for some } x\}$$

To obtain a full LL(k) parser, the method that we used to obtain a full LL(1) table driven parser can be extended to deal with pairs [A , L] where L is a $FIRST_k(\alpha)$ in a production $A \to \alpha$. This extension is straightforward and will not be discussed further.

LL(k) parsers with $k > 1$ are seldom used in practice, because the parse tables are huge, and there are not many languages that are LL(k) for some $k > 1$ but not $LL(1)$. Even the languages that are LL(k) for some $k > 1$ but not $LL(1)$, are usually for the most part LL(1), and can be parsed with conflict resolvers at places where the grammar is not LL(1).

2.5 Bottom-up Parsers

Bottom-up parsing creates the reverse of a right derivation. Bottom up analysis proceeds from leaf nodes and upwards towards the root construct.

The algorithm below gives a systematic approach to constructing a right-most derivation :

1. Begin with the string (i.e., leaves of the to-be created parse tree).
2. Try to reduce to the start symbol by finding the current *handle*: The handle is
   (a) the largest collection of terminals and non terminals in the leftmost part of the input which can be found on the right-hand side of some production and
   (b) such that all the symbols to the right of the handle are terminals and
   (c) such that replacing the handle with the left-hand side of the production eventually (by finding more handles) leads back to start symbol.

Figure 18 shows a step by step right-most derivation of $a * (a + a)$ from grammar G6 below :

$$\begin{array}{lcl} E & \rightarrow & T \mid E + T \\ T & \rightarrow & F \mid T * F \\ F & \rightarrow & id \mid (E) \end{array}$$

2.6 LR Parsers

Shift-reduce parsing is a general style of bottom-up syntax analysis. Shift-reduce parsing attempts to construct a parse tree for an input beginning at the leaves and working up towards the root. We can think of this process as one of reducing a string to the start symbol of the grammar. At each reduction step a particular substring matching the right hand side of a production is replaced by the left hand side of that production, and if the substring is chosen correctly at each step, the reverse of a rightmost derivation is obtained.

Operator-precedence parsing is one form of shift-reduce parsing, that works for a small but important class of grammars. Operator-precedence parsers can be constructed by hand, however a lot of restrictions must be imposed on the grammar which makes it less attractive. For example, the grammar can not have ϵ-productions, also it can not have two adjacent non terminals. Even though operator-precedence parsers have been written for entire languages, we are not going to discuss them in this thesis because they deal only with a small class of grammars.

The shift-reduce method to be described here is called LR(k) parsing. LR(k) parsing, introduced by Knuth in 1965 [Knu65], can be used to parse a large class of context-free grammars. There are a number of variants for LR parsing, such as SLR(1), LR(1), and LALR(1), but they all use the same driver. They differ only in the generated table. The L in LR(k) indicates that the string is parsed from left to right; the R indicates that the reverse of a right derivation is produced, and the k for the number of input symbols of lookahead that are used in making parsing decisions. When k is 1, k is usually omitted, so LR means LR(1). Use of more than one symbol token of lookahead puts

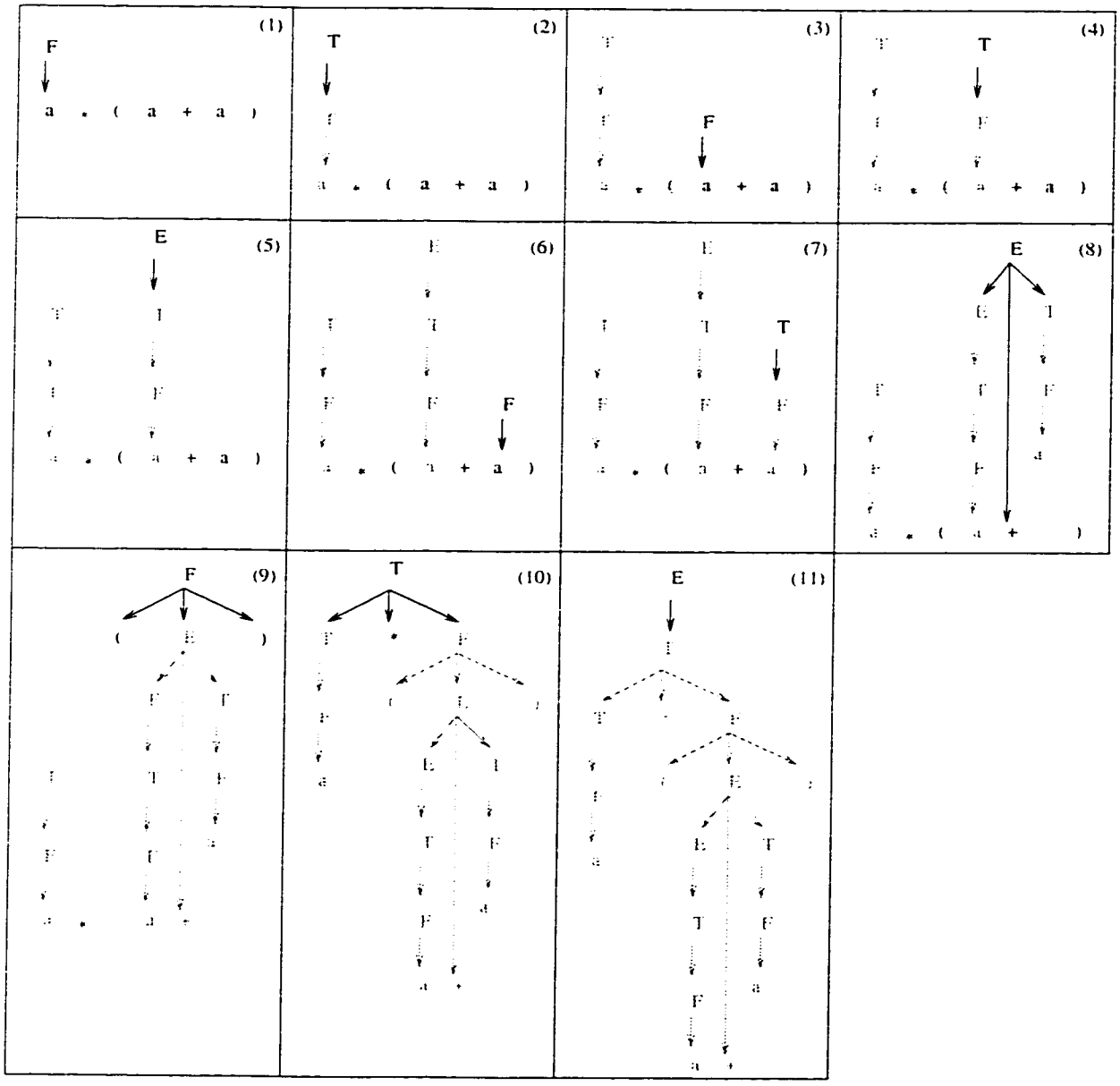


Figure 18: Step by step right-most derivation of "$a * (a + a)$"

an undesirable burden on the lexical analyser and results in very large parsing tables. Hopcroft and Ullman [HU79] showed that any deterministic context free language can be handled by an LR(1) parser. Therefore, in this chapter we will be only describe LR(1) parsing.

Shift-reduce parsing may be implemented using a standard pushdown automaton as follows:

- Use the initially empty pushdown stack to hold recognised constructs and symbols.
- Parsing proceeds by either accepting a legitimate continuation of input, or recognising a valid sentence which has been recognized on top of stack:
  - If the top elements of the stack matches the right hand side of some production, they are reduced by replacement with the non terminal on the left-hand side of the production.
  - If the current input token is a valid continuation of the input, it is accepted and shifted onto the stack.
- The input is accepted when input is exhausted and the stack is empty.

Figure 19 shows all the shift-reduce actions taken by the LR Parser upon parsing $a * (a + a)$, but it does not give us any hint about how decisions between shift and reduce were taken. It is not intuitive to know when and how to recognize stack items to be reduced, or when a construct may be extended by stacking the current input symbol. Additional information is needed to guide the decisions to be taken between a shifting and reducing. This information is provided in a parsing table.

Figure 20 shows the model of a LR parser, and how the stack and the parsing table fit into this parsing technique.

As shown in figure 20, the parsing table consists of two parts, the action part and the goto part. Figure 21 shows a parsing table for grammar G6.

We notice that the table has a column "States". This is because as we have said earlier, our shift-reduce parsing method is implemented via a push-down automaton. Each state summarizes the information contained in the stack below it. We have also seen in section 2.1 that a PDA has a finite-state controller. The parser table is the state table of the controller.

The action part has four different kinds of entries called actions:

- Shift: indicated by the "S#" entries in the table where # is a new state.
- Reduce: indicated by "R#" where # is the number of a production. Productions were numbered to save space when entered into the table.
- Accept: Accept is indicated by the "Accept" entry in the table. When we come to this entry in the table, we accept the input string. Parsing is complete.
- Error: The blank entries in the table indicate a syntax error.

The goto table provides information for state transition. The goto table and the states of the pushdown automaton defines a deterministic finite automaton that recognizes the viable prefixes of

STACK	INPUT	ACTION
	id * (id + id)	Shift
id	* (id + id)	Reduce by F → id
F	* (id + id)	Reduce by T → F
T	* (id + id)	Shift
T *	(id + id)	Shift
T * (	id + id)	Shift
T * (id	+ id)	Reduce by F → id
T * (F	+ id)	Reduce by T → F
T * (T	+ id)	Reduce by E → T
T * (E	+ id)	Shift
T * (E +	id)	Shift
T * (E + id	)	Reduce by F → id
T * (E + F	)	Reduce by T → F
T * (E + T	)	Reduce by E → E + T
T * (E	)	Shift
T * (E)		Reduce by F → (E)
T * F		Reduce by T → T * F
T		Reduce by E → T
E		Accept

Figure 19: Actions taken by an LR parser on "$id * (id + id)$" .

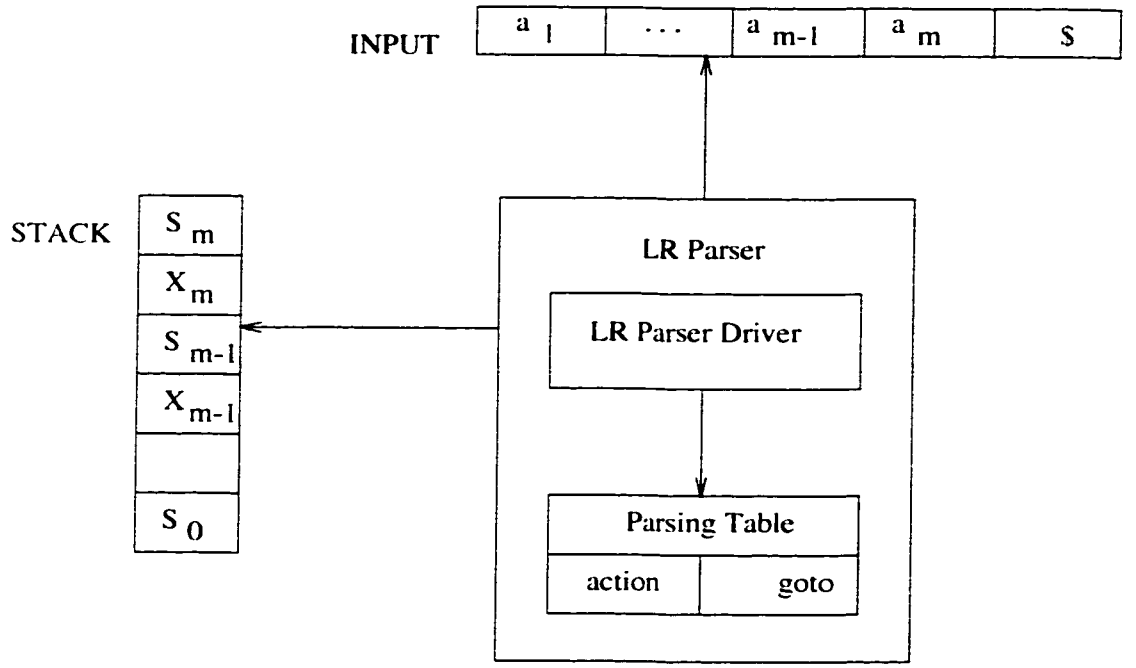


Figure 20: Model of a LR parser.

STATE	action						goto		
	id	+	*	(	)	$	E	T	F
0	S 5			S 4			1	2	3
1		S 6				Accept			
2		R 2	S 7		R 2	R 2			
3		R 4	R 4		R 4	R 4			
4	S 5			S 4			8	2	3
5		R 6	R 6		R 6	R 6			
6	S 5			S 4				9	3
7	S 5			S 4					
8		S 6		S 11					10
9		R 1	S 7		R 1	R 1			
10		R 3	R 3		R 3	R 3			
11		R 5	R 5		R 5	R 5			

Figure 21: LR parsing table.

the grammar. As in a finite state machine, it is consulted as to how recognition may further proceed. The goto table provides state transition information to model how much of a production has been recognized and how to proceed based on remaining input tokens.

2.6.1 LR Parser Driver

The driver reads the input and consults the table. The table has two parts: an action part and a goto part. The driver algorithm below, shows us how these actions and the goto table are used :

1. Push State 0 into the initially empty Stack.
2. Append $ to end of the input.
3. Repeat step 4 until input is accepted or an error is found.
4. Examine action table entry action[s_m, a_i] , where s_m is the current state (on top of stack) and a_i is the incoming token. Of the following four cases, choose one that corresponds to the action found:
   - action is Shift (Sn) :
     (a) Push the current input symbol a_i into the stack.
     (b) Push the new state (n) into the stack.
     (c) Advance input pointer to point to next input symbol
   - action is Reduce (Rn):
     (a) Pop $2 \times w$ elements from the stack, where w is the number of symbols on the right-hand side of production number n.
     (b) Let s' be the state on top of the stack now.
     (c) Push A on top of the stack where A is the non terminal on the left-hand side of production number n.
     (d) Let s'' be the state in table entry goto[s' , A]. Push s'' on top of the stack.
   - action Accept: Stop, parse is successful.
   - action Error: Stop, declare an error.

2.6.2 Simple LR(1) Parse-Table Construction

Constructing the States

We have said earlier that a LR parse table is the state table of the controller of the pushdown automaton. In order to build this table, we have to build the states of the PDA. A state indicates a part of a production which has been recognized; it also indicates what is left to be recognized.

The information about how much of a production has been recognized and what is left may be represented by an augmented production rule with a dot symbol on the right-hand side of the rule.

For example, the dot in this augmented production $A \rightarrow a \cdot Sc$ simply means that of the production's right-hand side a has been recognized and we expect to see a portion of the input string which can be derived from Sc. The dot symbol proceeds to the right as symbols are recognized. Consider the following production:

$$S \rightarrow aAc$$

augment the production with a dot and follow the motion of the dot. We notice that four different configurations must be encountered, as shown in Figure 22, the rule is said to be recognised and reduced.

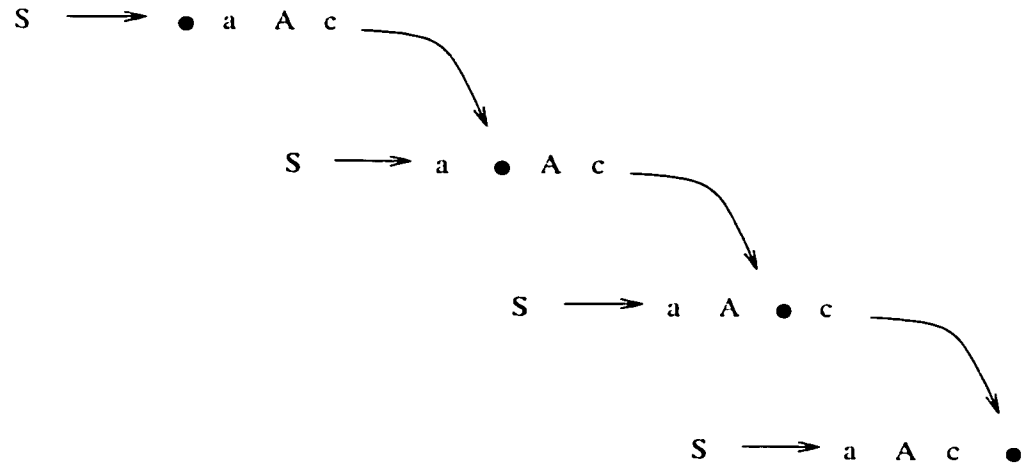


Figure 22: Configurations obtained during parsing due to the dot motion

When at the extreme left of the production, the dot symbol indicates that nothing of the rule has been recognized so far. Similarly, when at the extreme right of the production, it indicates that the complete rule has been recognized, and thus a reduction is possible.

Such an augmented production is called an item. Since items and states serve the same purpose, items can be viewed as the states of the pushdown automata. A state has one or many items. Items in a state are grouped together based on a relationship called the *closure*. The idea of a closure set can be explained by looking at the three productions below :

$$\begin{aligned} S &\rightarrow a \cdot A\ c \\ A &\rightarrow G\ x \\ G &\rightarrow z \end{aligned}$$

In the first production, the dot precedes A which means an A is anticipated next. However, if A is anticipated then a G must also be anticipated since the latter is a constituent of the former. Similarly it must anticipate a z symbol because it is an indirect constituent. This information can be summarized by this set of items :

$$\{S \rightarrow a \cdot Ac\ ,\ A \rightarrow \cdot Gx\ ,\ G \rightarrow \cdot z\}$$

This set of items is called the closure of the set containing item $S \rightarrow a \cdot Ac$.

The following algorithm shows how *closure(I)* can be constructed, where I is a set of items:

1. Initially, every item in I is added to *closure(I)*.

2. If $A \rightarrow \alpha \cdot B\beta$ is in *closure(I)* and $B \rightarrow \lambda$ is a production, then add the item $B \rightarrow \cdot\lambda$ to *closure(I)*, if it is not already there. Apply this rule until no more new items can be added to *closure(I)*.

Items are grouped together and each group becomes a state which represents a condition of the parse. The following algorithm shows how states are built:

1. Create a new nonterminal S' and a new production $S' \rightarrow S$ where S is the start symbol.

2. If S is the start symbol, put $S' \rightarrow \cdot S$ into a start state called state 0.

3. For every item in a state, add the closure of the set containing that item to the state.

4. Look for an item of the form $A \rightarrow x \cdot z\omega$ where z is a single terminal or nonterminal and build a new state from $A \rightarrow xz \cdot \omega$. Include in the new state all items with $\cdot z$ in the original state.

5. Continue until no new states can be created. A state is new if it is not identical to an old state.

The states built by this algorithm are called the LR(0) sets, or the canonical LR(0) sets.

Filling the Action and Goto Part

Consider the following grammar G7 :

$$\begin{array}{lcl} S & \rightarrow & a \;\; A \;\; c \;\; | \;\; b \\ A & \rightarrow & a \;\; S \;\; c \;\; | \;\; b \end{array}$$

The LR(0) states for this grammar, using the algorithm above, are shown in Figure 23.

We have said that the states, and the goto table define a DFA that recognizes the viable prefixes of the grammar. Having all the states of this DFA, all we have to do is to complete it, is to draw the transitions between states. By studying figure 23, we notice that there can be a transition from S_0 to S_1 labelled S. Also, there can be a transition from S_0 to S_1 labelled b, and so on. The complete DFA is shown in figure 24.

From figure 24 we can collect all the information to fill the parsing tables. All transitions from one state to another that are labelled with a nonterminal will fit into the goto table as shown in figure 25.

On the other hand, any transition from state m to state n, that is labelled with a terminal, a for example, will correspond to a "***shift*** n" or a "S n" entry in Action[m, a] . As for a state, x, with no transition going out from, if it contains an item with the dot at the right end of the right-hand side of the production, then there corresponds to it an entry "***reduce*** n", where n is the production number, or "R n" in Action[x,a] provided that terminal a is in FOLLOW(x). This is shown in figure 26.

The construction of the Simple LR(1) (also called SLR(1)) parsing table can be summarized by the following algorithm:

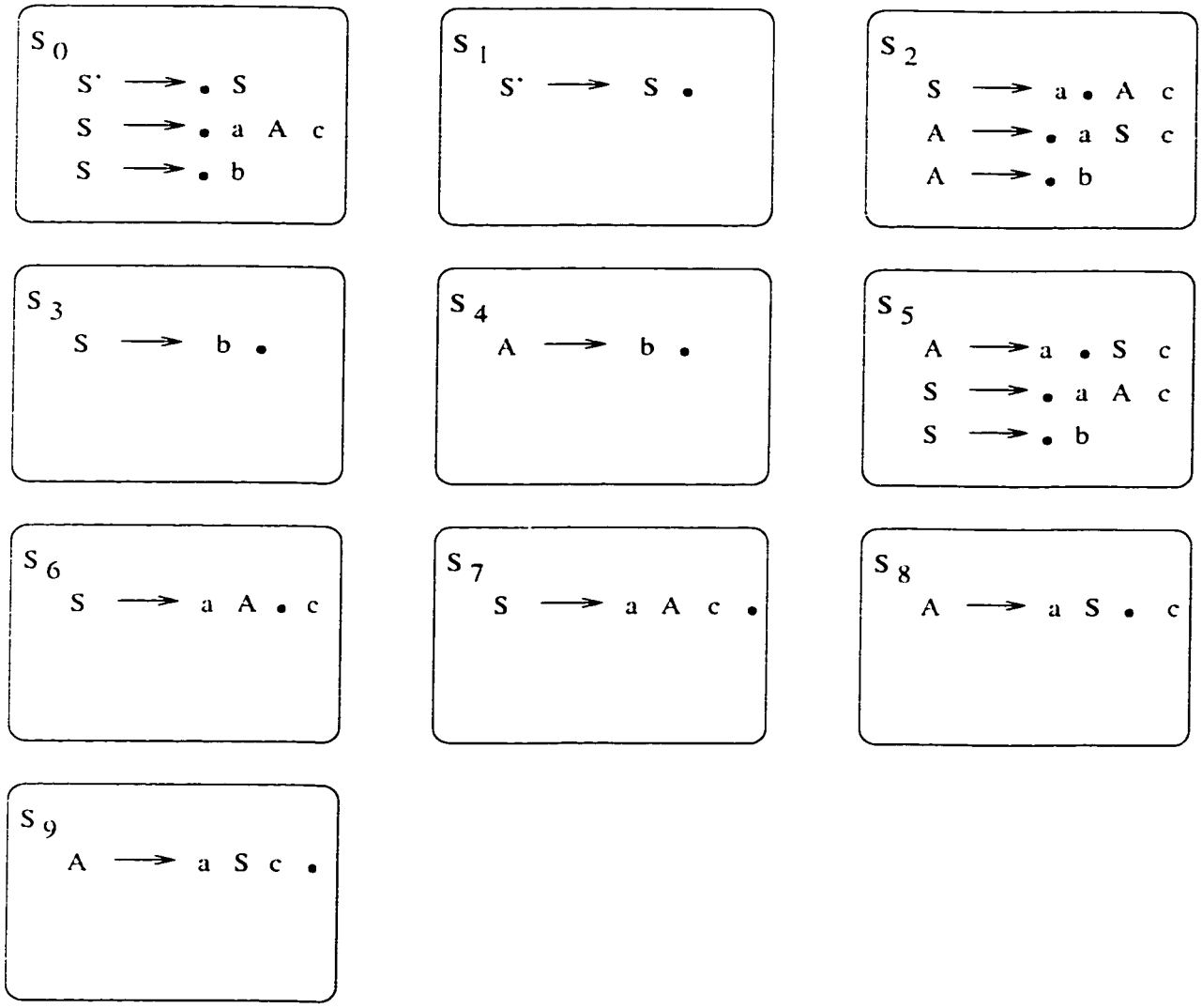


Figure 23: LR(0) states for grammar G7

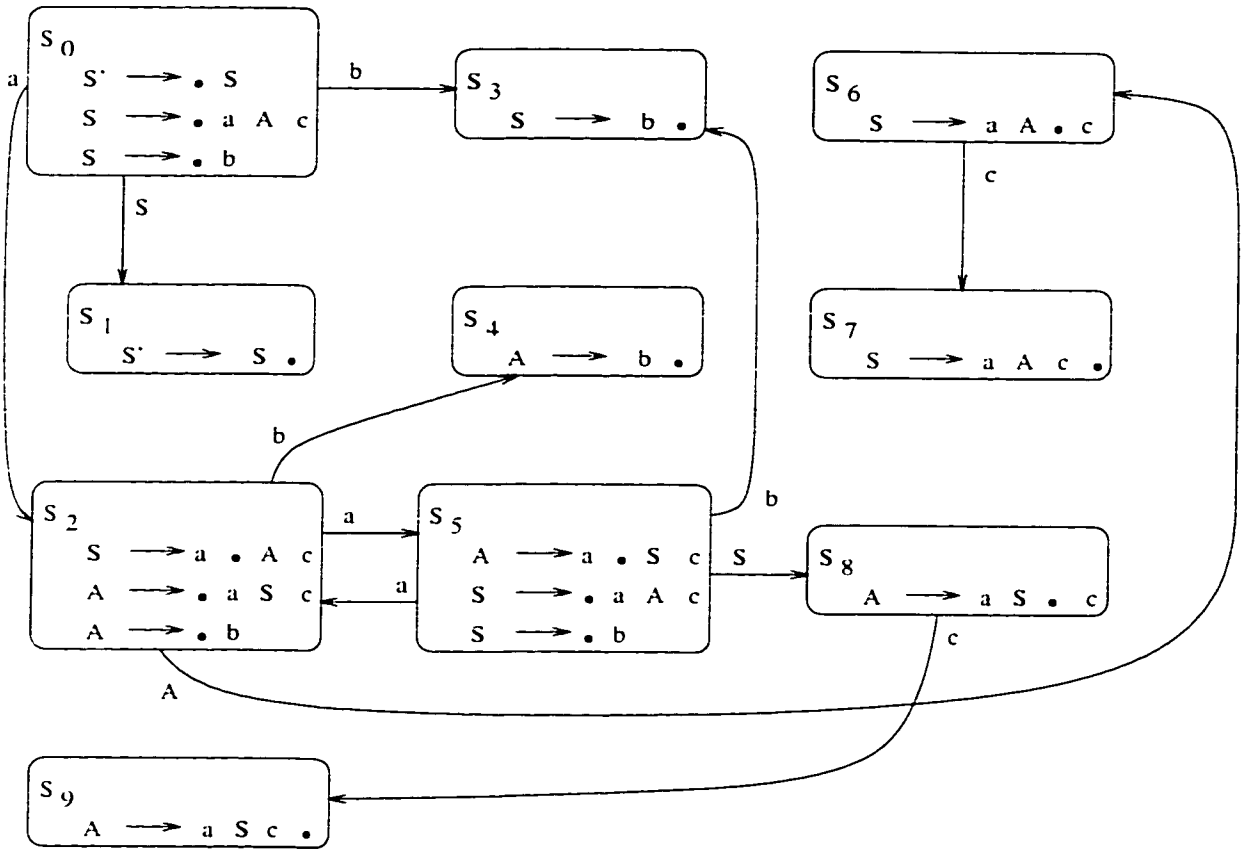


Figure 24: LR(0) parse configurations for grammar G7

state → / nonterminal ↓	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
S	S_1					S_8				
A			S_6							

Figure 25: Goto table for grammar G7

state → / nonterminal ↓	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
a	S 2		S 5			S 2				
b	S 3		S 4			S 3				
c				R 2	R 4		S 7	R 1	S 9	R 3
$		accept								

Figure 26: Action table for grammar G7

1. If $A \rightarrow x \cdot a\omega$ is in state m for input symbol a, and $A \rightarrow xa \cdot \omega$ is in state n, then enter Sn at Action$[m, a]$.

2. If $A \rightarrow \omega \cdot$ is in state n, then enter Ri at Action$[n, a]$ where i is the production number of $A \rightarrow \omega$, and a is in FOLLOW(A).

3. If $S' \rightarrow S \cdot$ is in state n, then enter "Accept" at Action$[n, \$]$.

4. If $A \rightarrow x \cdot B\omega$ is in state m, and $A \rightarrow xB \cdot \omega$ is in state n, then enter n at Goto$[m, B]$

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

The SLR(1) parser is a modification of another parser called the *LR(0)* parser. The states of the LR(0) parser are built in the same way described above. However, no lookahead is used to create the table. Unfortunately, LR(0) parsers do not recognize the constructs one find in typical programming languages. For example consider grammar G7, while it is straight forward to construct the SLR(1) parser, shown in Figure 27, it is impossible to generate an LR(0) parser for this grammar. This is due to the fact that some of the states indicate both a shift and reduce actions. This called a conflict. An example of such a conflict is seen in states S_2 and S_9. The SLR(1) is not affected by this conflict since it matches the lookahead symbol with the followers of the nonterminal before choosing a reduction.

2.6.3 LR(1) Parse-Tables

Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). For example consider this unambiguous grammar G8:

$$\begin{array}{lcll} S & \rightarrow & L = R \\ S & \rightarrow & R \\ L & \rightarrow & \times R \\ L & \rightarrow & id \\ R & \rightarrow & L \end{array}$$

The canonical collection of sets of LR(0) items for grammar G8 is shown in figure 28.

Using the algorithm for building an SLR(1) parser, Action$[S_2, =]$ would contain two entries : a shift (S 6) and a reduce (R 5). It is clear that $S \rightarrow L \cdot = R$ would correspond to the shift action. On the other hand since FOLLOW(R) contains $=$, $R \rightarrow L \cdot$ correspond to the reduce action.

Even though the grammar is unambiguous, no SLR(1) parser can be built for it. If our parser has additional information about the possible follower symbols of the production, the conflict can be resolved. For example, in S_2 knowing that nonterminal R can not be followed by "=" since there is no right-sentential form of the grammar that contains an R followed by an equal sign, the reduction by $R \rightarrow L \cdot$ can be ruled out when the lookahead symbol is "=". Hence it need not be included in the parsing table.

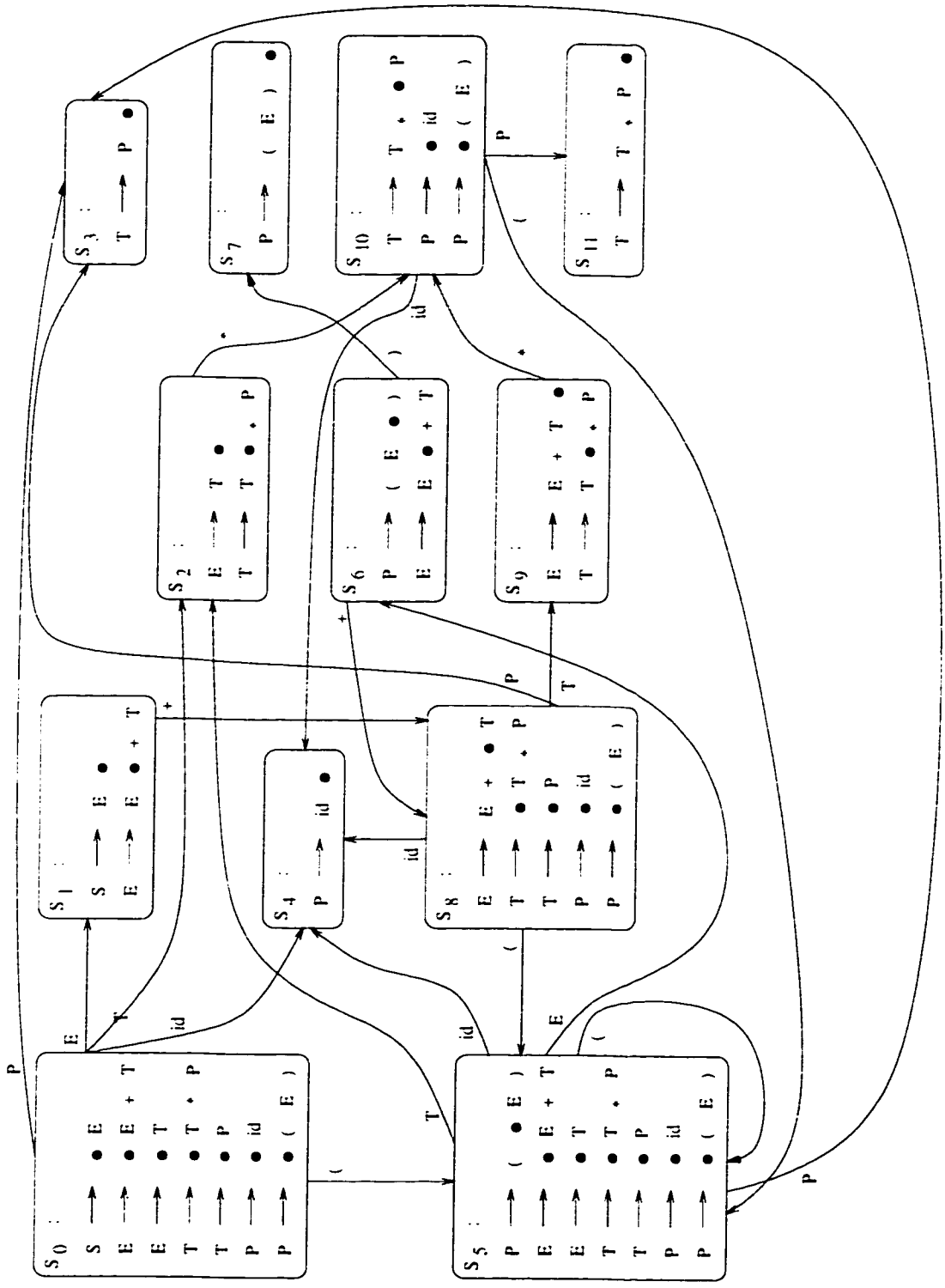


Figure 27: SLR(1) parse configurations for grammar G7

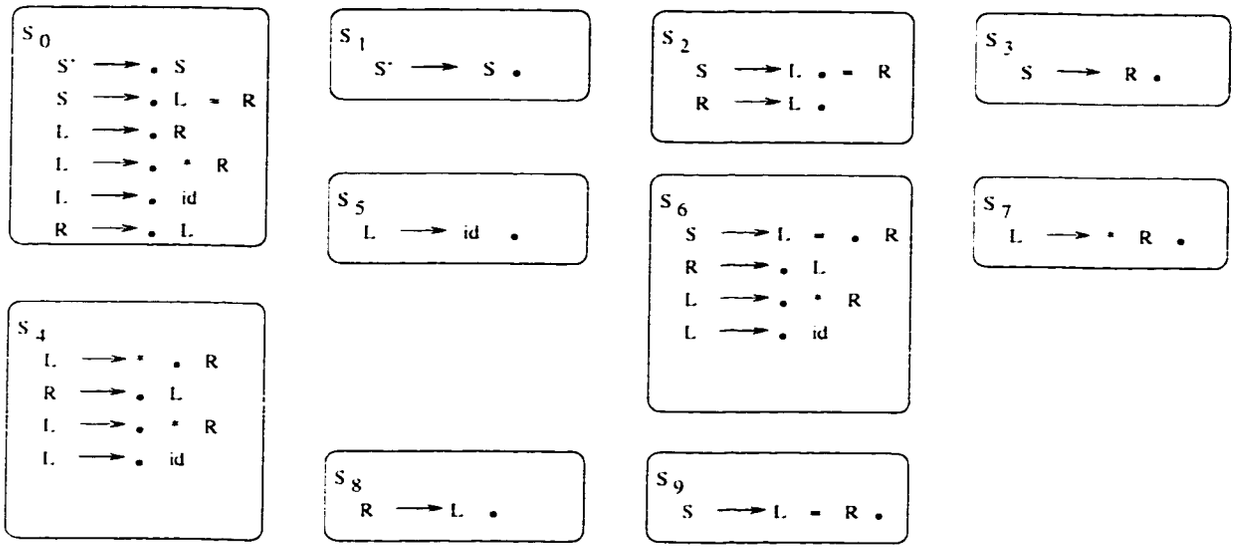


Figure 28: Cannonical LR(0) collection of sets for grammar G8

LR(1) parsing is an extension of SLR(1) parsing strategy, where each item carries additional lookahead symbols to indicate the possible follower symbols of the production. The general form of an item becomes $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the right end marker \$. For example $[E \rightarrow E \cdot +T, \$/+]$ indicates that we have seen an E and are expecting a "$+T$" which may then be followed by the end of string (indicated by \$) or by a "+". We call such an object an LR(1) item. The 1 refers to the length of the second component called the lookahead of the item. The lookahead has no effect in an item of the form $[A \rightarrow \alpha \cdot \beta, a]$, where β is not ϵ since the next input symbol should be in FIRST(β). On the opposite, an item of the form $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.

LR(1) parsing will succeed on a larger collection of grammars than SLR(1). It should be pointed out, however, that there are unambiguous grammars for which every LR parser will produce a parsing table with parsing action conflicts. Fortunately such grammars can be avoided in programming language applications.

Constructing the States

The algorithm for constructing the states is as follows :

1. Create a new nonterminal S' and a new production $S' \rightarrow S$ where S is the start symbol.
2. If S is the start symbol, put $[S' \rightarrow \cdot S, \$]$ into a start state called state 0.
3. For every item in a state, add the closure of the set containing that item to the state. The closure is calculated as follows: If $[A \rightarrow x \cdot Xy, L]$ is in the state, where L is the set of lookaheads, then add $[X \rightarrow \cdot z, M]$, where M is set of FIRST(yl) for each l in L, to the state for every production $X \rightarrow z$ where z is any sentential form.

4. Look for an item of the form $[A \rightarrow x \cdot z\omega, L]$ where z is a single terminal or nonterminal and L is the set of lookaheads, and build a new state from $[A \rightarrow xz \cdot \omega, L]$. Include in the new state all items with $\cdot z$ in the original state.

5. Continue until no new states can be created. A state is new if it is not identical to an old state.

Filling the Action and Goto Part

Consider grammar G8. Using the algorithm above we can build all the states of the LR(1) parser. The transition diagram for the DFA that accepts viable prefixes of this grammar is shown in figure 29.

Building this DFA is straight forward and not different from building the DFA from the LR(0) states earlier. In both cases, all what we have to do is to follow the dot and record by which symbol it had moved. The second component of the the item, has no effect on choosing the label of the transition. Thus the goto table still can be filled as we did for the SLR(1) goto table.

Since the second component of the item has no effect on items where the dot is not on the rightmost side of the first component, no modification also in this case on the algorithm for building SLR(1) tables is needed.

The slightly modified algorithm for building LR(1) parsing tables form that of SLR(1) is shown below:

1. If $A \rightarrow x \cdot a\omega$ is the first component of an item in state m for input symbol a, and $A \rightarrow xa \cdot \omega$ is the first component of an item in state n, then enter Sn at Action$[m, a]$.

2. If $[A \rightarrow \omega \cdot, a]$ is in state n, and A is different then S', then enter Ri at Action$[n, a]$ where i is the production number of $A \rightarrow \omega$.

3. If $[S' \rightarrow S \cdot, \$]$ is in state n, then enter "Accept" at Action$[n, \$]$.

4. If $A \rightarrow x \cdot B\omega$ is the left component of an item in state m, and $A \rightarrow xB \cdot \omega$ is left component of an item in state n, then enter n at Goto$[m, B]$

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

Every SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar.

2.6.4 LALR(1) Parse-Tables

A LR(1) parser has more states than an SLR(1) parser. For a language like Pascal the number of states for an SLR(1) parser is several hundred compared to several thousands states for an LR(1) parser for a language of the same size.

An LALR parser accepts as many constructs as an LR(1) parser, yet it has many fewer states than an LR(1) parser. In fact SLR and LALR parsers have always the same number of states.

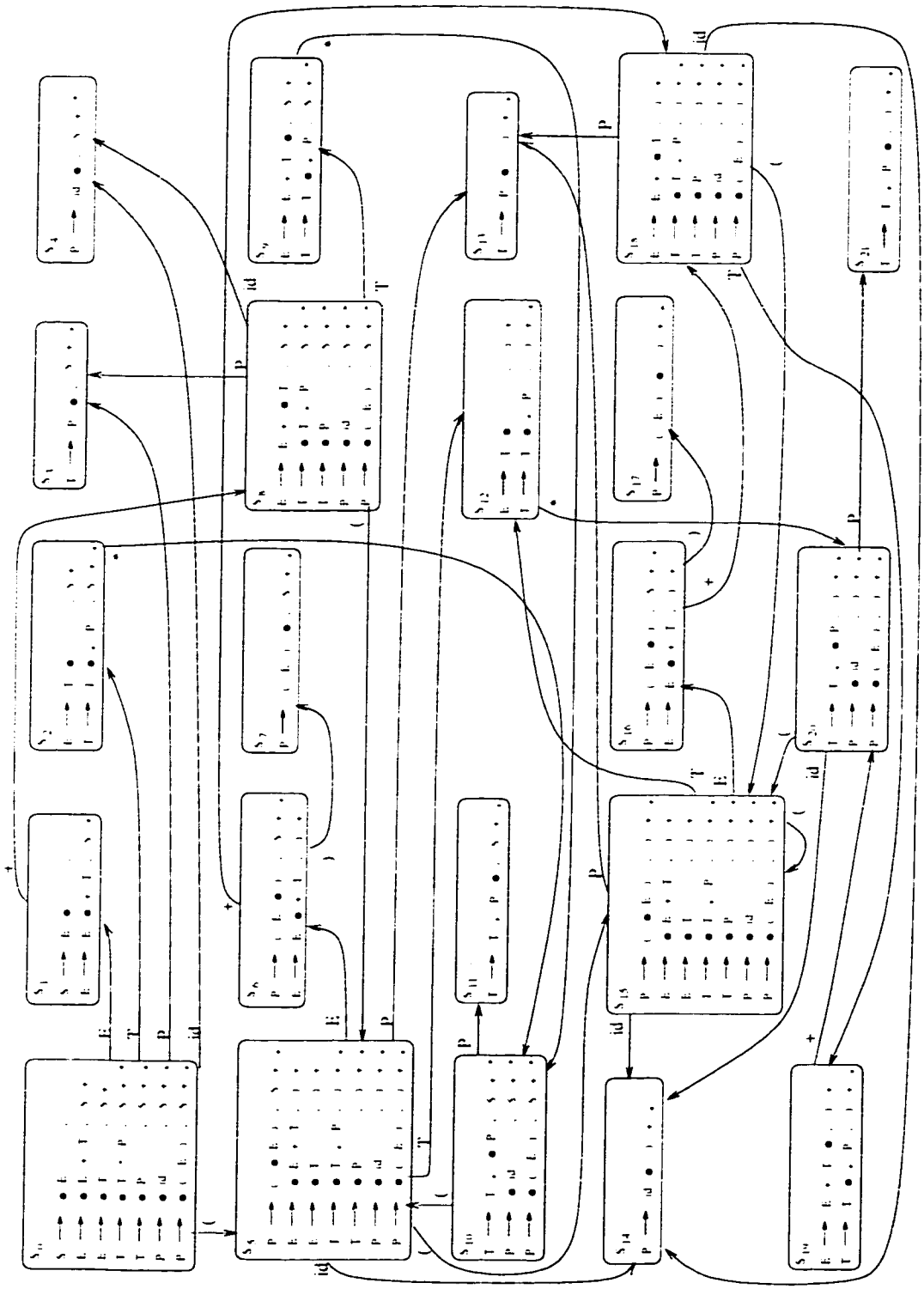


Figure 29: LR(1) parse configurations for grammar

Therefore, LALR parsers are more often used in practice since both of them have the same number of states yet an LALR accepts a much larger set of grammars than a SLR(1) parser.

An LALR parser is constructed by first constructing the LR(1) sets of items, and merging the sets whose items have the same first components. The merging process consists of taking the union of the lookahead sets of corresponding items: the result of each merger is one of the LALR states.

For example, by looking at figure 29 we notice that states S_2 and S_{12} have the same core: $\{E \rightarrow T\cdot, T \rightarrow T \cdot \star P\}$. Thus, they should be merged if a LALR(1) parser were to be built. The merger of these two states consists of merging corresponding items, i.e. items with the same first component, from each state together. $[E \rightarrow T\cdot, \$/+]$ and $[T \rightarrow T \cdot \star P, \$/ + /\star]$ from S_2 corresponds with $[E \rightarrow T\cdot,)/+]$ and $[T \rightarrow T \cdot \star P,)/ + /\star]$ from S_{12} respectively. When merging two items, the first component remains as is, but the second component, the lookahead, becomes the union of the lookaheads from the two items. Thus, merging items $[E \rightarrow T\cdot, \$/+]$ and $[E \rightarrow T\cdot,)/+]$ will give a new item $[E \rightarrow T\cdot, \$/)/+]$. Figure 30 shows all the LALR(1) states obtained from the LR(1) states of Figure 29.

Drawing the transitions between the states of figure 30 is no different from drawing them for an LR(1) parser. The goto table is constructed as follows. If J is the union of one or more states of LR(1) items, then GOTO(J , X), where X is a non terminal, is the state whose core set is the union of the core sets of the GOTO(K , X) states for every K that is one of the merged states. The action table entries are constructed as for the canonical LR parser.

It is not possible for a merger of two states to lead to a shift/reduce conflict that was not already existing in LR parser, but it is possible for it to produce a reduce/reduce conflict. The grammar is said to be LALR(1) only if the parsing table contains no parsing action conflict. Many algorithms have been described, to render the construction of LALR(1) parsers more efficient [DP82].

2.7 Semantic Actions

There are some characteristics of the structure of programming languages that are difficult to describe with BNF, and some that are impossible. As an example of a language rule that is difficult to specify with BNF, consider type compatibility rules. In Pascal, for example a real type value can not be assigned to an integer type variable, although the opposite is legal. Although this restriction is possible with BNF, it requires additional nonterminal symbols and rules. If all of the typing rules of Pascal were specified in BNF, the grammar would become many times larger. The size of the grammar determines the size of the parser.

As an example of a language rule that can not be specified in BNF, consider the common rule that all variables must be declared before being referenced. This rule can not be specified in BNF [Seb96].

A syntax-directed definition is one generalization of a context-free grammar in which additional information that was either too difficult or impossible to represent with a BNF can be added. Information is associated with a programming language construct by attaching attributes to the grammar symbols representing the construct. An attribute can represent anything we choose: a string, a number, a type, a memory address. Each symbol in the grammar can have one or many

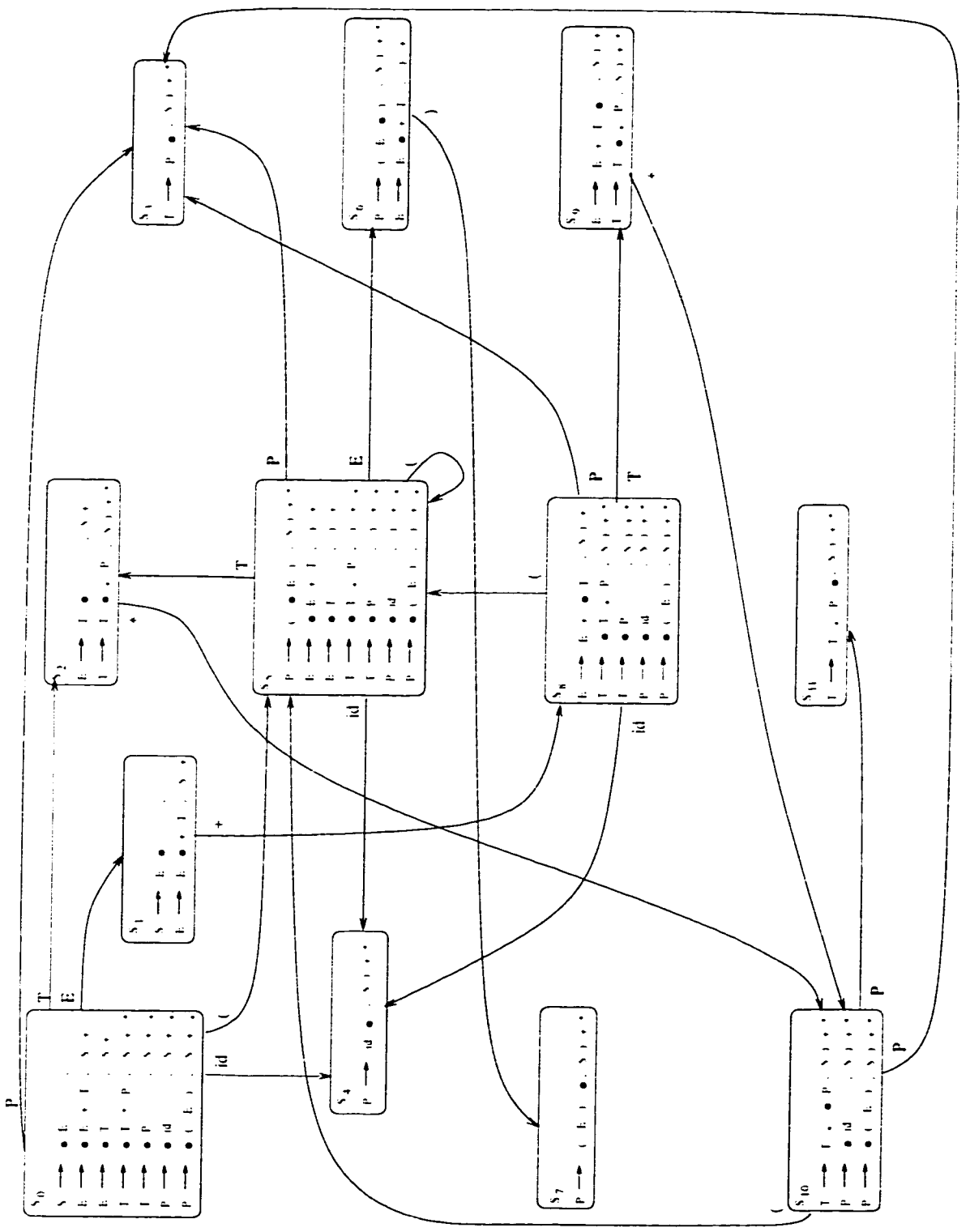


Figure 30: LALR(1) parse configurations for grammar G8

attributes.

Values for attributes are computed by semantic rules associated with the grammar productions. In a syntax-directed definition each grammar production has associated with it a set of semantic rules. Semantic rules are not only used to assign values to attribute, they can also be used to print the value of an attribute or to update a global variable. Such actions are called side effects. Figure 31 shows an example of syntax-directed definition.

Syntax	Semantics
$E_0 \rightarrow E_1 + T$	E_0 . Value = E_1 . Value + T . Value
$E \rightarrow T$	E . Value = T . Value
$T_0 \rightarrow T_1 * F$	T_0 . Value = T_1 . Value * T . Value
$T \rightarrow F$	T . Value = F . Value
$F \rightarrow (E)$	F . Value = E . Value
$F \rightarrow id$	F . Value = id . Value

Figure 31: Syntax-directed definition of a simple grammar.

Syntax-directed definitions are more commonly called attribute grammars. Attribute grammars, introduced by Knuth [Knu68], are syntax-directed definitions in which semantic rules can not have side effects on attributes and global variables.

Semantic actions are often enclosed within braces and are inserted within the right sides of productions. Such a notation is called a translation scheme.

Consider the expression $2 * 4 + 3$ and the syntax-directed definition in figure 31. If we build a parse tree for this expression and attach to every node in the tree the values of its attributes calculated according the semantic rules shown in our syntax-directed definitions, we obtain the tree shown in figure 32 . Such a tree is called an ***annotated parse tree.***

As noticed from the example of the syntax-directed definition in figure 31, and as is often the case, the value of an attribute is dependent on other attributes in the grammar. Based on these dependencies, attributes are divided into two subsets: **synthesized** and **inherited** attributes. If we consider a parse tree, a synthesized attribute at a node is one whose value is computed from the values of attributes at the children of that node. The value of an inherited attribute is computed from the value of attributes at the siblings and parent of that node.

The dependency between attributes is very important because it dictates the order in which attributes must be evaluated. For example, if an attribute b at a node in a parse tree depends

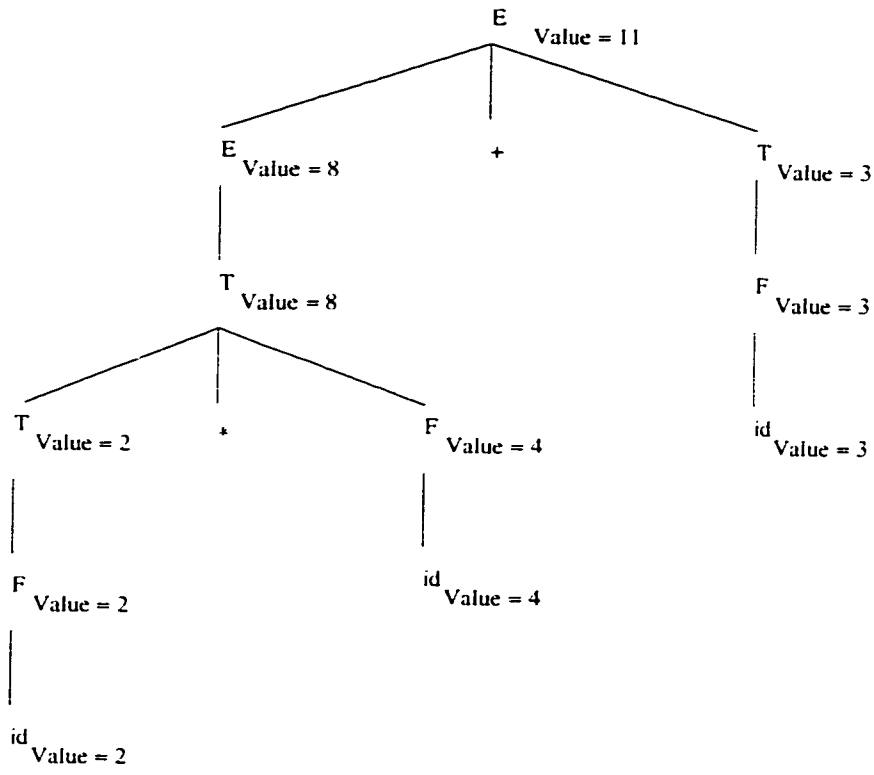


Figure 32: Annotated parse tree for "$2 * 4 + 3$".

on an attribute c. then the semantic rule for b at that node must be evaluated after the semantic rule that defines c. The interdependencies among attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph. Figure 33 shows the dependency graph for the syntax-directed definition in figure 31. An edge in the graph goes from an attribute a to an attribute b, if b depends on a.

Once the parse tree and the dependency graph are completed, a correct evaluation order for the semantic rules can be obtained. The application of the obtained evaluation order to evaluate attributes results with the correct values. This method is described in further detail by Ramanathan and Kennedy [KR77]. The only restriction put on the syntax-directed definition in this method is that the dependency graph can not contain any cycles.

There is another method for evaluating attributes which does not impose any restrictions on the syntax-directed definition. This method does not even require building a dependency graph. The following algorithm describes the method:

1. Start with the root node of the parse tree.
2. Evaluate all inherited attributes of the current node, if there are any.
3. On every child node of the current node starting from left to right apply steps 2,3 and 4.
4. Evaluate all synthesized attributes of the current node, if any.

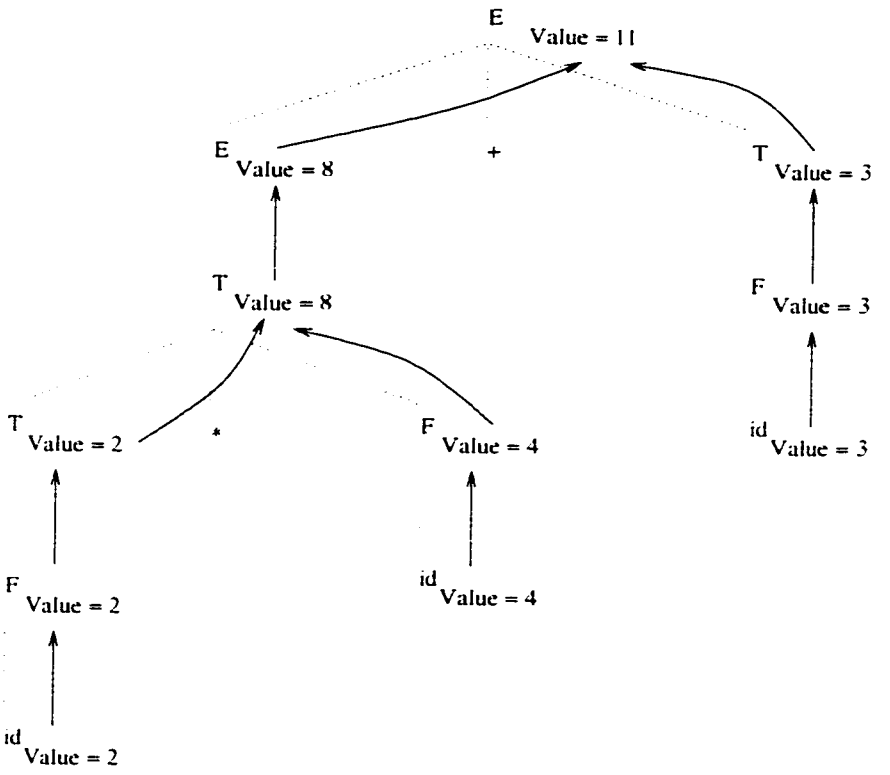


Figure 33: Dependency graph .

This algorithm requires gigantic storage requirements due to its recursive nature.

There are still many algorithms to evaluate attributes, however, most of them like the two already described require a parse tree to be built before starting the evaluation of attributes . In addition, more than one pass is needed. The complexity of these algorithms is inherent from the desire of unrestricting our syntax-directed definitions. Obviously, a top-down parser would have no difficulty evaluating attributes if all of them were inherited. Similarly, a bottom-up parser would have no difficulty evaluating attributes if all of them were synthesized. Special cases of syntax-directed definitions, like *L-attributed* and *S-attributed* definitions, that contain both synthesized and inherited attributes do not require an explicit construction of a parse tree, rather attributes can be evaluated in a single pass by evaluating semantic rules during top-down and bottom-up parsing. Thus they are worth studying.

When L-attributed definitions are attached to an LL(1) grammar, attributes values can be computed in a top-down fashion as input is being parsed. A syntax-directed definition is L-attributed if it has the following :

1. Each inherited attribute of a right-side symbol of a production must depend only on the inherited attributes of left-side symbol and/or arbitrary attributes of symbols on its left.

2. Each synthesized attribute of a left-side symbol of a production must depend only on the inherited attributes of that symbol and arbitrary attributes of right-side symbols.

S-attributed definitions imposes a further restriction on L-attributed definitions. This allows the attributes in an S-attributed definition to be evaluated at parse time during LR(1) parsing. An attribute grammar is S-attributed if and only if :

1. It is L-attributed.
2. Nonterminals have have only synthesized attributed.

2.8 Advantages/Disadvantages of Each Parsing Strategy

2.8.1 Table-driven Versus Recursive-descent LL(1) Parsing

Predictive recursive-descent parsing has the following characteristics:

- Easy to hand-write it. It does not explicitly use a stack. Instead, it relies on the implicit recursion stack. Because of this, the relation between the parser and the underlying grammar is usually clear.
- Semantic actions are easily embedded in the parsing routines. Because the parser works by prediction and parsing is modelled using functions, it is very easy to know at which point the semantic action is to be inserted.
- A parameter mechanism or attribute mechanism comes virtually for free: The parser generator can use the parameter mechanism of the implementation language.
- Non-backtracking recursive descent parsers are quite efficient, often more efficient than the table-driven ones.
- Dynamic conflict resolvers are implemented easily.

The main drawback to using recursive descent is:

- The size of a recursive-descent parser is usually larger than a table-driven one (including the table). However, this becomes less of a problem as computers memory gets bigger and bigger.

2.8.2 LL(1) Versus LALR(1) Parsing

The main differences between LL(1) and LALR(1) can be summarized as follows:

- Top down parsing is more suitable for online parsing.
- Theoretically, the set of languages accepted by top-down parsers is limited, compared with bottom-up parsers. However, most programming languages are LL(1) or are nearly so. A language such as Pascal was written with an LL(1) grammar. LALR(1) can handle a much larger set of grammar than LL(1).

- LL(1) generally requires larger modifications to be made to the grammar than LALR(1). Severe restrictions are imposed on the form of the productions in a LL(1) grammar. Specifically, left-recursion and common prefixes can not be used. If the language is not LL(1) the grammar can be transformed into LL(1) form, but the result is usually less clear.
- LL(1) allows semantic actions to be performed even before the start of an alternative; LALR(1) performs semantic actions only at the end of an alternative.
- LL(1) parsers are often easier to understand and modify.
- If an LL(1) parser is implemented as a recursive-descent parser, the semantic actions can use named variables and attributes, much as in a programming language. No such use is possible in a table-driven parser.
- LL(1) has a little edge on LALR(1) as to speed and memory requirements [Hol90].
- It is more difficult to make modifications to a recursive-descent parser since the parser is in the code not in a table.
- Error handling is similar in both techniques.

2.9 Summary

In this chapter we present a review of parsing. Early attempts to automate parsing are explained. Top-down parsers and their problems are explained. Specifically table-driven LL(1) parsers and recursive descent LL(1) parsers. A brief introduction about LL(k) was given. Bottom-up parsers are introduced. A description of the bottom-up LR parsers is given. Variants of LR parsing such as SLR(1), LR(1), and SLR(1) are also explained. Semantic actions are introduced. The chapter concludes with a comparison detailing advantages and disadvantages of table-driven versus recursive-descent LL(1) parsers, and of LL(1) versus LALR(1) pasres.

Chapter 3

Parser Generators

3.1 What Does a PG Do?

A parser generator is a program which compiles a syntax definition, usually in the form of a grammar, into a parser for the language defined. The parser must accept any sentence of the language, and reject any other input string.

As shown in figure 3-1, there are two possible forms in which a parser can be generated. In the first case (1), the parser is a procedure, or a set of procedures. This is usually produced as a text file containing a parser in source form. The second possibility (2) is to produce a parsing table. This is a sequence of data values which encode parsing decisions for a fixed parsing procedure, called the driver. The driver is usually supplied separately, often in a pre-compiled form.

Typically, the output of a parser generator is source code for some programming language (such as C, C++, or Java), which can then be compiled and linked to the client application. Semantic actions defined within the grammar are used to control how the client application processes data elements as they are parsed, by binding specific symbols from the grammar to programmer-written application code. When a given symbol is parsed by the parser, application code corresponding to the semantic action for that symbol gets executed. Semantic actions are usually defined directly within the grammar definition and consist of compilable statements from the target programming language. In many cases, semantic actions are used to construct internal representations of the parsed data for subsequent use by the application.

Shortly after CFG's were introduced for defining programming languages, programmers noticed that the rules of a grammar constituted a kind of flow-chart for a parsing procedure. Subject only to the restriction that the grammar not contain any left recursion, an experienced programmer could easily translate any grammar into a recursive-descent backtracking parser. Equivalent table-driven procedures were also described. These parsers were too inefficient to use in compilers production, but they demonstrated that parsers could be generated automatically from suitable grammars. After the theory of LL(1) grammars was formalized, it became possible to produce efficient top-down parsers

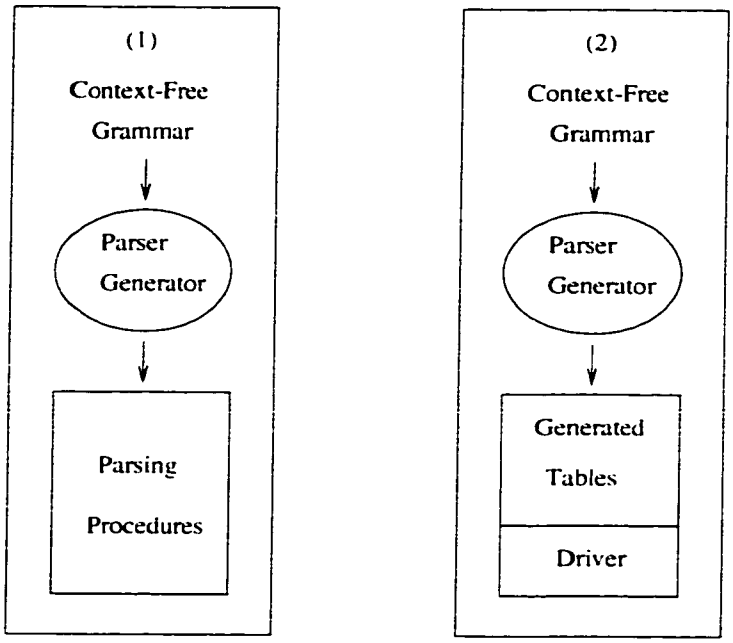


Figure 34: Automatic parser generation

mechanically, in either recursive descent or table-driven forms.

3.2 Approaches to Parser Generators

There is a very large number of parser generators in existence today. To name a few: AnaGram, Holub, LLgen, Lisa, Mango, Muskox, Precc, LalrGen, Trooper, VisualParse++, Yacc, Yacc++, Yocc, Antlr, GrammaTech, Ell, Cocktail, Javacc, Jell, ProGrammer, Cogencee, Coco/R, Depot4, Gray, wacco.

Most of the parser generators are similar in the techniques they use. Moreover, only few of them are object-oriented, or even claim to be object-oriented. In an object-oriented parser the emphasis should be on the data, in this case the grammar and its symbols. It is not enough to use the traditional parsing techniques and wrap everything in a class called Parser for the parser to be called object-oriented.

In this section we are going to introduce four approaches to parsing and parser generators that touch on the topic of object oriented parsers and parser generators.

3.2.1 Yacc

Today, it can be said that the most used parser generator is Yacc [Joh75]. Yacc was written by Steve Johnson at Bell Telephone Laboratory in 1975 for the Unix operating System and since then it has been ported to other environments. The word Yacc stands for Yet Another Compiler Compiler.

Yacc, written in C, takes as input a description of a grammar in a notation near to BNF and

produces a parser which is also a C program. Each production in the grammar may have an action associated with it. The class of specifications accepted by Yacc is LALR(1) grammars. Yacc uses also some disambiguating rules to deal with some ambiguous grammars.

The parser generated is a push-down automaton consisting of a large stack to hold current states, a transition matrix to derive a new state for each possible combination of current state and next input symbol, a table of user-definable actions which are to be executed at certain points in the recognition, and finally an interpreter to actually permit execution. The result is packaged as a function yyparse(), which calls repeatedly a lexical analyzer function to read standard input.

The input specification has three parts :

1. Declarations and definitions
2. Grammar : Rules in near-BNF notation plus associated actions
3. User-written procedures in C

Sections are separated from each other by %%.

Grammar and Declarations & Definitions Section

The grammar section of Yacc consists of a grammar which specifies the syntax of some language using a BNF-like format. A quoted single character, '+' for example, is taken to be the terminal symbol + and it means we are looking for character '+' on the input. Terminals that can not be described by a single character, an identifier for example, must be declared in the declaration section, as follows :

$$\%\ token \quad ID$$
$$\%\ token \quad NUMBER \quad BEGIN \quad END$$

Unquoted strings of letters and digits that are not declared in the declaration section are considered to be non terminal symbols. A colon is used to separate the left-hand side of a production from its right-hand side, not an arrow. The rules having the same left-hand side are separated by a vertical stroke: |. A semicolon follows each left side with its alternatives and semantic actions. Empty rules are denoted by rules having no symbols on the right-hand side. The left-hand side of the first rule in the grammar is by default the starting symbol. This default can be overwritten by including a command like this in the declaration and definition section :

% start E

A code segment, one or many C statements, may be associated with each grammar rule in Yacc. Once the parser can reduce the right-hand side of some rule, the semantic routine corresponding to this rule is invoked. The writer of the grammar can use this code to perform semantic processing, code generation, and so forth. The code is included directly in the grammar by following the rule, but before the semicolon, with "{*code*}" where *code* is the segment of code to be executed. The code, therefore, precedes a vertical stroke or a semicolon. Yacc also allows users to put an action in the middle of a rule, such as in :

```
FIRST   : Second
                                    { $$ = 1: }
          Third
                                    { x = $2 : y = $3 : }
        ;
```

Such actions put an extra burden on Yacc since it has to introduce a new nonterminal symbol to match the action in the middle as in :

```
DUMMY   :
                                    { $$ = 1 : }
        ;

FIRST   : SECOND  DUMMY  THIRD
                                    { x = $2 : y = $3 : }
        ;
```

Variables used in the code must be defined in the declarations and definitions section. These variables are declared in that section as ordinary C declarations delimited by %{ and %}. Between these two %{ and %} we also specify any global variable used by the functions of the third section, and also the names of the header files to be included in our generated parser can be specified using the usual C notation as in :

#include < *string.h* >

By default Yacc associates with each grammar symbol an attribute value. These attribute values can be referenced in a semantic actions using variables $$, $1, $2, and so on. $$ refers to the attribute value of the non terminal symbol on the left-hand side of the rule. $1 and $2 will refer the first and second symbol on the right hand of the production respectively. In general $*i* will refer to *ith* symbol, terminal or nonterminal, on the right. Terminal symbols receive their values when the scanner assigns a value to the external variable *yylval*. A non terminal, on the other hand, needs to receive its value via a semantic action. For example :

```
expr  : expr  '*'  term          { $$ = $1 * $3 : }
      | expr  '-'  term          { $$ = $1 - $3 : }
      | term
      ;
```

When no action is specified, as in the third alternative in the example above, the default action is {$$ = $1;}. Thus the value of *term*'s attribute would be copied to that of *expr*. The default attributes are all of type integer, so there is no problem copying one to the other.

Yacc also allows the user to define attributes of other types, including structures. The stack where Yacc stores the values of attributes is declared to be a union that the user can redefine in the declarations and definitions sections as follows:

```
%union  {
            body  of  union  ···
        }
```

Once the union is declared, the union member names must be associated with the various terminal and nonterminal names. This association is specified in the declaration section. The construction $< name >$ is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, %noassoc as follows :

```
% token   < name >  ID  FUNCTION
```

the union member name is associated with the tokens listed. The keyword %left, %right, and %noassoc are used to declare operators associativity. As the names imply, %left, %right, and %noassoc means left associative, right associative and, not associative respectively.

As for nonterminals, attributes can be associated to them in the same way as terminals, but using the keyword %type in place of %token.

Yacc will automatically insert the appropriate union member names to each token and nonterminal symbol having a value. However, there are many situations where it is difficult for Yacc to figure out the types of values, and the user is requested to explicitly insert the union name between < and > after the first $, as in :

```
FIRST  :  SECOND
                    {  $ < val1 > $  =  3  :  }
          THIRD
                    {  x  =  $2  :   y  =  $3  :  }
       ;
```

"This syntax has little to recommend it..." the author of Yacc says [Joh75].

User-Written Procedures Section

This section contains the main program that invokes the parser. The user can write there functions that are needed in his program. For example an error recovery function called *yyerror* can be provided. When Yacc encounters an error, it just prints a message, it is up to the user to rewrite *yyerror* to supply more information.

Figure 35 gives a clearer idea about the input file for Yacc.

3.2.2 Yacc Meets C++

In a paper entitled "Yacc Meets C++" [Joh88], Johnson the creator of Yacc suggests some modifications on Yacc.

Yacc, the parser generator, allows attributes to be associated with grammar symbols. Yacc uses a LALR(1) method which is a bottom-up parsing algorithm, hence only synthesized attributes can be handled directly. More complex translations must be done by building a parse tree, and then walking this tree doing the desired actions.

```
%{
    #include <stdio.h>
%}
%start      Program
%token      Id
%token      Number

%%

Program     :   Statements                  {printf("Program \n");}
            ;

Statements  :   Statement   Statements      {printf("Statements \n");}
            |   Statement                   {printf("Statements \n");}
            ;

Statement   :   Id      '='   Expression    {printf("Statement \n");   $$ = $2; }
            ;

Expression  :   Expression  '+'  Term        {printf("Expression \n");  $$ = $1 + $2; }
            |   Expression  '-'  Term        {printf("Expression \n");  $$ = $1 - $2; }
            |   Term                        {printf("Expression \n");  $$ = $1; }
            ;

Term        :   Term    '*'  Factor         {printf("Term \n");        $$ = $1 * $2; }
            |   Term    '/'  Factor         {printf("Term \n");        $$ = $1 / $2; }
            |   Factor                      {printf("Term \n");        $$ = $1;      }
            ;

Factor      :   '('  Expression  ')'        {printf("Factor \n");      $$ = $1;      }
            |   Id                          {printf("Factor \n");      $$ = $1;      }
            |   Number                      {printf("Factor \n");      $$ = $1;      }
            ;

%%

main()
{
    yyparse();
}
```

Figure 35: Yacc specification for the sequence of assignment statements.

In Yacc, a unique datatype may be associated with each nonterminal symbol. In this case, every rule deriving that nonterminal must return a value of the defined type. Each token may also have a defined type; in this case, the values are computed by the lexical analyzer. An action computes the value associated with the left side of a rule; this action depends on the particular rule and the values of the components on the right of the rule.

Languages such as C++ support abstract data types that permit functions as well as values to be associated with objects of a given type. Since values and actions are associated with a nonterminal, it looks natural to represent a nonterminal as a C++ object. Johnson shows us how C++ abstract data types can be applied to the semantic actions associated with a parser to achieve the effect of attribute grammars.

Some examples can clarify how functions can be defined on these types (C++ object) and incorporated in the grammar to replace actions used in Yacc.

Consider this rule:

$$expr \quad : \quad expr \quad '+' \quad expr \quad :$$

We can define a function print() on the object 'expr' as:

$$print() \quad \{ \quad \$1.print() \quad : \quad putchar('+') \quad : \quad \$3.print(); \quad \}$$

We might also define another function type() as:

$$type() \quad \{ \quad return \quad (\quad exprtype \quad (\quad '+' \quad , \quad \$1.type() \quad , \quad \$3.type() \quad) \quad) \quad : \quad \}$$

This shows us that using functions we can take advantage of synthesized attributes. By allowing these rule-defined functions to have arguments and return values we get many of the effect of inherited attributes:

$$type(type \quad t) \quad \{ \quad \$1.type(t) \quad : \quad \$2.type(t) \quad : \quad \}$$

In Y++ (The suggested modified version of Yacc using C++) when a function f is defined on a nonterminal/type X, not only the value, but also the function definition itself depends on the rule used to define the instance of X, for example consider the rule above. Two functions polish and revpolish can be defined on it.

$$polish() \quad \{ \quad putchar('+') \quad : \quad \$1.polish() \quad : \quad \$3.polish() \quad : \quad \}$$

and

$$revpolish() \quad \{ \quad \$1.revpolish() \quad : \quad \$3.revpolish() \quad : \quad putchar('+') \quad : \quad \}$$

Another example is also given on these two rules defined as:

$$line \quad : \quad "PRE" \quad expr$$
$$line \quad : \quad "POST" \quad expr$$

and a function print() is defined as:

$$print() \quad \{ \quad \$2.polish() \quad ; \quad \}$$

on the first rule, and:

$$print() \quad \{ \quad \$2.revpolish() \quad ; \quad \}$$

on the second rule. Then, after a line has been recognized, print will print the expression in either prefix or postfixform depending on the initial keyword of the line.

Some features of Y++ are :

1. Every grammar symbol, token and nonterminal alike, is associated with a C++ class.
2. Every class has 0 or more functions, defined on it.
3. Every grammar rule may have associated with it 0 or more functions that may be invoked to access and change the values accessible to that rule. These functions may access and change values of the result (left side) of the rule, and the components (right side) of the rule, and invoke other functions defined on the components of the rule.

In practice, Y++ specifications are transformed to C++ programs and compiled. yyparse returns a value that is, in effect a pointer to the parse tree. When yyparse is called, it creates a data structure that represents the tree. For tokens it creates space large enough to hold the value, if any, included in the token. For nonterminals, the space created depends on the rule used to create to create the particular instance of the nonterminal. The rule

$$A \quad : \quad B \quad C \quad D \quad ;$$

would cause space to be allocated as follows:

```
integer rule number
space for the A values
pointer to the B value
pointer to the C value
pointer to the D value
```

In the case where the actions depend on the rule number (remember the example above), we generate a conditional based on the stored rule number.

Some of the problems Johnson speaks about are

1. **Scope:** If there are two calls to yyparse, does the second tree overwrite the first or should the two remain active?
2. **Default values:** There is little point in wasting space on characters and literal keywords when they could be known from the rule number. How can this be exploited?
3. **Default functions:** If a function is called for a rule that contains no definition for that function should a default definition be assumed?

4. **Error Handling:** There is a premium in being able to return a sensible structure for any input, even those in error, to allow the user to craft special functions that give particularly good error messages.

5. **User interface:** Given a parse tree, it is very nice to be able to rewrite it.

Y++ allows grammar and lexical analyzers to be far more reusable than yacc, since it is easy in Y++ to augment the grammar with new rules and actions.

3.2.3 Yacc++

Barbara Zino, the creator of Yacc++, explains the features of Yacc++ [Zin91].

- Yacc++ combines the lexing and parsing functions.
- Yacc++ is based on the O-O programming model, the features of which enhance ease of programming, use and reuse of lexer and parser objects.
- Yacc++ improves upon Yacc BNF by providing for regular expression constructs, which is a simpler and more natural way to describe languages. (lists, optional items).
- **Multiple Class-Based Inheritance:** Yacc++ takes advantage of multiple inheritance of the following way: The base class contains token and parser nonterminal rules. The rules are accompanied by semantic actions. Derived classes inherit tokens and parser nonterminals rules, and their associated semantic actions from one or more base class. Changes or additions to the token and/or parser nonterminals and their semantic actions can be made in the derived classes, without any need for rewriting the grammar.
- **Multiple-Entry-Point Parsing:** Yacc uses a structured, procedure-oriented programming model. Therefore, Yacc creates monolithic lexer and parser program that has a single entry point at the root of the parse tree. The entire parsing tree has to be matched with this lexer and parser. There is no possibility of matching just a branch of the tree. O-O programming is a data rather that a procedure-oriented programming model. Yacc++ creates one lexer and parser program and multiple data objects that can have multiple entry points into a parse tree. The lexer/parser program processes input that matches the entire pattern tree or just a selected branch of the tree. Programmers control which branches are visible to applications by preceding the list of wanted nonterminal by "PUBLIC" inside the class definition.
- **Call-Back Mode for Integration with GUIs:** Yacc produced parsers are oriented to the serialized I/O model and batch file processing. In Yacc lexer and parser procedures control the computing processes and they call I/O routines to obtain input data. With the advent of GUI that manages I/O and calls procedures to handle events, the old style of Yacc is no longer acceptable. Lexer and parser objects created by Yacc++ can operate in a call-back mode for use within the GUI notification model. The lexer and parser objects are notified of events, perform the appropriate actions and then return control to the GUI system. Lexer and parser objects operating in call-back mode are coroutines that retain state between calls.

- **Polymorphism** is the ability to manage concurrent objects that are of differing classes. Lex and Yacc create only one lexer and parser. With its GUI driven approach, Yacc++ provide the programmer with the ability to create a parser object each time a user opens a text window to check if it conforms to certain dialect.

- **Abstract Syntax Tree Objects** support the construction of an internal representation of the parsed input. This takes the form of an abstract syntax tree. Yacc++ generates classes derived from the AST base class for CONSTRUCT and BASE CONSTRUCT declarations specified by the user in the grammar. These CONSTRUCT declarations define classes for tokens, and non-terminals. Yacc++ also generates the code which constructs the AST objects during lexing and parsing.

3.2.4 Eiffel Parse Library

In his book, "Reusable Software: The Base Object-Oriented Component Libraries" [Mey94], Bertrand Meyer presents his approach to parsing. It comes as an Eiffel parse library.

Meyer summarizes the need to this library in four points:

1. The need to interface the parsing tasks with the rest of an object-oriented (O-O) system, such as a compiler, in the simplest and most convenient way.

2. The desire to apply O-O principles as fully as possible to all aspects of a system, including parsing so as to gain the method's many benefits in particular reliability, reusability, and extendibility.

3. The need to tackle languages whose structures are not easily reconciled with the demands of a common parser generator which usually requires the grammar to be LALR(1). The parse library uses a more tolerant LL scheme that allows backtracking whose only significant constraint is absence of left-recursiveness.

4. The need to define several possible semantic treatments on the same syntactic structure. For example in Eiffel, a compiler, an interpreter, a pretty-printer, software documentation tools, browsing tools, and several other mechanisms all need to perform semantics actions on software texts that have the same syntactic structure. (With Yacc, the description of syntactic and semantics are mixed, so one needs a new specification for each tool). The parse library keeps syntax and semantic analysis separate, and uses inheritance to allow many different semantic descriptions to rely on the same syntactic stem.

The grammar is divided into two kinds of constructs: a nonterminal and a terminal. Nonterminals are described in terms of productions, which give the structure of the construct's specimens. The parse library relies on the convention that every nonterminal is defined by at most one production.

A production describing a nonterminal can be one of three types:

1. Aggregate Production: defines a construct whose specimens are obtained by concatenating specimens of a list of specified constructs, some of which may be optional. Example:

$$Conditional \quad ::= \quad if \quad then\_part\_list \quad [else\_part] \quad end$$

2. Choice Production: defines a construct whose specimens are of one among a number of specified constructs. Example:

$$Type \quad ::= \quad Class\_Type \quad | \quad Class\_Type\_Expanded \quad | \quad Formal\_Generic\_Name$$

3. Repetition Production: defines a construct whose specimens are sequences of zero or more specimens of a given construct (called the base of the repetition construct) separated from each other, if any, by a separator. Example:

$$Compound \quad ::= \quad \{ \quad Instruction \quad ``;'' \quad \}$$

The nonterminal will be accordingly called an aggregate, choice, or repetition construct. A terminal construct has no defining production. It is defined by the lexical grammar and not the syntactical grammar.

The parse library represents these notions with few classes: CONSTRUCT, AGGREGATE, CHOICE, REPETITION, TERMINAL, KEYWORD, L_INTERFACE, and INPUT.

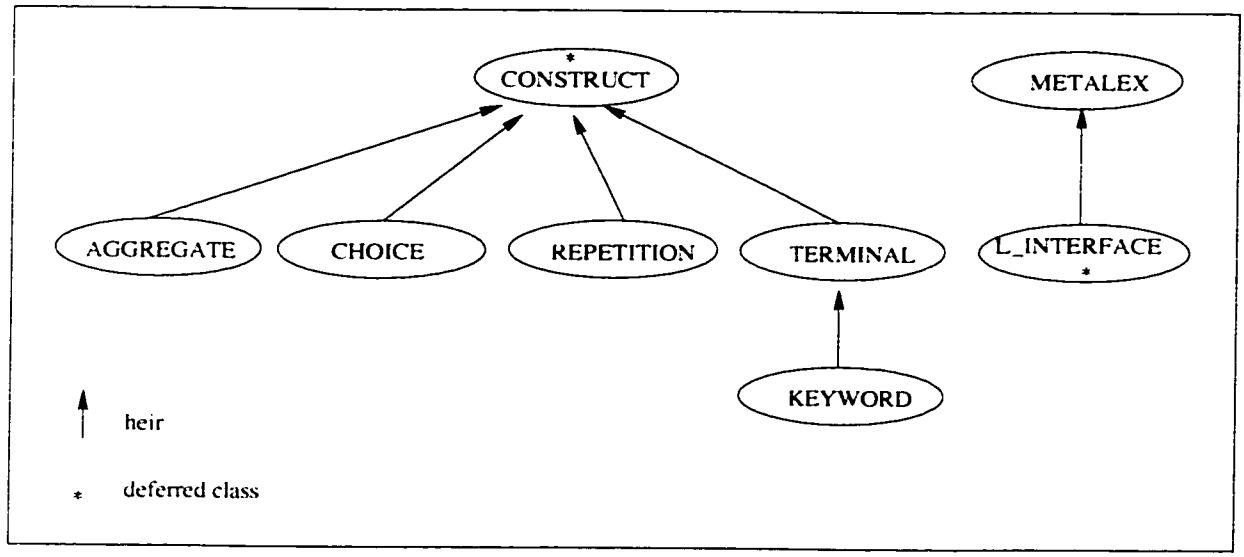


Figure 36: Class representation of the various components of grammar.

CONSTRUCT is a deferred class that describes the general notion of syntactical construct. CONSTRUCT contains a deferred function called "production", which is a direct representation of the corresponding production.

AGGREGATE, CHOICE, REPETITION, and TERMINAL inherit CONSTRUCT. Function "production" remains deferred inside these classes. Every effective construct class that the user writes must provide an effecting of that function.

Although it is deferred, CONSTRUCT describes a parsing mechanism that its descendants, the specimens of the constructs of a grammar should follow. For example, its procedure "process" appears as:

```
parse;
if parsed then
    semantics
end
```

Where parse and semantics are expressed in terms of some more specific procedures, which are also deferred. This defines a general scheme while leaving the details to the descendants of the class.

AGGREGATE, CHOICE, REPETITION, and TERMINAL should also describe the corresponding types of constructs, with features providing the operations for parsing their specimens and applying the associated semantic actions.

To build a processor for a given grammar, one writes a class, called a construct class, for every construct of the grammar, terminal or non-terminal. The class should inherit from AGGREGATE, CHOICE, REPETITION or TERMINAL depending on the nature of the construct. It should describe the production for the construct and any associated semantic actions.

To complete the processor, one must choose a "top construct" for that particular processor, and write a root class. This top construct is only defined with respect to a particular processor for that grammar. Different processors for the same grammar may use different top constructs.

Meyer argues that in O-O methods tops and roots should be chosen last, and that the concept of a top component of a grammar contradicts the O-O approach, which deemphasises any notion of top component of a system.

The effect of processing a document with a processor built from a combination of construct classes is to build an abstract syntax tree for that document and to apply any requested semantics to that tree. Class CONSTRUCT is a descendant of a class called "TWO_WAY_TREE" that describes a versatile implementation of trees. So, as a consequence, are CONSTRUCT's own descendants. The effect of parsing any specimen of a construct is therefore to create an instance of the corresponding construct class. This instance is a tree node and is automatically inserted at this right place in the abstract syntax tree.

To parse a document, one needs to get tokens from a lexical analyzer. This is achieved by making some construct classes, in particular those describing terminals, descendants of one of the lexical classes. The best technique is usually to write a class covering the lexical needs of the language at hand, from which all construct classes that have some lexical business will inherit. Such a class is a called a lexical interface class. L_INTERFACE describes this pattern followed by the lexical interface class.

Class TERMINAL includes a deferred function "token_type" of type integer. Every effective descendant of TERMINAL should effect this feature as a constant attribute, whose value is the code for the associated regular expression, obtained from the lexical interface class.

The principal features for defining semantic actions are pre_action and post_action. These are features of class CONSTRUCT. Procedure pre-action describes the actions to be performed before a construct has been recognized; post_action, the actions to be performed after a construct has been recognized.

As defined in CONSTRUCT, both pre_action and post_action do nothing by default. Any construct class which is a descendant of CONSTRUCT may redefine one or both so that they will perform the semantic actions that the document processor must apply to specimens of the corresponding construct. These procedures are called automatically during processing, before or after the corresponding structures have been parsed.

For TERMINAL, only one semantic action makes sense, so pre_action and post_action have been respectively defined unused_pre_action and action.

In order to separate syntax and semantics, Meyer suggests using the following approach:

- First write purely syntactic classes, that is to say construct classes which only effect the syntactical part (in particular function production). As a consequence, these classes usually remain deferred. The recommended convention for such syntactic classes is to use names beginning with S_, for example S_INSTRUCTION or S_LOOP.

- Then for each construct for which a processor defines a certain semantics, define another class called a semantic class, which inherits from the corresponding syntactic class. The recommended convention for semantic classes is to give them names which directly reflect the corresponding construct name, as in INSTRUCTION or LOOP.

This method described, relying on multiple inheritance, achieve the goal of letting different processors share the same syntactic descriptions.

Classes AGGREGATE, CHOICE, TERMINAL, and REPETITION are written in such a way that one does not have to take care of the parsing process. They make it possible to parse any language built according to the rules given - with one limitation, left recursion.

The parse library uses backtracking to try various choices in sequences and recursively to recognize a certain specimen. To avoid too much backtracking, function "commit" is used. A call to "commit" in Aggregate A is a hint to the parser, which means that "if you get to this point trying to recognize a specimen of A as one among several possible choices for a choice construct C, and you later fail to obtain an A, then forget about finding an A here. You may go back to the next higher-level choice before C - or admit failure if there is no such choice left. "

One disadvantage about the use of "commit" is that it assumes global knowledge about the grammar and its future extensions. Another one, is that requires a lot of care in order not to lead to the rejection of valid texts as invalid.

There is a tool that complements the parse library called YOOC(Yes ! an O-O Compiler). YOOC is a translator tool that takes a grammar specifications as input and transforms it into a set of classes, all descendants of CONSTRUCT and built according to the rules defined above.

3.2.5 Trends in Compiler Construction

Bruce Watson [Wat95] says that a naive approach to O-O parsing is to create a parser class that has a member function which constructs a parse tree, given a token stream. This approach is no more than an O-O wrapper around a traditional procedural parser.

O-O design of a compiler requires us to think in terms of the data structures between the phases of a compiler. he says.

Figure 43 shows the organization suggested for an O-O compiler by Watson. as opposing to the traditional approach shown in left diagram.

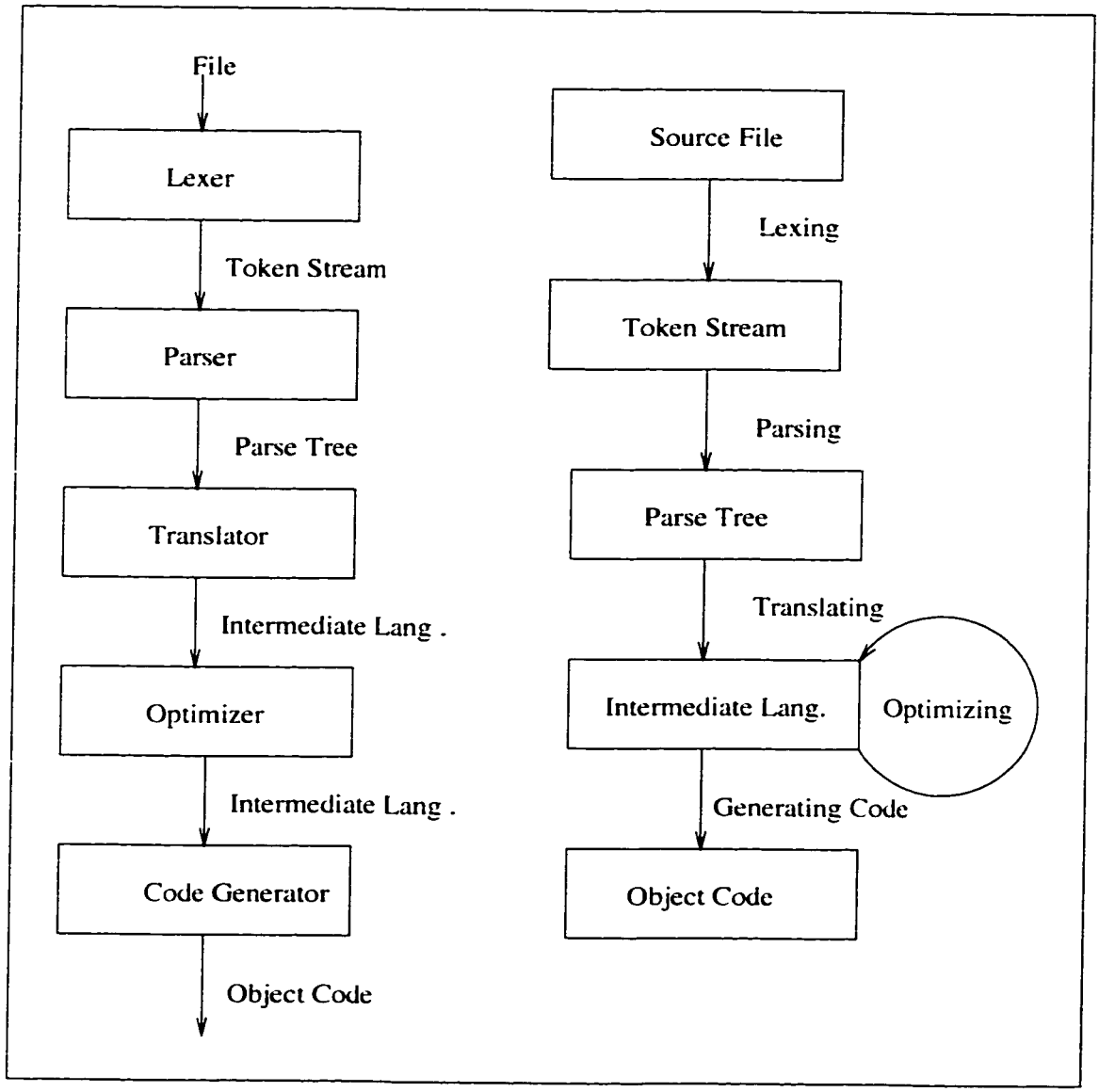


Figure 37: On the right Watson's organization of a compiler construction. on the left traditional organization.

Once the data-structures are designed (as classes), the phases of the compiler arise naturally as the constructors of each of these classes. For example, a parse tree object knows how to construct itself from a token-stream. The parsing algorithm itself will be embodied in the constructors of the parse tree nodes.Watson presents new O-O approaches for top-down and bottom-up parsing.

Watson's Top-Down Parsing

We begin by constructing an object representing the root of the parse tree. After passing the token-stream to the constructor of the root, the root is used to construct the subtrees of the root. For example in :

$$E \ : \ I \ + \ E \ | \ I$$

The constructor for E invokes the constructor for I (passing it the token stream), constructing a new I object. In the case that I is a token, the I will simply be extracted form the token stream. Then the I constructor would then use the token-stream to build its subtree. The E constructor then determines if the next constructor is a +, if it is, the + is consumed from the token stream and the constructor constructs a new E object, again passing it the token stream.

Watson's Bottom-Up Parsing

Watson [Wat95] presents a new approach to thinking about Bottom-Up parsing. Watson claims that his algorithm proved to be easier to understand than the traditional approach when tested on students.

Consider this grammar:

$$E \ : \ E \ * \ I \ | \ I$$

and let us imagine we have the parse tree available at hand. We are going to figure out how to un-parse the tree into a stream of tokens, starting with the rightmost leaf. We invoke a member function of the root object. The member function of the root is the currently active function. This member function determines that each of the object's children must be un-parsed in turn. It therefore destroys the root object, returning enough information so that the un-parse member function of its children are invoked from left to right (each simply invokes the next sibling, until the rightmost sibling). The act of destroying itself and passing control to the children is known as a produce step. The currently active member function, that of the rightmost node, un-parse itself and returns control to its left sibling. This process of returning control to the left sibling is called a shift. And so on, until the parse tree is completely decomposed. Every time a leaf is decomposed, the value of its token is printed to the screen from right to left. Figure 38 shows in a step by step the un-parsing of the bottom tree representing "$I * I * I$".

The bottom-up parsing process is the time-reversal of the above un-parse process. We start from the last step and we go backward. The reverse of a produce step is the reduce step, while a shift step remains the same. While the un-parse process is deterministic the time-reversal is not, because the tree is not in reality drawn, but it was shown only as a matter of course. This can be solved by introducing one or more tokens of lookahead or by restricting the class of grammars (eg. LR(0)).

Backtracking

Backtracking means that a part of the tree must be destroyed and control must be returned so that another possible solution can be tried. Exceptions can be used to backtracking. Whenever a

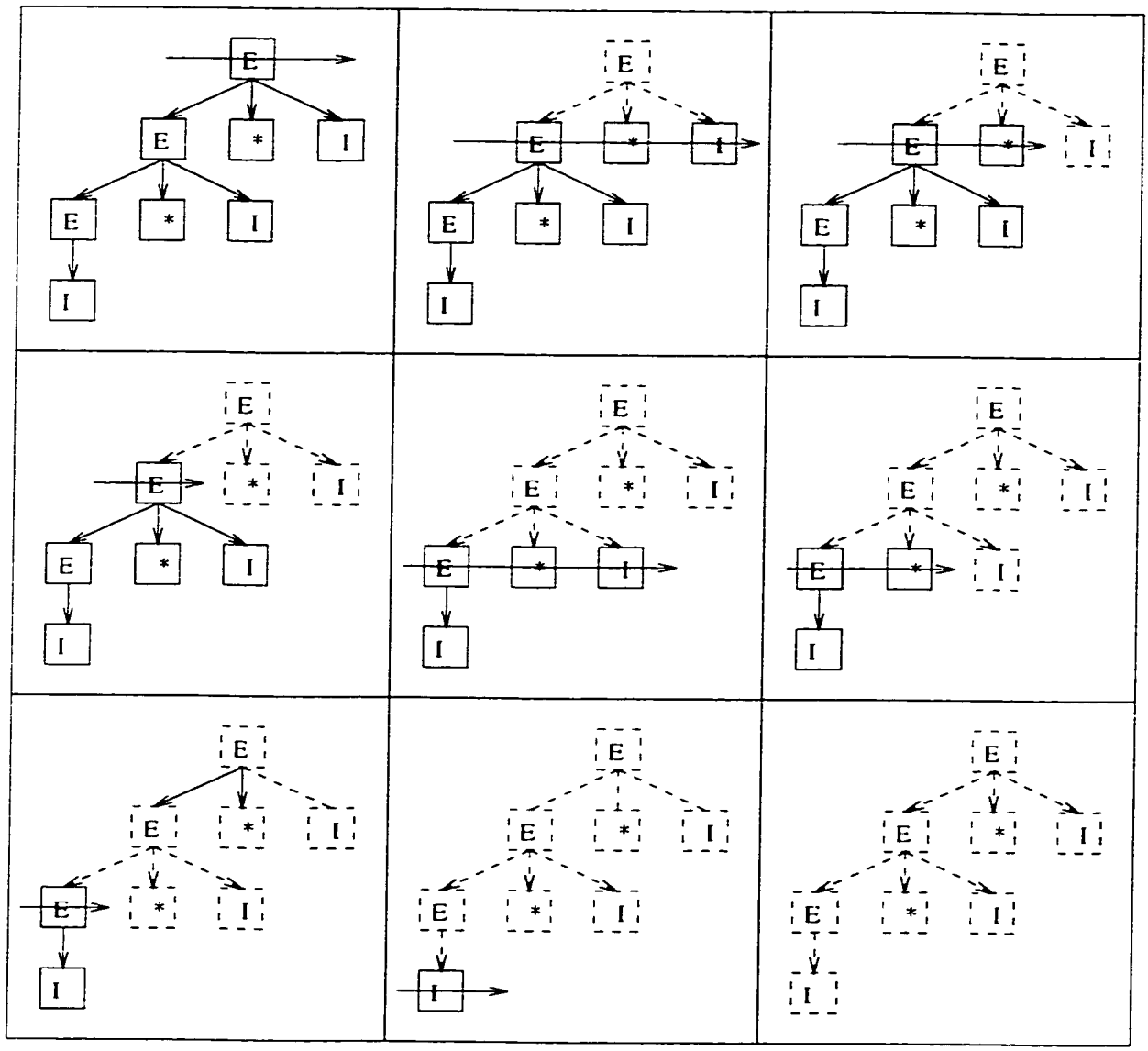


Figure 38: A step by step un-parsing of the bottom tree representing "$I * I * I$"

failure occurs an exception can be raised. An exception handler will be placed at the point where an alternative parsing algorithm is to be made. This facilitates the transfer of control to the point at which a new attempt to parsing is to be made.

3.3 Summary

This chapter defines the role of a parser generator. The famous parser generator Yacc is introduced. The chapter contains also a survey of the work done on object-oriented parsing and object-oriented parser generators by four other researchers: Johnson, Zino, Meyer, and Watson.

Chapter 4

Design of an Object Oriented Parser

4.1 Why a Recursive-Descent Parser?

The most used parsing techniques in compiler construction are LL(k) and LR(k). Since the size of a parser in both techniques increase dramatically when the value of k is incremented, k is usually restricted to 1 which eventually narrows down parsers used into two : LL(1) and LALR(1).

In many compiler construction books, recursive-descent is treated as something of limited power, or only as an introduction to learning table-driven techniques. Such a classification of recursive-descent parsers is unfounded, as we are going to show, according to the criteria given by Fisher and LeBlanc in their textbook: Crafting a Compiler [FL88]. The criteria, they put to answer the question of which algorithm is more powerful LL(1) or LALR(1)? is based on five points. A sixth point related to speed of parsers is needed and thus was added:

1. **Simplicity**: It is beyond any doubt that recursive-descent parsers are more simple and easy to both write and understand than any other parser. This may be due to the fact that top-down parsing is more natural and easy to understand than bottom-up. This is informally shown in figure 39.

2. **Generality**: While both LL(1) and LALR(1) in practice are capable of handling many programming languages definitions. LALR(1) covers a wider scope of grammars than LL(1). Also, it is harder to write a LL(1) grammar than a LALR(1), because a lot of restrictions has to be imposed on the grammar. LALR(1) has the advantage over LL(1) by this criterion.

   However, there are two reasons that make us believe there is light at the end of the tunnel for the problem of limited generality of grammars handled by a recursive-descent parser:

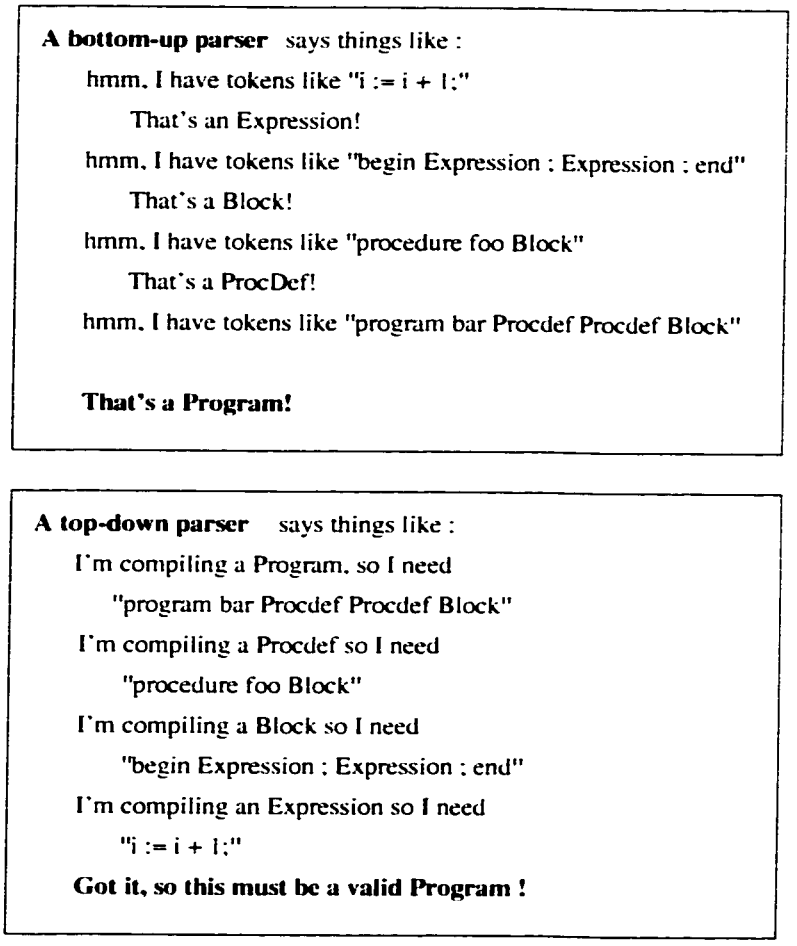


Figure 39: A witty comparison between top-down and bottom-up parsing

(a) LL(k) are grammars of about the same generality of LALR(1) for a certain $k > 1$. Thomas Christopher [Chr99] invented an algorithm that takes an LL(k) grammar and generates LL(1) tables for it. Thus, the algorithm can be used to convert an LL(k) into an LL(1) grammar. The generated grammar is usually bigger than the original because the algorithm expands every nonterminal in a production with its right hand sides. A conversion on a grammar will probably affect its readability. Further study from that point of view is needed on the algorithm.

(b) The concept or core of the algorithm for a recursive-descent parser is the same for $k = 1$ or $k > 1$. What differs however, is how lookaheads are calculated. Having calculated the lookaheads, the algorithms for a LL(1) and a LL(k) would look the same. When $k = 1$, the calculation of lookaheads is very straightforward. In contrast, when $k > 1$ the calculation becomes very complex and the size of information needed becomes extremely large. This is why LL(k) have been ignored in practice [DeR69]. Parr [Par93] showed a way to reduce space requirements for calculating lookaheads. The manifestation of this algorithm was ANTLR [PQ95] : a parser generator for LL(k) grammars. Having the possibility of using LL(k) grammars suggests using a recursive-descent parser.

3. **Action symbols**: The question here is "How flexible is the interface to semantics routines?" LL(1) can call an action routine anywhere during the recognition of a right hand side of a production. LR algorithms can only call action routines while reducing a right hand side to a left hand side. However, LALR(1) can call actions in the same places by a simple transformation of the grammar, and many LALR(1) parser generators (e.g. YACC) will do the transformation themselves. LL(1) has only a slight edge here [FL88].

4. **Error Recovery and Repair** Which parsers are better at recovering from or repairing errors in the input sentence?

(a) Error Recovery: One of the advantages of top-down parsers is that effective error recovery is easy to implement [Hol90]. LL parsers, including recursive-descent, know what symbols are expected to be recognized later in the input, while LR parsers have a stack of states they were in earlier in the parse. The information LL parsers have is much easier to use to make good repairs. LL has a distinct advantage here.

(b) Error Repair: Many authors put down, unfairly, recursive-descent when it comes to this point. Fischer and LeBlanc [FL88] for example, say "Recursive descent error repair is rarely, if ever, done". The following parser generators implement an error-correction technique: LLgen [GJ88], Ell [Gro90a]. Stirling [Sti85], Röhrich [Röh80] provide theoretical study of this topic.

5. **Parser sizes**: A recursive-descent parser is larger in size than a LALR(1). For example in a comparison between a recursive descent parser generated by LLgen and an LALR(1) parser generated by Yacc for the same grammar, it was found, when tests were done on three different machines, that the earlier has an increase of 84%, 129%, 64% the size of the later, [GJ88].

However, as memory becomes cheaper, computers have more storage space. An increase in space, if not affecting speed, is not a dig deal.

6. **Speed**: A recursive descent parser is faster than a LALR(1). A recursive descent generated by LLgen was faster than a LALR(1) parser generated by Yacc for the same grammar by 33% [GJ88].

Most important, recursive descent will permit to apply the object-oriented approach to parsing that we are going to explain in the next section.

4.2 Object-Oriented Parsing Strategy

There are standard algorithms in textbooks on compiler construction. However, these algorithms can not be used directly in an object-oriented environment because they are usually explained in the context of procedural programming, whereas we will be working with objects.

In this section we introduce our approach to building an object-oriented parser. The parser is a top-down parser for LL(1) attributed grammars written in an object oriented language.

The idea of implementing a node in a syntax tree as a record with several fields is a very well known one. The idea of representing the nodes in an annotated parse tree with objects becomes very natural when the tree is built top-down.

When top-down parsing is being used, a node of the parse tree is expanded into several other nodes. Thinking in terms of objects, an object in the tree constructs other objects. An object, which is an instance of a class, on the tree corresponds to an instance of the corresponding symbol in a program. This means that each symbol in the language corresponds to a class or an instance of a class.

For example, suppose the grammar contains the rule

$$Assignment \rightarrow Variable \quad AssOp \quad Expression$$

after parsing an expression we will have an object belonging to the class "Expression". The constructor for a class performs the parsing for the corresponding symbol. The constructor for class Assignment would call the constructors for Variable, AssOp, and Expression, and would construct an Assignment object. Thus, an Assignment object would contain pointers to classes : Variable, AssOp, and Expression.

So in order to parse a document and construct a parse tree for it, we begin by constructing the root of that tree by creating an object representing the non terminal start symbol. The constructor of the root would in turn construct the subtrees of the root.

The attributes of a symbol correspond to data members inside the class representing that symbol. If we use objects to represent nodes of the parse tree (or abstract syntax tree), then these are the natural objects to send messages to. A semantic action is a message sent to an object, and is represented by a member function inside a class.

4.3 Language Specification and Code Generated

The parsers we are describing are to be automatically generated by the parser generator we are going to introduce in the next chapter. However, for the sake of clarity and explanation, we proceed as if they are hand coded.

For specifying the languages accepted by the parser, an attribute extended BNF (EBNF) grammar will be used. Our EBNF grammar has the following main features :

- Symbols: terminals and non-terminals.
- Regular Expressions: Optional, Repetitive, and Alternated Subrules.
- Attributes and Semantic Actions

We refer to symbols, regular expressions, and semantic actions as "constructs". So any time the word construct is used it means any one of them.

4.3.1 Symbols

Terminals

We divide terminals into two categories, depending on whether data obtained from the lexical analyzer must be stored or not:

- Keywords such as Identifier, Integer, String, etc... contain data that must be stored. When we read the identifier "Foo" from a program we must store the string "Foo" somewhere. We refer to such terminals as **Terminals with information (WIT)**. The following class is used to represent WITs :

```
class WithInfoTerminal : public Construct
{
    protected :
        int TokenVal : /* Value given to this terminal and used by the lexical analyzer */
        char *lexeme : /* String corresponding to this symbol returned by the lexical
                                                    analyzer */
    public :
        WithInfoTerminal(int val , char *text)
        {
            TokenVal = val :
            lexeme = strdup(text) :
        } :
} :
```

We will explain later why WithInfoTerminal inherits Construct.

When an identifier "Foo" is encountered, an instance of class WithInfoTerminal is created.

- Operators and keywords are terminals that do not need any information to be stored other than their own name. For example, AssOp corresponds to the string "=" but we do not need to store this string because we know what an AssOp is. We refer to such terminals as **No Information Terminals (NITs)**. The following class is used to represent NITs :

```
class NoInfoTerminal : public Construct
{
    protected :
        int TokenVal :
    public :
        NoInfoTerminal(int val )
        {
            TokenVal = val;
        } :
} :
```

We will show later why NoInfoTerminal inherits Construct.

NonTerminals

It is natural that nonterminals would have common characteristics, therefore we want all nonterminals to be descendants of the same class.

```
class NonTerminal : public Construct
{
    protected :
        Scanner *scanner :
    public :
        NonTerminal(Scanner *scan)
        {
            assert(scan) :
            scanner = scan :
        } ;
} :
```

Non terminals, such as Statement, Expression, etc... , will be represented with a class inherited from class NonTerminal shown above. This class contains pointers to its components (right hand sides in the grammar).

For example consider this rule:

$$Assignment \quad ::= \quad Variable \quad AssOp \quad Expression$$

where Variable is an WIT, AssOp is a NIT, and Expression is a NT.

We said that every non terminal contains pointers to its components. Therefore, the class Assignment, contains three pointers: s0, s1, and s2 for Variable, AssOp, and Expression respectively. The constructor for Assignment which is also responsible for parsing would look like :

```
1:  Assignment::Assignment(Scanner *MyScanner) : NonTerminal(MyScanner)
2:  {
3:      char * lexeme ;
4:      if (MyScanner→Token() == _VARIABLE) /* _VARIABLE: value returned by
5:                                             the scanner upon lexing VARIABLE */
6:      {
7:          lexeme = MyScanner→Lexeme();
8:          MyScanner→match(_VARIABLE);
9:          s0 = new WithInfoTerminal(_VARIABLE, lexeme);
10:         MyScanner→match(_AssOp) ;
11:         s1 = new NoInfoTerminal(_AssOp);
12:         s2 = new Expression(MyScanner);
13:     }
14:     else
15:         _Error( MyScanner→LineNo() ) ; /* Preceded by an underscore to
16:                                           differentiate from user defined functions. */
17: }
```

The existence of AssOp is implied by the object being an Assignment (since we know its right hand side). One might say that it is not necessary for Assignment to keep a pointer to AssOp. The overhead of a pointer to AssOp is negligible especially when compared with the clarity and consistency a complete parse tree provide. A complete parse tree is not usually necessary. For example, "*if E then* S_1 *else* S_2" can be represented abstractly as IF(E, S1, S2). There are advantages in having the complete tree however, and it simplifies the task of building a parser generator (in that you do not have to decide which NITs to discard). Another advantage is that the generator can provide a function that prints the complete parse tree to verify the accuracy of the parser. Some optimization can still be achieved by having the AssOp pointing to a global object AssOp. This way, our parser will create only one instance of AssOp during the parsing process, to which all pointers to AssOp will be directed. The same applies to all NITs. This can be achieved by declaring class NITCollection.

If the grammar has four NITs : AddOp, MulOp, LP, and RP. The declaration of class NITCollection would look like this:

```
class NITCollection
{
    private :
```

```
        static NoInfoTerminal * symbol[4] :
    public :
        static void IniSymbols() :
        static NoInfoTerminal *GetSymbol(int):
};

NoInfoTerminal * NITCollection::symbol[4]:

void NITCollection :: IniSymbols()
{
    symbol[0] = new NoInfoTerminal(_AddOp):
    symbol[1] = new NoInfoTerminal(_MulOp):
    symbol[2] = new NoInfoTerminal(_LP):
    symbol[3] = new NoInfoTerminal(_RP):
}

NoInfoTerminal * NITCollection :: GetSymbol(int val)
{
    if ((val >= _AddOp) && (val <= _RP))
        return symbol[val - _AddOp]:
    else
    {
        cout << "Unknown value for a NoInfoTerminal \n " :
        exit(0):
    }
}
```

Thus, line 11 in the constructor of Assignment becomes:

```
s1 = NITCollection::GetSymbol(_AssOp) :
```

Things become more complicated when the non terminal has two or more alternate rules. Consider the following rule:

Statement → *Variable* *AssOp* *Expression*
| ''*if*'' *Expression* ''*then*'' *Statement* ''*else*'' *Statement*

As we have said earlier the class representing "Statement" should contain pointers to its components:

```
class Statement : public NonTerminal
{
```

```
private :
    WIT * a :      // Variable
    NIT * b :      // AssOp
    NT * c :       // Expression
    WIT * d :      // "if"
    NT * e :       // Expression
    WIT * f :      // "then"
    NT * g :       // Statement
    WIT * h :      // "else"
    NT * i :       // Statement
    ...
}
```

The readability, and clarity of the program will decrease especially if the non terminal has many productions, because it becomes very difficult to keep track of which pointers belong to which alternate rule.

NT, NIT, and WIT are all constructs within a grammar. If we define an abstract class Construct, NT, NIT, and WIT will be heirs of class Construct. Figure 40 shows the relation between the created classes. By defining *s* as a pointer to Construct we can benefit from polymorphism and allow *s* to accept pointers to NT, NIT, and WIT. This means that there is no need for a class of a non terminal to contain a number of pointers that is equal to the total sum of symbols in each alternate rule, but rather it is sufficient to contain a number of pointers that is equal to the maximum number of symbols from any of its alternate rules.

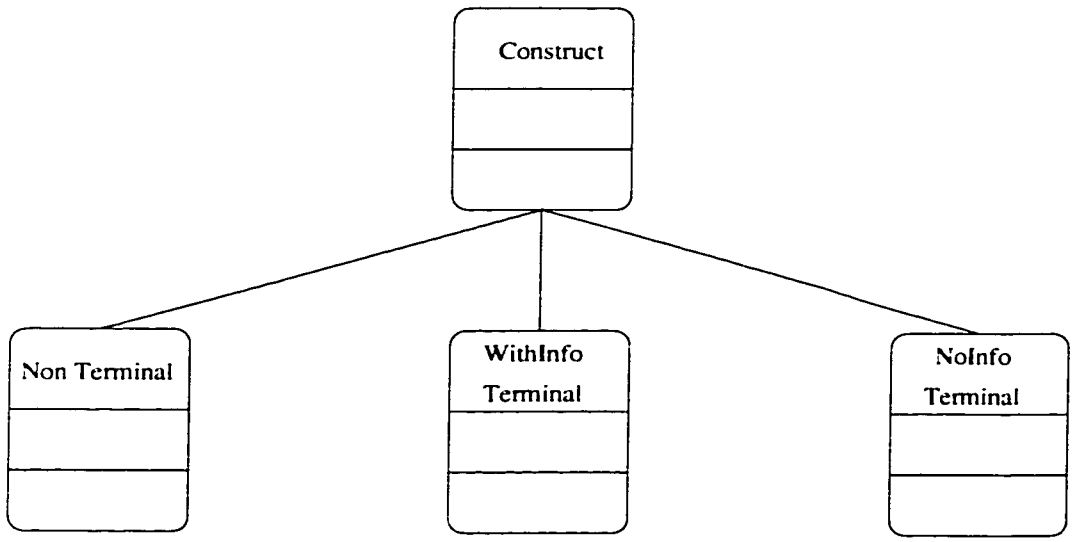


Figure 40: Grammar symbols inherit **Construct**

Thus the class declaration of Statement would look like this:

```
class Statement : public NonTerminal
{
    private :
        Construct * s0 ;
        Construct * s1 ;
        Construct * s2 ;
        Construct * s3 ;
        Construct * s4 ;
        Construct * s5 ;
    public :
        ...
}
```

The first alternate rule of Statement, if chosen, will use only the first three pointers: s0, s1, and s2, while the rest of the pointers will be set to null. The second alternate rule is in need of all the six pointers.

The constructor of Statement will look like this :

```
Statement::Statement(Scanner *MyScanner) : NonTerminal(MyScanner)
{
    char * lexeme ;
    s0 = NULL ;
    s1 = NULL ;
    s2 = NULL ;
    s3 = NULL ;
    s4 = NULL ;
    s5 = NULL ;
    if ((MyScanner→Token() == _Variable) )
    {
        Expression * tmp2 ;
        lexeme = MyScanner→Lexeme();
        MyScanner→match(_Variable);
        s0 = new WithInfoTerminal(_Variable, lexeme);
        MyScanner→match(_AssOp) ;
        s1 = NITCollection::GetSymbol(_AssOp);
        tmp2 = new Expression(MyScanner) ;
        s2 = tmp2 ;
    }
    else
    if ((MyScanner→Token() == _IF) )
    {
```

```
            Expression * tmp1 ;
            Statement * tmp3 ;
            Statement * tmp5 ;
            MyScanner→match(_IF) ;
            s0 = NITCollection::GetSymbol(_IF);
            tmp1 = new Expression(MyScanner) ;
            s1 = tmp1 ;
            MyScanner→match(_THEN) ;
            s2 = NITCollection::GetSymbol(_THEN);
            tmp3 = new Statement(MyScanner) ;
            s3 = tmp3 ;
            MyScanner→match(_ELSE) ;
            s4 = NITCollection::GetSymbol(_ELSE);
            tmp5 = new Statement(MyScanner) ;
            s5 = tmp5 ;
        }
        else
            _Error( MyScanner→LineNo() ) ;
    }
```

If the nonterminal has more than one alternative rule and one of them derives the null string we treat it as if it was placed as the last one.

However, even with this approach, we can not say that the generated constructor will be very readable. The number of data members in "Statement" is not large now. But, for a language with may kinds of statements, the variables are going to become very overloaded. Also the number of "tmp_x" generated will make the constructor less readable. It would be better to create subclasses for every production of "Statement". Subclasses "Assignment" and "Conditional" would be created for the first and second productions of "Statement" respectively. The parser generator can not invent such nice names, but rather it will generate: "Statement_1", "Statement_2". Obviously, subclasses Statement_1 and Statement_2 inherit class Statement. All the attributes of the parent class Statement, even its inherited ones, need to be mirrored into the subclasses generated: Statement_1 and Statement_2. This is very important to guarantee that the semantic values gathered during the parsing and belonging to Statement will not be lost by introducing the new subclasses. The parser generator generates such a member function, for every subclass, that copies attributes' values of the parent class to the generated subclass.

The class for Statement contains only one data member "s0" that will be eventually pointing to either "Statement_1" or "Statement_2". The class declaration for class Statement will be :

```
class Statement : public virtual NonTerminal
{
```

```
    protected :
        Construct * s0 ;
    public :
        Statement(Scanner *);
        static void _Error(int line = -1);
} ;
```

while that of Statement_1 which represents the assignment production:

```
class Statement_1 : public virtual NonTerminal, public Statement
{
    protected :
        WithInfoTerminal * s0 ;
        NoInfoTerminal * s1 ;
        Expression * s2 ;
    public :
        Statement_1(Scanner *);
        void InitialiseInheritedAttrib(Statement *);
        static void _Error(int line = -1);
} ;
```

and that of Statement_2 which represents the conditional production:

```
class Statement_2 : public virtual NonTerminal, public Statement
{
    protected :
        NoInfoTerminal * s0 ;
        Expression * s1 ;
        NoInfoTerminal * s2 ;
        Statement * s3 ;
        NoInfoTerminal * s4 ;
        Statement * s5 ;
    public :
        Statement_2(Scanner *);
        void InitialiseInheritedAttrib(Statement *);
        static void _Error(int line = -1);
} ;
```

The constructor of Statement_1 is almost the same as that of "Assignment" on page 66. The constructor of Statement_2 looks like this:

```
Statement_2::Statement_2(Scanner *MyScanner) : NonTerminal(MyScanner),
```

```
                                        Statement(MyScanner)
{
    char * lexeme ;
    s0 = NULL ;
    s1 = NULL ;
    s2 = NULL ;
    s3 = NULL ;
    s4 = NULL ;
    s5 = NULL ;
    if ((MyScanner→Token() == _IF))
    {
        MyScanner→match(_IF) ;
        s0 = NITCollection::GetSymbol(_IF);
        s1 = new Expression(MyScanner) ;
        MyScanner→match(_THEN) ;
        s2 = NITCollection::GetSymbol(_THEN);
        s3 = new Statement(MyScanner) ;
        MyScanner→match(_ELSE) ;
        s4 = NITCollection::GetSymbol(_ELSE);
        s5 = new Statement(MyScanner) ;
        return ;
    }
    _Error( MyScanner→LineNo() ) ;
}
```

The constructor for class Statement is less complicated than the one shown before :

```
Statement::Statement(Scanner *MyScanner) : NonTerminal(MyScanner)
{
    char * lexeme ;
    s0 = NULL ;
    if ((MyScanner→Token() == _Variable))
    {
        Statement_1 * tmp0 = new Statement_1(MyScanner);
        tmp0→InitialiseInheritedAttrib(this);
        s0 = tmp0 ;
        return ;
    }
    if ((MyScanner→Token() == _IF))
    {
        Statement_2 * tmp0 = new Statement_2(MyScanner);
```

```
            tmp0→InitialiseInheritedAttrib(this);
            s0 = tmp0 ;
            return ;
        }
        Error( MyScanner→LineNo() ) ;
    }
```

It is true that using this strategy more classes will be generated, but each class will be easier to understand because it would be handling a simple problem. Therefore we adopted the latter approach.

Using this approach, we have to decide on when to generate a new sub-class and when not. A subclass should not be generated for a production that contains only a Terminal, or a non terminal, accompanied by zero or many actions. For example, imagine that "Statement" has a third production containing a nonterminal "FunctionCall". "s0" of "statement" can be used to represent "FunctionCall" and there is no need to generate a subclass.

The following criteria is used to decide when to create or not to create a subclass :

- Do not create a subclass if:
  1. The non terminal has only one production on the RHS
  2. The non terminal has two productions on the RHS but one of them contains only actions or just the null string
  3. The non terminal has two or more productions each of them containing only one nonterminal or a terminal accompanied by any number of actions, or an optional containing only one nonterminal or a terminal accompanied by any number of actions
- Create a subclass if:
  1. The non terminal has more than one production and one of them at least has:
     (a) multiple symbols or an optional containing multiple symbols or
     (b) contains a repetition or an alternation or an optional containing a repetition or an alternation .

4.3.2 Regular Expressions

Optional Subrules

A square bracket [ ] pair is used to delimit a sequence of one more constructs as optional subrule.

When a Optional is encountered, it just means:

```
if the current token is within the FIRST set of this Optional
    parse all the symbols and execute all semantic actions inside this Optional
```

else

all the symbols inside this Optional should be set zero.

For example, consider the non terminal "IFStmt" defined by this production:

IfStmt ::= ''*if*'' *Condition* ''*then*'' *Stmt* ''[' ''*else*'' *Stmt* '']''

The class of "IfStmt" will contain pointers to all the elements inside the Optional. As we notice below, s5 and s6 are the pointers to "else" and Stmt contained inside the Optional. It can be written as :

```
IFStmt::IFStmt(Scanner *MyScanner) : NonTerminal(MyScanner)
{
    char * lexeme ;
    if ((MyScanner→Token() == _IF) )
    {
        MyScanner→match(_IF) ;
        s0 = NITCollection::GetSymbol(_IF);
        s1 = new Condition(MyScanner) ;
        MyScanner→match(_THEN) ;
        s2 = NITCollection::GetSymbol(_THEN);
        s3 = new Stmt(MyScanner) ;
        if ((MyScanner→Token() == _ELSE) )
        {
            MyScanner→match(_ELSE) ;
            s4 = NITCollection::GetSymbol(_ELSE);
            s5 = new Stmt(MyScanner) ;
        }
        else
        {
            s4 = 0 ;
            s5 = 0 ;
        }
    }
    else
        _Error( MyScanner→LineNo() ) ;
}
```

Repetitive Subrules

A pair of brackets followed by a star [ ]* is used to delimit a repetitive sequence (with zero or more occurrences). The sequence can be of any number of constructs.

Consider the following production :

Expression → *Term* [AddOp Term]* :

An easy approach to parse this production is:

```
BinaryOp = NULL :
create an instance of Term :
while (next_token == AddOp)
{
    match(AddOp) :
    Left = An instance of AddOp :
    Right = An instance of Term :
    BinaryOp = make_bin(Left, Right, BinaryOp) :
}
connect BinaryOp to the parse tree.
```

Where make_bin() constructs a binary operator node, pointed to by BinaryOp, in the parse tree. BinaryOp can be represented by a class as shown below, hence make_bin would be the constructor of that class.

```
class BinaryOp : public Construct
{
    private :
        Construct * left , * right :
        BinaryOp * previous :
    public :
        BinaryOp(Construct *l = 0, Construct *r = 0, BinaryOp *p = 0)
        {
            left = l :
            right = r :
            previous = p :
        };
};
```

Talking C++, the constructor of Expression would look like this:

```
Expression::Expression(Scanner *MyScanner) : NonTerminal(MyScanner)
{
    char * lexeme :
    if ((MyScanner→Token() == _ID) )
    {
        s0 = new Term(MyScanner) :
        if ((MyScanner→Token() == _AddOp))
        {
            BinaryOp * bin_op = 0 ;
```

```
            NoInfoTerminal * left = 0 ;
            Term * right = 0 ;
            while ((MyScanner→Token() == _AddOp) )
            {
                MyScanner→match(_AddOp);
                left = NITCollection::GetSymbol(_AddOp);
                right = new Term(MyScanner);
                bin_op = new BinaryOp(left , right , bin_op);
            }
            s1 = bin_op ;
        }
        else
            s1 = NULL ;
    }
    else
        _Error( MyScanner→LineNo() ) ;
}
```

Figure 41 shows the parse tree obtained according to our algorithm. It is noted in the figure that the last created BinaryOp is the one connected to the tree, and not the first. We need to have the ability to build the parse tree from the the input, and inversely restructure the input file from the parse tree. This not difficult to do.

One of the characteristics of this algorithm is that it isolates what is inside the brackets "[ ]*" from the rest of the parsing algorithm by emphasizing one variable, the one that will be connected to rest of the parse tree. Thus we can easily figure out how to parse

$$`` \quad [\quad ID \quad [\quad AddOp \quad ID \quad]* \quad] \quad ''$$

The BinaryOp shown above has only two pointers, left and right, hence it allows a maximum of two symbols inside the repetition. It was used only as a mean of explanation. We need to develop BinaryOp to allow using sequences of whatever sizes. For example, if the repetition subrule contains n non terminals: $\alpha_1\alpha_2\cdots\alpha_n$, binaryOp should look like this:

```
class BinaryOp : public Construct
{
    private :
        Construct **array;
        int array_size ;
        BinaryOp * previous ;
    public :
        BinaryOp(Construct ** arr = 0, int size = 0 , BinaryOp *p=0)
        {
            array = arr ;
```

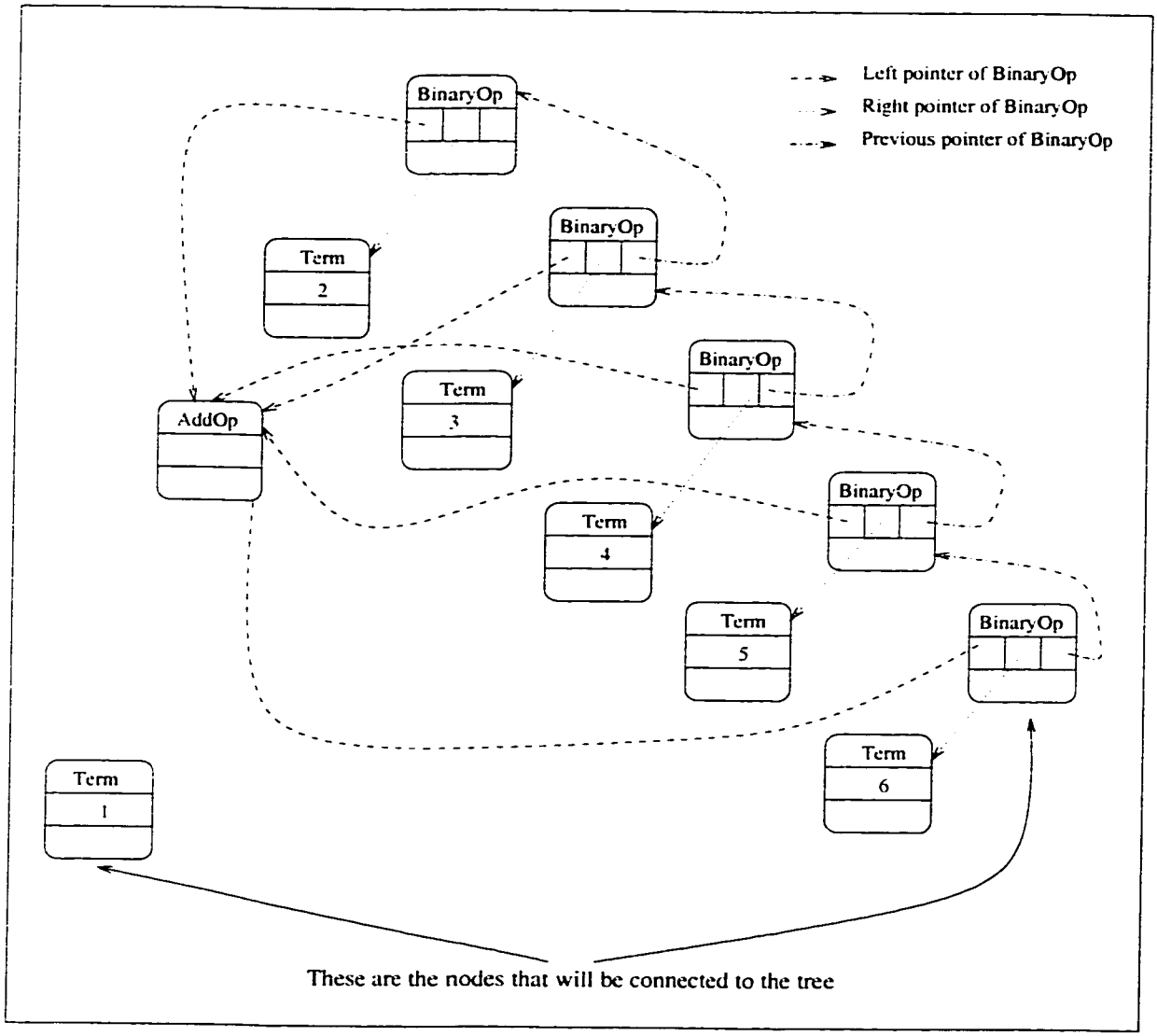


Figure 41: The use of **BinaryOp** to parse a repetitive subrule

```
            array_size = size ;
            previous = p ;
        } ;
} ;
```

Instead of having two pointers, BinaryOp has an array of pointers. The array will be created dynamically. The algorithm below shows us how this is done for this repetition: [*MulOp AddOp U*]* . where U is a nonterminal:

```
BinaryOp * bin_op = 0 ;
while ((MyScanner→Token() == _MulOp))
{
    MyScanner→match(_MulOp);
    NoInfoTerminal * tmp0 = NTCollection::GetSymbol(_MulOp);
    MyScanner→match(_AddOp);
    NoInfoTerminal * tmp1 = NTCollection::GetSymbol(_AddOp);
    U * tmp2 = new U(MyScanner);
    Construct ** array = new (Construct * ) [3];
    array[ 0 ] = tmp_rep_0;
    array[ 1 ] = tmp_rep_1;
    array[ 2 ] = tmp_rep_2;
    bin_op = new BinaryOp(array, 3, bin_op);
}
s_r = bin_op ;
```

Alternated Subrules

A pair of parenthesis ( ) is used to indicate a choice between two or more alternatives, separated from each other by | . An alternative can be a symbol (terminal or non terminal) accompanied by any number of semantic actions. Only the symbol in the last alternative can be the null symbol. This means that only one pointer is needed to represent an alternated subrule. Since the pointer can be pointing to a terminal or a non terminal, it better be a pointer to Construct.

The code corresponding to an alternated subrule ($\alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \cdots \mid \alpha_n$) varies based on three conditions as follows:

1. None of the symbols inside the alternated subrule is or derives the null string

```
if the current token is within the FIRST set of α1
    parse the symbol and execute all semantic actions inside α1
else
if the current token is within the FIRST set of α2
    parse the symbol and execute all semantic actions inside α2
...
```

...

else

Error

2. The symbol in the last alternated subrule is the null string

if the current token is within the FIRST set of α_1

parse the symbol and execute all semantic actions inside α_1

else

if the current token is within the FIRST set of α_2

parse the symbol and execute all semantic actions inside α_2

...

...

else

set the pointer representing this subrule to null.

3. One of the symbol inside an alternate α_i, not necessarily the last alternate, derives the null string:

if the current token is within the FIRST set of α_1

parse the symbol and execute all semantic actions inside α_1

else

if the current token is within the FIRST set of α_2

parse the symbol and execute all semantic actions inside α_2

else

...

else

if the current token is within the FIRST set of α_{i-1}

parse the symbol and execute all semantic actions inside α_{i-1}

else

if the current token is within the FIRST set of α_{i+1}

parse the symbol and execute all semantic actions inside α_{i+1}

else

...

if the current token is within the FIRST set of α_n

parse the symbol and execute all semantic actions inside α_n

else

parse the symbol and execute all semantic actions inside α_i

4.3.3 Attributes and Semantic Actions

Attributes

Terminals other than keywords, i.e. WITs and non terminals can have zero or many attributes. For example in the production:

$$Expression \rightarrow Expression \quad AddOp \quad Term$$

Expression, and Term both need to have an attribute "val", that will be used to hold the semantical values obtained by applying a semantic rule such as :

$$Expression.val = Expression.val + Term.val$$

The attributes of a symbol correspond to data members inside the class representing that symbol. For example, to give Expression an attribute val that is an integer, we define an integer data member val in the class declaration of Expression:

```
class Expression : public virtual NonTerminal
{
    protected :
        int Expression_val ;
        ...
    ...
};
```

When an attribute is given to an WIT, it becomes necessary to define a new class with the name of that WIT. The new class inherits class WithInfoTerm. For example, if we want to give WIT "ID" an attribute val then a new class has to be defined for ID :

```
class ID : public WithInfoTerminal
{
    protected :
        int ID_val;
        ...
} ;
```

We will explain later (when we speak about inheritance), why we declared the attributes as protected and not as private.

One of the problems is how to handle attributes' types. I just assumed that the values were ints.

Even though the attributes used so far are of type integer, the user has the ability to declare different types of attributes. In the next chapter we will show how attributes' types can be specified by the user.

Actions

Since the nodes of a parse tree are represented as objects, then these are natural objects to send messages to. Each semantic action should be a message sent to an object. A message is represented with a function call whose arguments are attributes of grammar symbols corresponding to the subtrees of the node.

Below is an example of a grammar rule and its associated semantic action written in Yacc :

$$Expr \;:\; Term \;\; '+' \;\; Expr \;\; \{ \;\; \$\$ \;=\; \$1 \;+\; \$3 \;\; : \;\; \}$$

In Yacc, the semantic action is written in C code in the grammar. This is language dependent and can be verbose (although you can put everything in the form of function calls). For us, the semantic action should be a message written in as much general form as possible like "$do\_it()$". The Yacc rule can be translated using our approach to:

$$Expr \;:\; Term \;\; '+' \;\; Expr \;\; \{ \;\; this \rightarrow add(Expr.val, Term.val) \;\; : \;\; \}$$

The use of "this" coincides with the C++ "this". We merely intended by it a pointer to the current node. "add" is a member function of Expr and thus its header function has to appear inside the class declaration of Expr :

```
int Expr_add(int , int ) ;
```

A function call of "add" has to appear also inside the constructor of Expr.

We have to pay attention how to link the arguments of the function inside the semantic action with the pointers to different symbols inside the class. For example, the arguments $Expr.val$ and $Term.val$ correspond to $s0 \rightarrow var$ and $s1 \rightarrow var$ respectively. Thus, the constructor of Expr, assuming the grammar is LL(1), would look like :

```
1:  Expr::Expr(Scanner *MyScanner) : NonTerminal(MyScanner)
2:  {
3:      char * lexeme ;
4:      Expr_val = 0 ;
5:      if ((MyScanner→Token() == _ID) )
6:      {
7:          s0 = new Term(MyScanner) ;
8:          MyScanner→match(_AddOp) ;
9:          s1 = NITCollection::GetSymbol(_AddOp);
10:         s2 = new Expr(MyScanner) ;
11:         Expr_add(s0?s0→_Get_Term_val(): 0, s2?s2→_Get_Expr_val(): 0);
12:     }
13:     else
14:         _Error( MyScanner→LineNo() ) ;
15: }
```

On line 11 of this algorithm, the function call for the semantic action, we notice that the name of the function "add" was preceded by the class name "Expr", this is explained below in the subsection about inheritance. We also notice that arguments of the function were represented by functions _Get_Term_val() and _Get_Expr_val(). These functions are automatically generated member functions of classes Term and Expr respectively. They are needed because attributes are usually private data members and can not be accessed directly by other classes.

The user is responsible only for coding the definition of the function add. The definitions will be in a separate file so they do not have to be changed when the grammar is changed. The parser generator can however facilitate the job of the user by generating a skeleton of every function in a separate file. For example, for "add" the following skeleton can be generated.

```
int Expr::Expr_add(int a1, int a2)
{
}
```

This skeleton can serve as a reminder for the user of how many functions have to be defined, what are the return types, and how many arguments they take. It is enough for the user to write between the two bracelets of this skeleton:

```
return Expr_val = a1 + a2 ;
```

everything else will be generated by the parser generator.

The only problem in the notation we are using to specify semantic actions occurs when the same name of a symbol is used more than once in the rule. For example :

$$E \;:\; T \;\; AddOp \;\; T \;\; \{ \;\; ConvertType(T.type, ``int''); \;\; add(T.val, T.val); \;\; \};$$

There is no way in this example to tell which "T" of the two we are indicating. Yacc uses $\$i$ to denote the i'th symbol in the rule, and replaces this by an array reference in the generated code. It would be better if the user could write something a bit less error-prone than $\$i$. Our parser generator will be more natural, and more advantageous to the user.

By default, the first occurrence of T in the production will be considered. However, in order to overwrite this default, this notation has to be used:

$$E \;:\; T \;\; AddOp \;\; T \;\; \{ \;\; ConvertType(T:2.type, ``int''); \;\; add(T.val, T:2.val); \;\; \};$$

The colon followed by a number i indicates the ith occurrence of the symbol in the production. If no colon and number were specified, then by default, the first occurrence of the symbol is being referred to. When counting the number of occurrences of a symbol, we ignore the existence of [ ], [ ]* , and () within a production.

As for such an Expression :

$$E \;\rightarrow\; T \;\; ``+'' \;\; E$$

In this production we notice the existence of an "E" on the right hand side and another on the left hand side of the production. We can access the attributes of E on the left hand side of the

production with the "this" pointer which is defaulted. A colon followed by a number is only used for symbols occurring on the right hand side of the production.

There are no restrictions on where an action can occur on the right hand side of a production. However there is a restriction on their order of evaluation. The order of evaluation of these rules is L-attribute.

If an action is being called by a symbol that occurs to its right the action will be ignored. the same thing applies to attributes. If the attribute belongs to a symbol that occurs to the right of the action. the attribute will be replaced by NULL by the parser generator.

An action can not reference a symbol inside a repetition or an alternation unless the action is within the same repetition or alternation.

Inheritance

Sometimes after the user defines the productions. attributes. semantic actions in his grammar he decides to experiment by adding some rules. attributes. semantic actions on some symbols. This is dangerous because it could mess up his entire grammar. and these modifications would be difficult to remove if later found unneeded.

Let us assume that A is the non terminal we want to experiment on. We define a new non terminal B with all the new productions. attributes. and semantic actions we want to add. If we add a production of the following form to our grammar:

$$A \rightarrow B$$

then it means an object A on the parse tree can give birth to a new object B. It would be very natural for B to inherit all attributes and functions defined on A since it is a direct substitute for it. This can be done by having class B inherit A. Therefore. whenever a production of the above mentioned form is encountered. its class declaration would contain in addition to what have been said so far:

```
class B : public virtual NonTerminal, public A
{
    ...
};
```

If there is another production:

$$C \rightarrow B$$

in the grammar. then B would inherit C too. Thus the class of B would look like

```
class B : public virtual NonTerminal, public A, public C
{
    ...
};
```

Both A. and C inherit class NonTerminal. This explains why the word virtual precedes NonTerminal in the class declaration above.

The definition of the constructor of B should reflect this inheritance as C++ requires:

```
B::B(Scanner *MyScanner) : NonTerminal(MyScanner), A(MyScanner), C(MyScanner)
{
}
```

B would still inherit A even if there were semantic actions on the right hand side of the production with B.

Because we wanted to avoid conflicts between the inherited and inheritor classes we have preceded the names of attributes and semantic functions with the class name such as in Expr_var and Term_var. However, when the user writes semantic actions in the grammar specification he does not have to worry about adding the name of the class to the name of the function or the attribute inherited. For example if $func()$ is a member function of A and B inherits A then the user can write something like:

$$B \;:\; \alpha_1 \;\cdots\; \alpha_n \quad \{this.func();\};$$

4.4 Interfacing with a Lexical Analyzer

Lexing is done using our Scanner class. This class wraps up "flex". Flex is Gnu's version of lex, the lexical analyzer generator. It generates code that is compatible with C++. Scanner provides the interface for flex with the rest of the system, hence flex is invisible to the rest of the program and can be replaced easily.

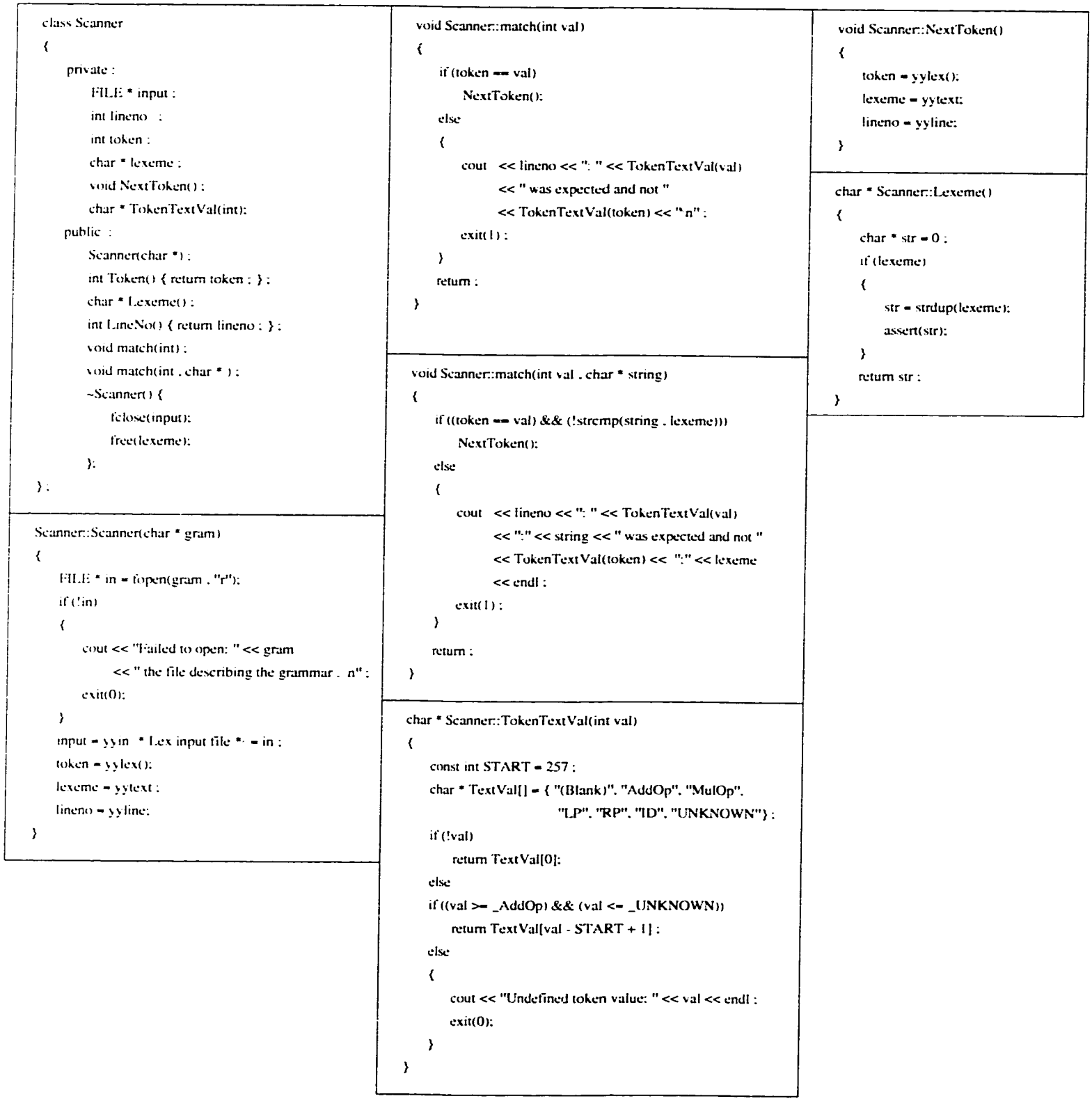
Figure 42 shows the implementation of the Scanner class.

4.5 Problems and Solutions

We can improve our parser generator by:

1. Including error repair/recovery functions in our generated parser. Right now the parser will stop as soon as an error occurs, prompting the user with a message indicating the error and on which line to be found. We expect that including error repair/recovery functions would pose no problems. In Section 4.1 we introduced some reference about this subject.
2. Allowing it to accept a single quoted character such as '+' and treat it as a NIT.
3. Allowing it to accept literal strings such "Hello World" and treat it as WIT
4. Allowing the user to write comments in his grammar specification
5. Allowing repetitive subrules to be of a specific number times and not only 0 or many.

The last three suggestions are more easy to implement than the first one which requires more effort. These suggestions are by no means the only possible ones, other suggestions can be found.



```
class Scanner
{
    private :
        FILE * input ;
        int lineno   ;
        int token ;
        char * lexeme ;
        void NextToken() ;
        char * TokenTextVal(int);
    public  :
        Scanner(char *) ;
        int Token() { return token ; } ;
        char * Lexeme() ;
        int LineNo() { return lineno ; } ;
        void match(int) ;
        void match(int , char * ) ;
        ~Scanner() {
            fclose(input);
            free(lexeme);
        };
} ;
```

```
Scanner::Scanner(char * gram)
{
    FILE * in = fopen(gram , "r");
    if (!in)
    {
        cout << "Failed to open: " << gram
             << " the file describing the grammar .  n" ;
        exit(0);
    }
    input = yyin  * Lex input file *- = in ;
    token = yylex();
    lexeme = yytext ;
    lineno = yyline;
}
```

```
void Scanner::match(int val)
{
    if (token == val)
        NextToken();
    else
    {
        cout  << lineno << ": " << TokenTextVal(val)
              << " was expected and not "
              << TokenTextVal(token) << "n" ;
        exit(1) ;
    }
    return ;
}
```

```
void Scanner::match(int val , char * string)
{
    if ((token == val) && (!strcmp(string , lexeme)))
        NextToken();
    else
    {
        cout  << lineno << ": " << TokenTextVal(val)
              << ":" << string << " was expected and not "
              << TokenTextVal(token) <<  ":" << lexeme
              << endl ;
        exit(1) ;
    }
    return ;
}
```

```
char * Scanner::TokenTextVal(int val)
{
    const int START = 257 ;
    char * TextVal[] = { "(Blank)", "AddOp", "MulOp",
                         "LP", "RP", "ID", "UNKNOWN"} ;
    if (!val)
        return TextVal[0];
    else
    if ((val >= _AddOp) && (val <= _UNKNOWN))
        return TextVal[val - START + 1] ;
    else
    {
        cout << "Undefined token value: " << val << endl ;
        exit(0);
    }
}
```

```
void Scanner::NextToken()
{
    token = yylex();
    lexeme = yytext;
    lineno = yyline;
}
```

```
char * Scanner::Lexeme()
{
    char * str = 0 ;
    if (lexeme)
    {
        str = strdup(lexeme);
        assert(str);
    }
    return str ;
}
```

Figure 42: Class **Scanner** implementation.

4.6 Summary

This chapter contains a comparison that proves wrong the idea that recursive-descent parsing is of limited value. It also details our object-oriented strategy to parsing. Language specification and the code generated are explained in details. Many examples are given about the use of terminals, nonterminals, regular expressions, attributes and semantic actions. The interface of the parser with the lexical analyzer is shown. Some suggestions to improve the parser are given.

Chapter 5

Design of a Parser Generator

5.1 Approach to Building a Parser Generator

Figure 43 is an example of an input file showing the format that we are using to specify grammars we want our parser generator to generate parsers for.

Grammar G, shown in figure 44, specifies the set of all input files acceptable by our parser generator.

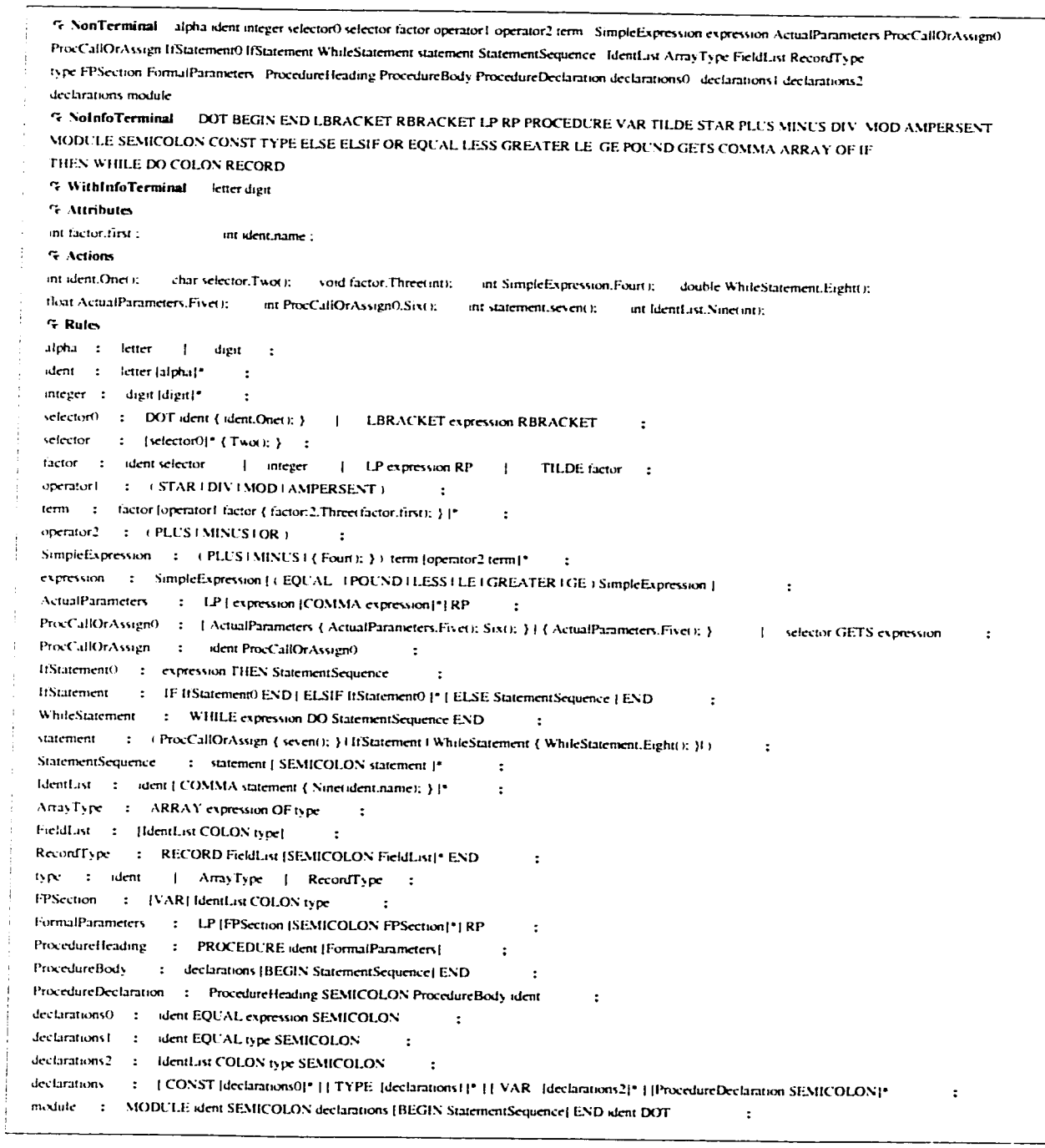
When the user specifies his grammar according to the format specified by G, it becomes very simple for us to point out non terminals, keywords, terminals, grammar rules, regular expressions (optionals, repetitions, and alternatives), attributes, and semantic actions.

Our approach to creating a parser generator is very simple. We represent every construct of the afore mentioned constructs with a class. Every class contains one or more member functions to generate code about itself that constitutes part of the parser to be generated. When the user runs the parser generator program on his input file, an instance of these classes will be created for every symbol in his grammar. By calling the code generating member functions of these objects a parser for the user's grammar will be generated.

5.2 Grammar Constructs

Figure 45 shows the classes that have to be created for different grammar constructs. It also shows some of its most important data members. In this figure we can point out the following classes:

- **Construct:** An abstract class. Classes Terminal, ActionInfo, WithInfoTerm, NonTerminal, Optional, Repetition, and Alternation inherit class Construct. Construct indicates these classes' common properties such as:
  - name



```
NonTerminal  alpha ident integer selector0 selector factor operator1 operator2 term  SimpleExpression expression ActualParameters ProcCallOrAssign0
ProcCallOrAssign IfStatement0 IfStatement WhileStatement statement StatementSequence  IdentList ArrayType FieldList RecordType
type FPSection FormalParameters  ProcedureHeading ProcedureBody ProcedureDeclaration declarations0  declarations1 declarations2
declarations module
NoInfoTerminal  DOT BEGIN END LBRACKET RBRACKET LP RP PROCEDURE VAR TILDE STAR PLUS MINUS DIV  MOD AMPERSENT
MODULE SEMICOLON CONST TYPE ELSE ELSIF OR EQUAL LESS GREATER LE  GE POUND GETS COMMA ARRAY OF IF
THEN WHILE DO COLON RECORD
WithInfoTerminal  letter digit
Attributes
int factor.first ;  int ident.name ;
Actions
int ident.One(): char selector.Two(): void factor.Three(int): int SimpleExpression.Four(): double WhileStatement.Eight():
float ActualParameters.Five(): int ProcCallOrAssign0.Six(): int statement.seven(): int IdentList.Nine(int):
Rules
alpha : letter | digit ;
ident : letter {alpha}* ;
integer : digit {digit}* ;
selector0 : DOT ident ( ident.One(): ) | LBRACKET expression RBRACKET ;
selector : {selector0}* ( Two(): ) ;
factor : ident selector | integer | LP expression RP | TILDE factor ;
operator1 : ( STAR | DIV | MOD | AMPERSENT ) ;
term : factor [operator1 factor ( factor2.Three(factor.first): ) ]* ;
operator2 : ( PLUS | MINUS | OR ) ;
SimpleExpression : ( PLUS | MINUS | ( Four(): ) ) term {operator2 term}* ;
expression : SimpleExpression [ ( EQUAL | POUND | LESS | LE | GREATER | GE ) SimpleExpression ] ;
ActualParameters : LP [ expression {COMMA expression}* ] RP ;
ProcCallOrAssign0 : [ ActualParameters ( ActualParameters.Five(): Six(): ) ] ( ActualParameters.Five(): ) | selector GETS expression ;
ProcCallOrAssign : ident ProcCallOrAssign0 ;
IfStatement0 : expression THEN StatementSequence ;
IfStatement : IF IfStatement0 END | ELSIF IfStatement0 ]* [ ELSE StatementSequence ] END ;
WhileStatement : WHILE expression DO StatementSequence END ;
statement : ( ProcCallOrAssign ( seven(): ) | IfStatement | WhileStatement ( WhileStatement.Eight(): ) ) ;
StatementSequence : statement { SEMICOLON statement }* ;
IdentList : ident [ COMMA statement ( Nine(ident.name): ) ]* ;
ArrayType : ARRAY expression OF type ;
FieldList : [IdentList COLON type] ;
RecordType : RECORD FieldList {SEMICOLON FieldList}* END ;
type : ident | ArrayType | RecordType ;
FPSection : [VAR] IdentList COLON type ;
FormalParameters : LP [FPSection {SEMICOLON FPSection}*] RP ;
ProcedureHeading : PROCEDURE ident [FormalParameters] ;
ProcedureBody : declarations {BEGIN StatementSequence} END ;
ProcedureDeclaration : ProcedureHeading SEMICOLON ProcedureBody ident ;
declarations0 : ident EQUAL expression SEMICOLON ;
declarations1 : ident EQUAL type SEMICOLON ;
declarations2 : IdentList COLON type SEMICOLON ;
declarations : [ CONST {declarations0}* ] [ TYPE {declarations1}* ] [ VAR {declarations2}* ] {ProcedureDeclaration SEMICOLON}* ;
module : MODULE ident SEMICOLON declarations [BEGIN StatementSequence] END ident DOT ;
```

Figure 43: Example of an input file to our parser generator

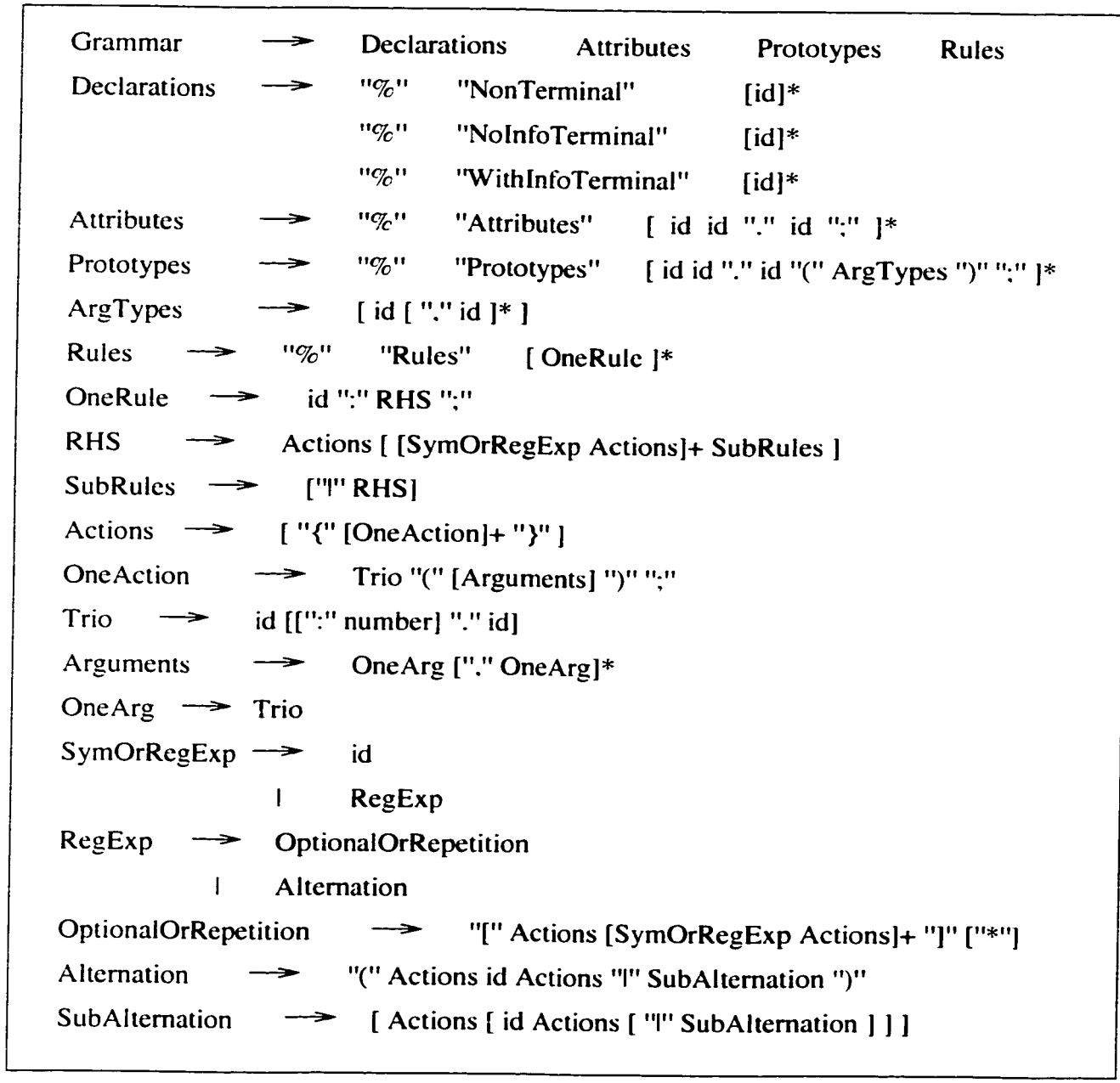


Figure 44: Grammar "G" specifying the set of input files acceptable by our parser generator

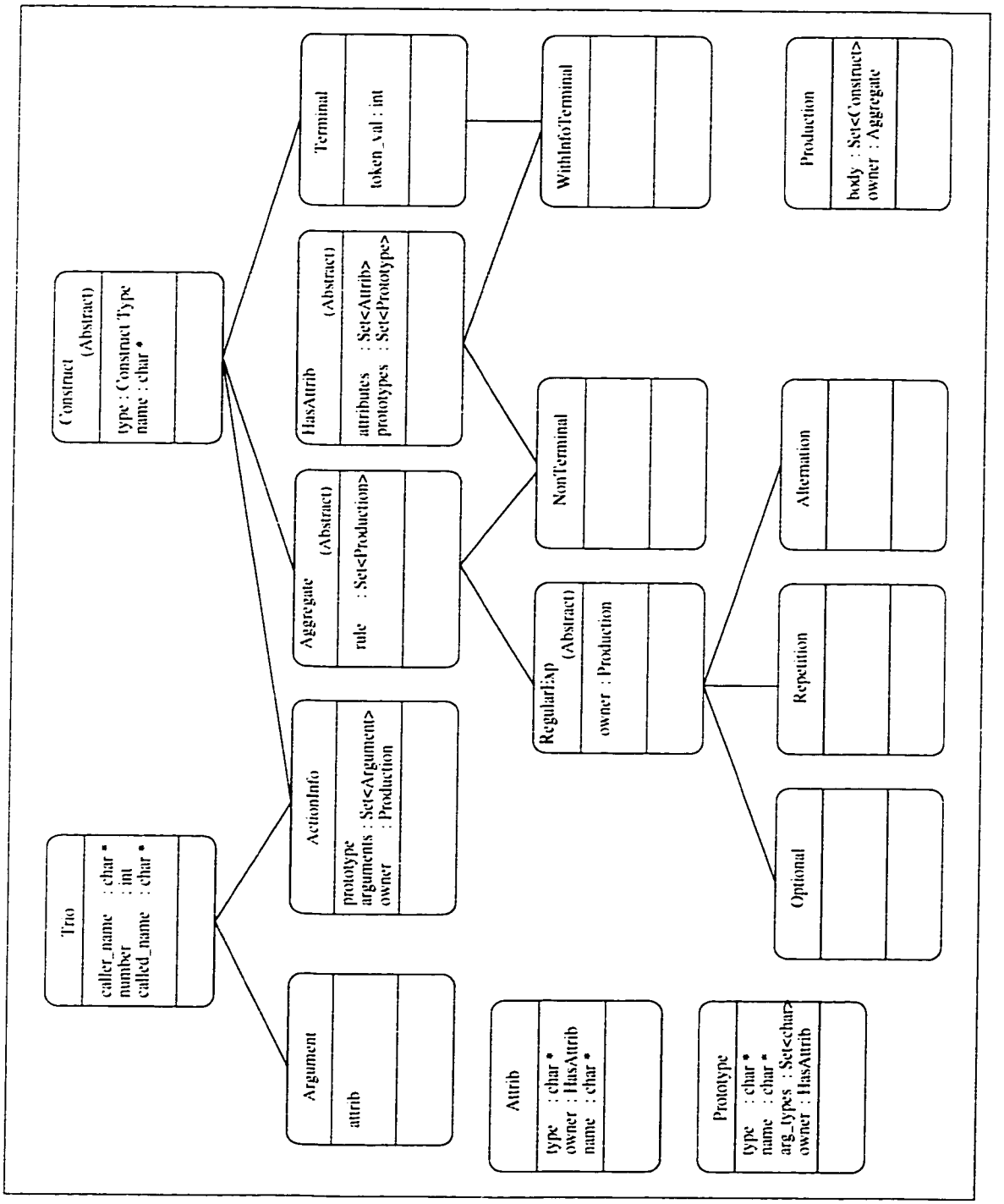


Figure 45: Classes representing constructs that can be found in a user's input grammar file.

- type: this attribute specifies whether the construct is a NonTerminal, Terminal, WithInfoTerm, ActionInfo, Optional, Repetition, or Alternation.

- **Aggregate:** An abstract class. Classes Nonterminal, Optional, Repetition, and Alternation are all aggregates. Aggregates are meaningless if not explained by means of productions. Therefore, an "Aggregate" has a field "rule" which is the list of all productions defining it.

- **Terminal:** This class describes all terminals. To every terminal must be assigned a value that the lexical analyzer makes use of. Therefore, "Terminal" contains the field token_val.

- **HasAttrib:** An abstract class. Both non terminals and terminals (only WITs) can have attributes and actions defined on them. Therefore two data members are necessary:

  - attributes: The set of attributes defined on this aggregate. The attributes are found in the section labelled "% Attributes" in the user's input file. Attributes are represented with class "Attribute".

  - prototypes: The prototypes of actions defined on this aggregate. These prototypes are found in the section labeled "% Actions" in the user's input file. Prototypes are represented with class "Prototype".

- **WithInfoTerminal:** Represents all terminals that have information that need to be stored (as described in chapter 4). Such a terminal can have attributes, and actions. In that case a special class is created for that terminal that inherit class WithInfoTerminal.

- **NonTerminal**

- **ActionInfo**

- **RegularExp:** An abstract class. Optional, Repetition, and Alternation are regular expressions. The only place regular expressions (nested or not) can appear are productions. Therefore, we can say that regular expressions are owned by productions. The "owner" field indicates which production owns this regular expression. "RegularExp" is an heir of Aggregate, and therefore inherits "rule".

- **Optional**

- **Repetition**

- **Alternation**

- **Production:** Is a way of representing the set of symbols defining an Aggregate. Therefore, "Production" has a field "body" which is a list of "Constructs". Every "Production" is owned by an "Aggregate" and keeps track of that owner in the field "owner".

- **Attrib:** Represents the attributes defined in the attribute declaration section of the user's input file. The following data members are needed to describe an attribute:

  - type: a string that saves the type of the attributes, for example an integer, a float, ...

- name
- owner: a pointer to "HasAttrib". Indicates the owner of this attribute that could be either a NonTerminal, or a WithInfoTerminal.

- **Prototype:** Represents the actions, which are functions, defined in the actions declaration section of the user's input file.
  - type: a string that saves the return type of the prototype, for example an integer, a float, ...
  - name
  - arg_types: a list of strings each of which is the type of one argument of this function
  - owner: a pointer to "HasAttrib". Indicates the owner of this prototype that could be either a NonTerminal, or a WithInfoTerminal.

- **Trio:** a class that is inherited by classes "Arguments" and "ActionInfo". When an argument or a function call is encountered in an input grammar file, their names are indicated in the following form: $id_1[[$ `` : ''$number]$ `` . ''$id_2]$ as shown in figure 44 in rules "OneArg" and "OneAction". Trio holds these tree pieces of information id_1, *number*, and id_2 in caller_name, number, and called_name respectively.

- **ActionInfo:** Describes the function calls of semantic actions found within grammar rules. It specifies what are the arguments and who is the caller.
  - prototype: a pointer to the prototype of this function call.
  - arguments: the list of arguments of this function call. Every element in the list is a pointer to an object "Argument".
  - owner: a pointer to the non terminal or the WIT owning this function.

- **Argument:** describes the argument of a function call that appears inside a grammar rule. It inherits "Trio". The arguments of a function call must be attributes of a non terminal or a WIT. Therefore, "Argument" contains a data member "attrib" which is a pointer to the "Attrib" this argument is referring to.

5.3 Phases of Parser Generation

The entire process of parser generation will be launched by object "ParserGen" (Parser generator). Figure 46 shows the phases of parser generation.

5.3.1 Scanning

Scanning is done using our Scanner class. This class wraps up "flex". Flex is Gnu's version of lex, the lexical analyzer generator. It generates code that is compatible with C++. Scanner provides

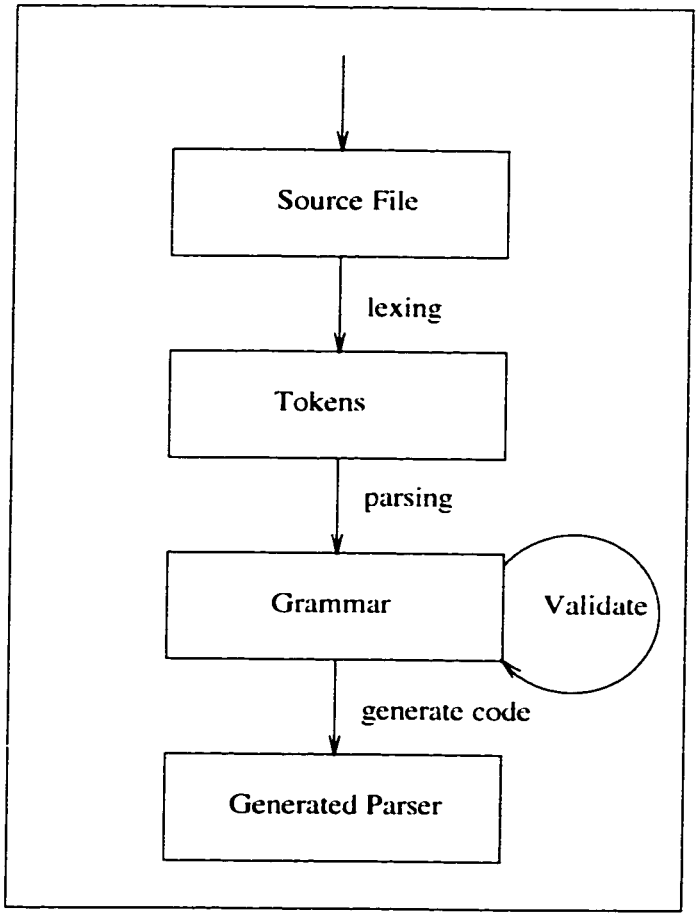


Figure 46: Phases of parser generation

the interface for flex with the rest of the system. hence flex is invisible to the rest of the program and can be replaced easily.

5.3.2 Parsing

We have said earlier that we need to represent every construct of the user's input grammar with a class. Having grammar G at hand. it is possible to write a parser that parses the user's input file and create an object for every matched construct.

Class Grammar will act as a container and will hold pointers to all non terminals. keywords. and terminals. All other constructs. attributes and actions for example. are in no danger of getting lost. since they can be accessed from within terminals and non terminals. The class for grammar would look like this:

```
class Grammar
{
  private :
    Set<NonTerminal> * NTs  ;
    Set<Terminal>    * NITs ;
    Set<WithInfoTerm> * WITs ;
    ....
};
```

One way to implement this parser is to create a class "Parser" that contains a member function for every rule in grammar G shown in figure 44. Figure 47 shows class Parser's declaration. We are not interested in showing how to parse grammar G. since it is an LL(1) grammar. and it is very clear how to write a parser for it. We are rather interested in how to transform the user's input grammar file into a data structures of classes. This task can be achieved by adding semantic actions into grammar G. Figure 48 shows grammar G after adding semantic actions into it. Actions are printed in bold letters and enclosed within {}. The best way to understand figure 48 is to consider it rule by rule and explain it with its actions.

- **Declarations:** By this rule. the user declares all the non terminals. and terminals (keywords or non-keywords) that he will be using in his grammar.
  - **action1:** Creates instances of NT symbols.

    Construct *symbol*
    if $(\neg\exists\ symbol : symbol \in Grammar$ **and** $symbol\cdot\text{name} == < id_1 \cdot lexeme >)$
    $symbol =$ **new** NonTerminal(...) :
    $Grammar \cdot NTs = Grammar \cdot NTs + symbol$;
    /* **else**

```
class Parser
{
    private :
        Scanner * scanner ;
        Grammar * grammar ;
    public :
        void ParseDeclarations();
        void ParseAttributes();
        void ParsePrototypes();
        Set<char> * ParseArgTypes();
        void ParseRules();
        void ParseOneRule() ;
        void ParseRHS(NonTerminal *);
        void SubRules(NonTerminal *);
        Production * ParseActions(NonTerminal *, Aggregate *, Production * prod = 0);
        Production * ParseOneAction(NonTerminal *, Production *);
        Trio * ParseTrio(NonTerminal *, HasAttrib * &);
        Set<Argument> * ParseArguments(NonTerminal * s);
        Argument * ParseOneArg(NonTerminal *);
        Production * ParseID(Aggregate * , Production * prod = 0);
        Production * ParseSymOrRegExp (NonTerminal *, Aggregate * , Production * prod = 0) ;
        Production * ParseRegExp (NonTerminal *, Aggregate * , Production * , int) ;
        RegularExp * ParseOptionalOrRep(NonTerminal *);
        RULE * ParseAlternation(NonTerminal *, Alternation *) ;
        RULE * ParseSubAlternation(NonTerminal *, Alternation *) ;
        Parser (char * gram) ;
        Grammar * ParseGrammar () ;
        . . .
        ~Parser () ;
};
```

Figure 47: Class **Parser** declaration.

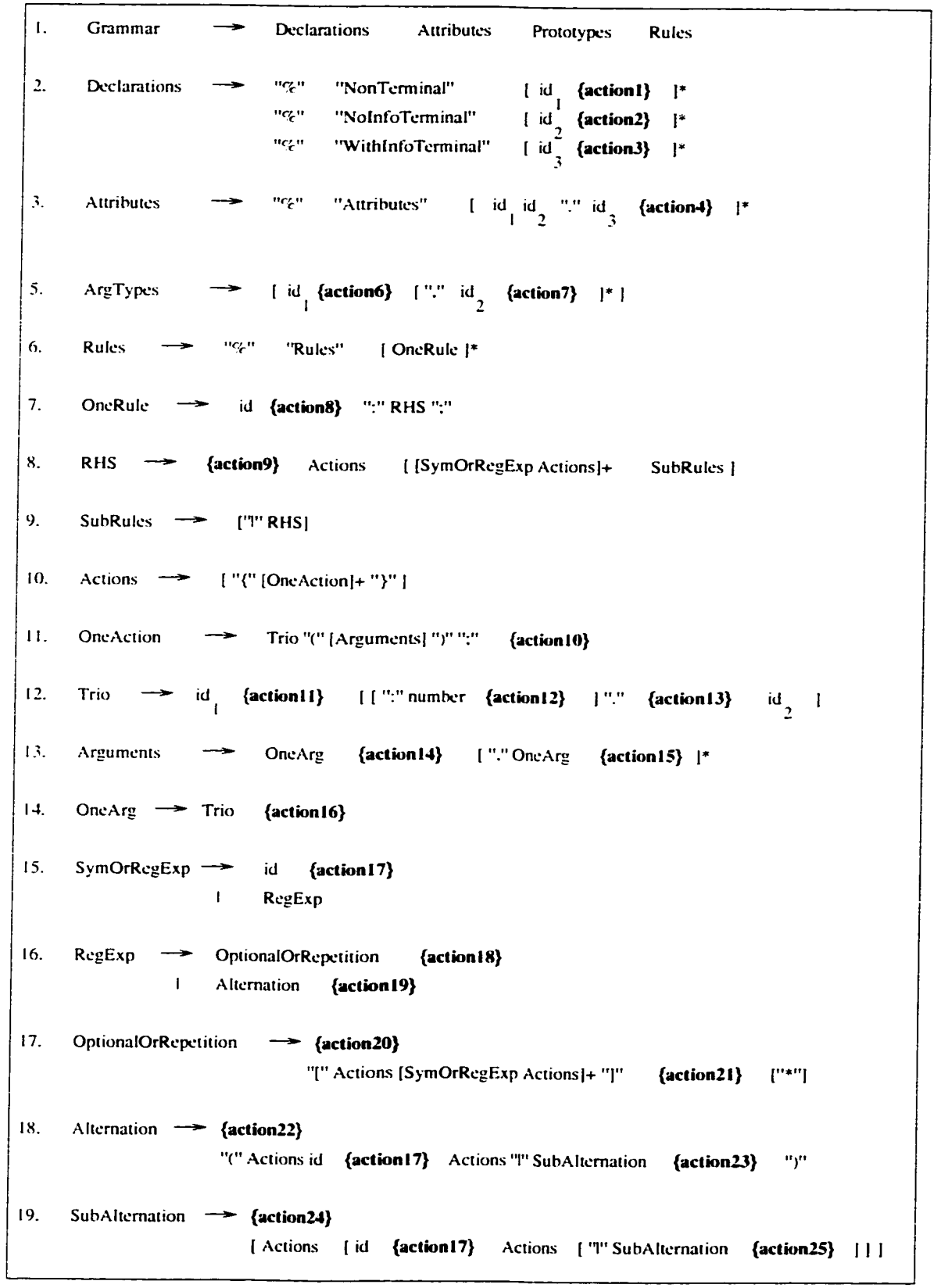


Figure 48: Grammar G after adding semantics actions to it.

A symbol with this name was already declared in the grammar as an NT
*We assume it was typed again as an error */*

- **action2:** Creates instances of NIT symbols.

  Construct *symbol*
  if ($\neg\exists$ *symbol* : *symbol* $\in$ *Grammar* **and** *symbol*·name == $< id_2 \cdot lexeme >$)
  symbol = **new** Terminal(...)
  $Grammar \cdot NITs = Grammar \cdot NITs + symbol$:
  else
  if (*symbol* **is not** "NIT")
  Error
  /* **else**
  A symbol with this name was already declared in the grammar as a NIT
  *We assume it was typed again as an error */*

- **action3:** Creates instances of WIT symbols.

  Construct *symbol*
  if ($\neg\exists$ *symbol* : *symbol* $\in$ *Grammar* **and** *symbol*·name == $< id_3 \cdot lexeme >$)
  symbol = **new** WithInfoTerm(...)
  $Grammar \cdot WITs = Grammar \cdot WITs + symbol$
  else
  if (*symbol* **is not** "WIT")
  Error
  /* **else**
  A symbol with this name was already declared in the grammar as a WIT
  *We assume it was typed again as an error */*

- **Attributes:** By this rule, the user declares the attributes of NTs, and WITs (non-keywords terminals) if they have any.

  - **action4:** Creates instances of every attribute.

    Construct *symbol*
    if ($\exists$ *symbol* : *symbol* $\in$ *Grammar* **and** *symbol*·name == $< id_2 \cdot lexeme >$)
    if (*symbol* **is not** "WIT" **nor** "NT")
    Error */* make sure that symbol can not have attributes */*
    else
    Error
    if ($\exists$ *symbol* : *symbol* $\in$ *Grammar* **and** *symbol*·name == $< id_3 \cdot lexeme >$)
    Error
    Attrib *attrib* := **new** Attrib(...)
    / Make sure symbol does not already have an attribute with the*
    *same name and of a different type. */*

if ($\exists$ $attrib_2$: $attrib_2 \in symbol \cdot attributes$ **and** $attrib_2$·name == $< id_3 \cdot lexeme >$)
if ($attrib_2$·type <> $attrib$·type)
Error
else /* *since it is of the same type, assume it is just an error* */
return
/* *No such attribute has been defined in this symbol. Add it to this symbol* */
$symbol \cdot attributes = symbol \cdot attributes + attrib$:

- **Prototypes:** By this rule, the user declares the prototypes of semantic actions defined on NTs and WITs (non-keywords terminals) if they have any.
  - **action5:** Creates instances of every prototype. A prototype collects information about action return type, action name, and arguments types.

Construct *symbol* :
if ($\exists$ *symbol* : $symbol \in Grammar$ **and** *symbol*·name == $< id_2 \cdot lexeme >$)
if (*symbol* **is not** "WIT" **nor** "NT")
Error
else
Error
/* *Check if this name is a valid function name. A valid function name can not be: "this", the name of a symbol, the name of an attribute of symbol* */
if ($< id_3.lexeme >$ == "**this**")
Error
if ($\exists$ *symbol* : $symbol \in Grammar$ **and** *symbol*·name == $< id_3 \cdot lexeme >$)
Error
if ($\exists$ *attrib* : $attrib \in symbol \cdot attributes$ **and** *attrib*·name == $< id_3 \cdot lexeme >$)
Error
/* *Let arg_types be the value returned by the function for parsing non terminal "ArgTypes" that occurs to the left of action5. arg_types is a set of strings representing arguments' types* */
Prototype *prototype* := **new** Prototype(... , *arg_types*)
/* *check if the symbol given has already such a function if it does not add it to the list of functions of this symbol. To compare two prototypes, compare their types, their names, then their arguments' types one by one* */
if ($prototype \notin symbol \cdot prototypes$)
$symbol \cdot prototypes = symbol \cdot prototypes + prototype$

- **ArgTypes:** This rule is used to specify the types of a prototype's arguments. The function corresponding to this rule will return a set of strings each of which is the type of an argument.
  - **action6:** Creates a set of strings containing the type of the first argument.

Set<char> *arg_types* := **new** Set($< id_1 \cdot lexeme >$)

- **action7:** Appends the rest of the arguments' types into the set created in action6.

  $arg\_types = arg\_types + < id_2 \cdot lexeme >$

- **OneRule:** Describes a rule in the user's grammar.

  - **action8:** Make sure there is a non terminal with this name in the grammar.

    Construct *symbol* :
    if ($\exists$ *symbol* : *symbol* $\in$ *Grammar* **and** *symbol*-name == $< id \cdot lexeme >$)
        if (*symbol* **is not** "NT")
            Error
    else
        Error

- **RHS:** By this rule the user defines the right hand side of a non terminal symbol. The function for parsing RHS has as an argument a NonTerminal variable. The only place from which this function is called is within the function for parsing OneRule. Therefore, the function for parsing RHS takes as an argument the non terminal *symbol*, which RHS is supposed to define its right hand side, represented by id_1 in the right hand side of OneRule. Check action8 to see how *symbol* was obtained.

  - **action9:**

    if (lookahead $\notin$ {''{'', *ID*,''['', ''('' }) /* *there is nothing on the right hand side* */
        symbol-nullable = 1 :
        return
    else
        Production *prod* = **new** Production(*symbol*) /* *symbol owns prod* */
        symbol-rule = *symbol*-rule + *prod*

- **OneAction:** The function for parsing OneAction takes as an argument a variable of type Production. Everytime an Action is constructed by this function it is added to the body of that production. For example when the function for parsing RHS calls the function for parsing Actions, the function for parsing OneAction is eventually called, and hence the value of the Production variable passed to it will be the production constructed in action9.

  - **action10**

    /* *The name of the function is obtained from the object trio returned by the function for parsing Trio the non terminal that occurs to the left of action10.* */
    if ($trio \cdot Called\_name$ ==" *this*")
        Error /* *Not a valid function name* */
    if ($\exists$ *symbol* : *symbol* $\in$ *grammar* **and** $symbol \cdot name == trio \cdot Called\_name$)
        Error /* *Not a valid function name* */

if ($\exists$ *attrib* : *attrib* $\in$ *sym* · *attributes* **and** *attrib* · *name* == *trio* · *Called_name*)
Error /* *Not a valid function name* */
ActionInfo info = **new** ActionInfo(...)
prod · *body* = *prod* · *body* + *info* :

- **Trio:** This object is used to specify an attribute or a function that will be used in an action call. The information needed for this specification is the name of the attribute or function, the name of the owner of the attribute or function, and the i^{th} occurrence of that owner in the production. One of the arguments of the function for parsing Trio is *symbol* which is the non terminal symbol created in action8 and that during the course of defining its right hand side the function for parsing Trio was called.

  - **action11:** Create an instance of object Trio.

    int *number* := 1
    /* < id_1.*lexeme* > *contains the function or attribute name. The owner is symbol* */
    if (*lookahead* $\notin$ {'' : '' , '' . ''})
    Trio *trio* := **new** Trio(*symbol*· name, 0 /* *number* */, < id_1 · *lexeme* >, ...)
    return *trio*

  - **action12:**

    if (*number* $\leq$ 0)
    Error /* *invalid specification of occurrence of symbol in production* */
    if (*str*1 == "**this**")
    Error /* *a number is specified when "this" is used as the symbol name.* */
    number = < *number* · *val* >

  - **action13:** Create an instance of object Trio.

    if (< id_1.*lexeme* > == "this")
    Trio *trio* = **new** Trio(*symbol* · *name* , 0 /* *number* */ , < id_2 · *lexeme* > , ...)
    return *trio*
    else
    if ($\exists$ *symbol* : *symbol* $\in$ *Grammar* **and** *symbol*·name == < id_1 · *lexeme* >)
    if (*symbol*) **is not** "NT" **nor** "WIT")
    Error
    else
    Error
    Trio *trio* = **new** Trio(< id_1.*lexeme* >, *number*, < id_2 · *lexeme* > , ...)
    return *trio*

- **Arguments:** The arguments of a function call are represented each with an object of type Argument. This rule groups these objects within one set.

- **action14:** The function for parsing non terminal OneArg occurring to the left of action14 returns an object of type Argument "arg". Pick this object and create a set that contains it.

  /* *arg is the object returned by the function for parsing non terminal* $OneArg_1$ */
  Set<Argument> arg_lst = **new** Set < Argument >(arg)

- **action15:** Appends the objects created by the function for parsing non terminal *OneArg* into the set of Argument created in action14.

  $arg\_lst = arg\_lst + arg$;

- **OneArg:** Creates an object of type argument that will be returned by the function for parsing it.

  - **action16:** Based on the information stored in *trio*, that is returned by the function for parsing non terminal Trio that occurs on the left of action16, an object of type Argument is created. But the data stored in trio has to be verified if valid for an attribute name.

    if ($trio \cdot called\_name$ == "**this**")
      Error /* *not a valid attribute name* */
    if ($\exists$ $symbol$: $symbol \in Grammar$ **and** $symbol \cdot name == < trio \cdot called\_name >$)
      Error /* *not a valid attribute name* */
    Argument arg = **new** Argument($trio$- caller_name, $\cdots$)

- **SymOrRegExp:** Occurs in two rules: RHS and OptionalOrRepetition. The context in which SymOrRegExp occurs, is description of the body of a production *prod*. In RHS prod is built by action9, while in OptionalOrRepetition *prod* is built by action20 as will be shown below.

  - **action17:** Upon encountering an identifier, we have to make sure there is a symbol with such a name in the grammar. The symbol which is either an NT, NIT, or a WIT has to be added to the body of the production *prod*.

    if ($\exists$ $symbol$: $symbol \in Grammar$ **and** $symbol \cdot name == < id \cdot lexeme >$)
      $prod \cdot body = prod \cdot body + sym$
    else
      Error

- **RegExp:** Occurs in rule SymOrRegExp.

  - **action18:** The function for parsing OptionalOrRepetition will return an object that is either of type Optional or Repetition. Both classes inherit form the **RegExp** class. The object returned by RegExp has to be added to *prod* that SymOrRegExp is describing as seen earlier.

    $prod \cdot body = prod \cdot body + RegSym$;

- **action19:** Creates an Alternation whose body is the set of productions returned by the function for parsing Alternation. We append the object created into the body of the production *prod* that was constructed in action9 and that represents the rhs of the rule we are trying to parse. *prod* is passed to the function for parsing RegExp as an argument.

  ```
  /* rule: Set< Production > returned by the function for parsing Alternation. */
  Alternation Alternate = new Alternation(rule)
  prod · body = prod · body + Alternate
  ```

- **OptionalOrRepetition:** Create an instance of either an object of type Optional or Repetition. The created object will be returned by the function for parsing this non terminal.

  - **action20:** Create a new production to represent what is inside the optional or repetition to be parsed.

    ```
    Production prod = new Production(...)
    ```

  - **action21:** Creates either an Optional or a Repetition depending on whether the bracket is followed by a star or not. Surely, the body of the created object is represented by the production created in action20.

    ```
    RegularExp symbol
    if (lookahead = "*")
        symbol = new Repetition(...)
    else
        symbol = new Optional(...)
    ```

- **Alternation:** An alternating subrule "(.. | ..)" has at least two alternatives. We represent every alternative with a production. Thus we need to create and return a set of Production.

  - **action22:** Create a production that will represent the first alternative of the Alternation.

    ```
    Production prod = new Production(...)
    ```

  - **action23:** The function for parsing SubAlternation will return a set of production called *ProdList* that represent second to last alternatives in the body of the alternation. Append the production constructed in action22 to *ProdList*.

    ```
    ProdList = ProdList + prod /* prod is the production created in action22 */
    ```

- **SubAlternation:** The function for parsing SubAlternation returns a set production representing each one of the alternatives. This function takes as an argument a variable of type Alternation which is the alternation, created in action19, whose body is being parsed.

- **action24**

  if (*lookahead* $\notin$ { ID . "{" }
  alternate · *nullable* = 1 :
  return
  else
  Production *Prod* = **new** Production(*alternate* /*owner*/);

- **action25:**

  /* *Prod: in action24. ProdList: returned by function for parsing SubAlternation* */
  Alternate · *rule* = *Alternate* · *rule* + *Prod* + *ProdList*

Figures 49, 50, 51, and 52 are examples of the data structures obtained after parsing rules selector0, operator1, term, and ProcedureBody of figure 43.

5.3.3 Validating the Grammar

It is a good idea to check if the grammar at hand is LL(1) or not before generating it.

In their "Dragon Book", Aho, Ullman and Sethi [ASU86] say that a nonrecursive grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$, and α and β are two distinct productions of G the following conditions hold:

1. FIRST(α) $\cap$ FIRST(β) = ϕ .
2. At most one of α and β can derive the empty strings.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then FIRST($\alpha$) $\cap$ FOLLOW(β) = ϕ.

In our case, since our grammars contain regular expressions: [ ], [ ]*, and ($\cdots \mid \cdots$), the three conditions above still have to apply on regular expressions but with slight modifications :

1. In an alternation no two of its productions can start with the same symbol.
2. In an alternation at most one of its productions can derive the empty string.
3. For optionals, repetitions, and alternations that contain one nullable productions: their FIRST sets their FOLLOW sets must be disjoint.

If we implement these conditions we can make sure no ambiguous grammars can be parsed. Algorithm 1 shows how to apply these conditions. However, the algorithm shows only how condition 2 is applied. The rest of conditions are shown in algorithm 2 for calculating the FIRST sets.

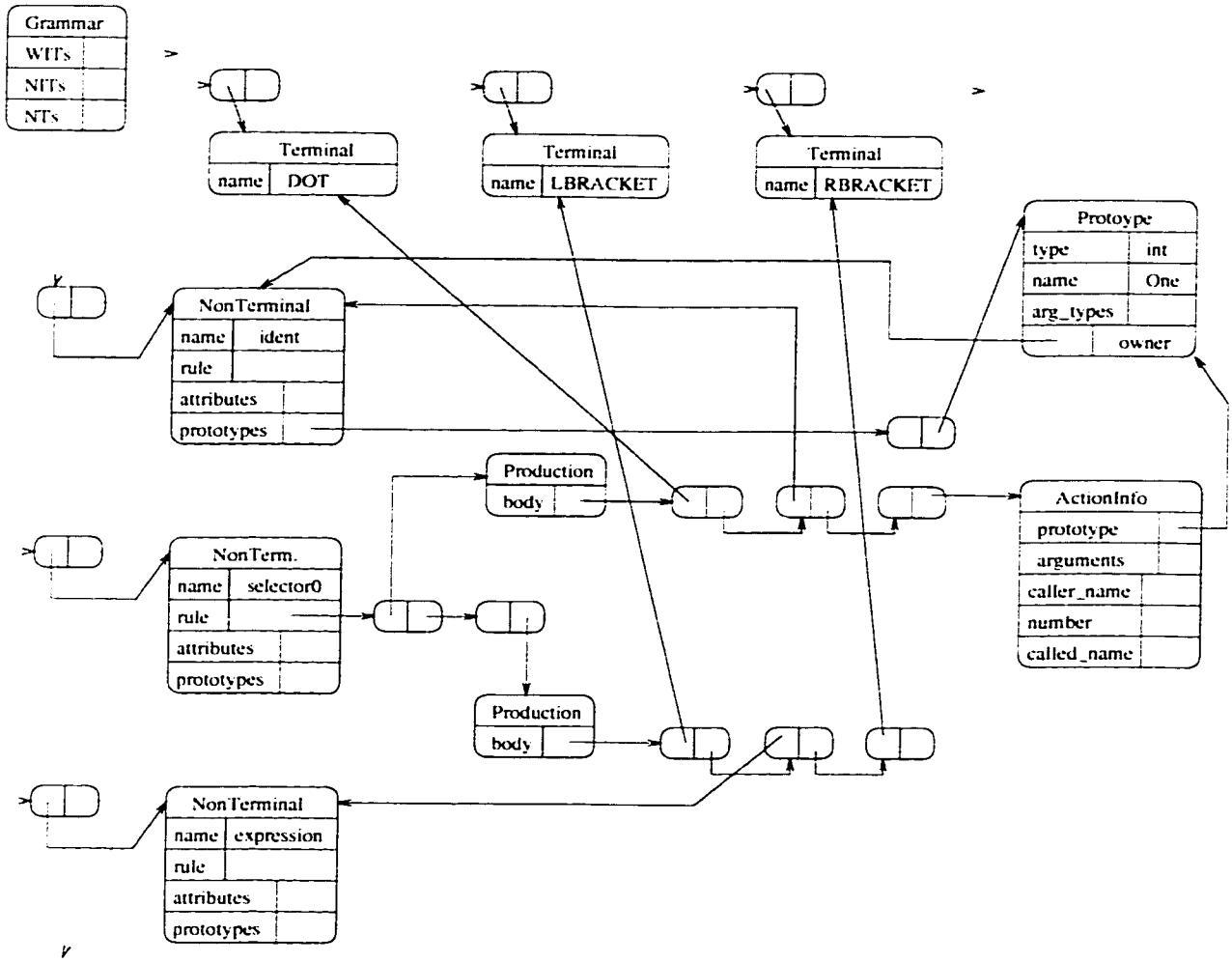


Figure 49: The data structure obtained by parsing rule "selector0" of fig 43

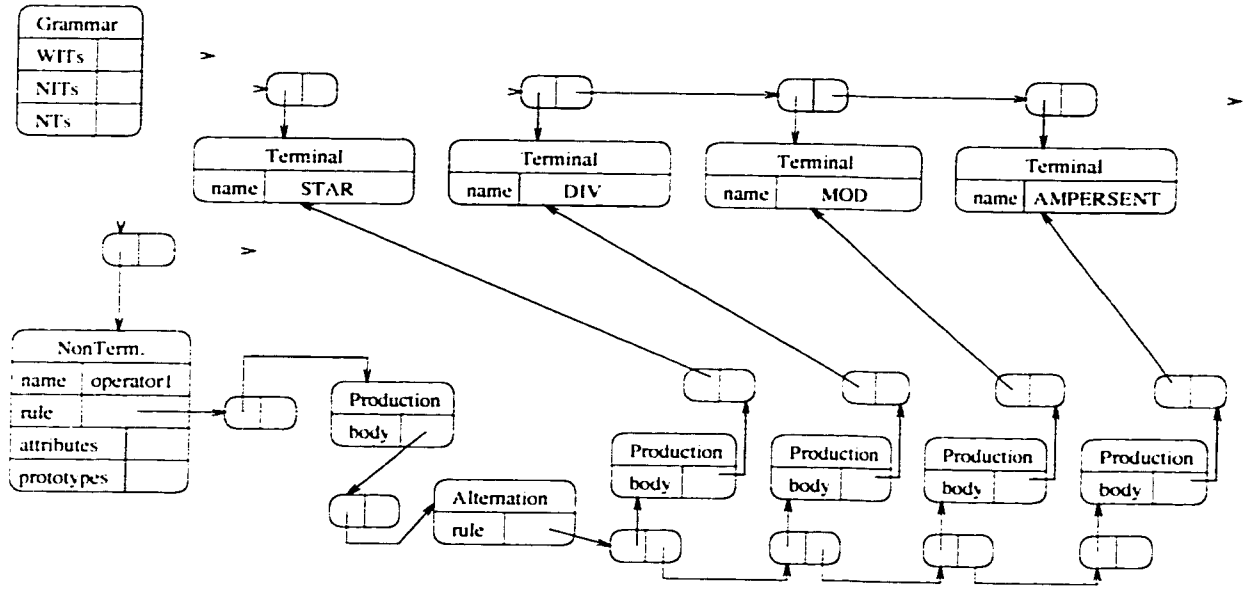


Figure 50: The data structure obtained by parsing rule "operator1" of fig 43

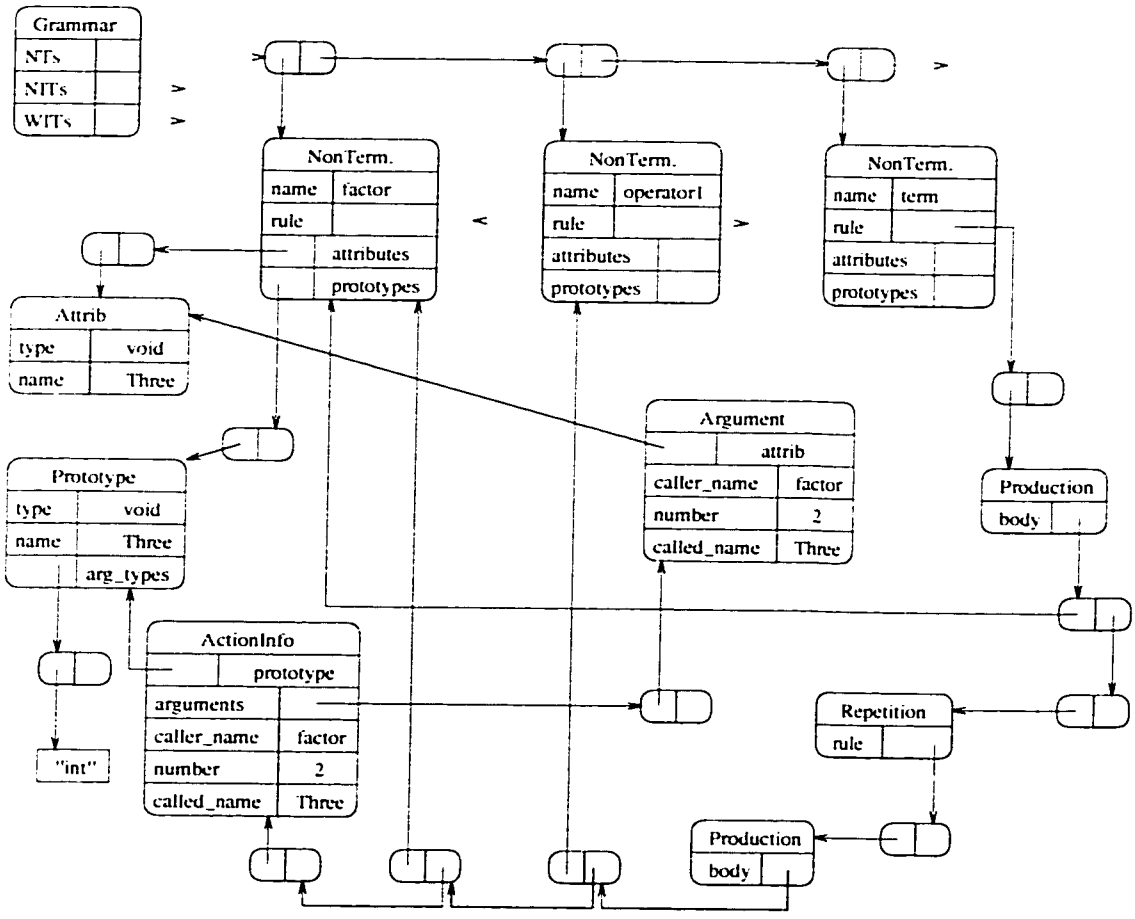


Figure 51: The data structure obtained by parsing rule "term" of fig 43

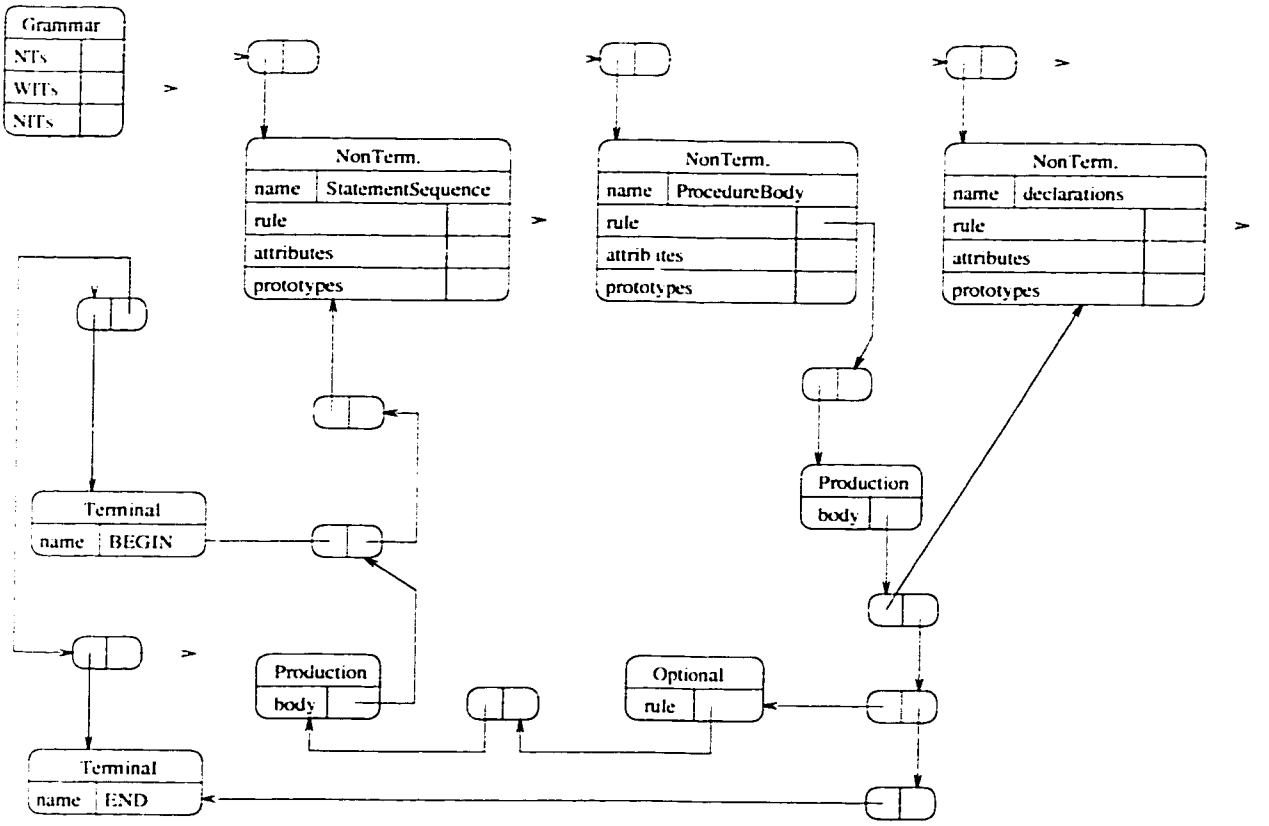


Figure 52: The data structure obtained by parsing rule "ProcedureBody" of fig 43

```
int Grammar::Is_LL1()
begin

    /* CollectFollows (Alg 3): calculates the firsts and follows of all non terminals and
    Optionals. Repetitions. Alternations existing in the definition of these non terminals. */

    for every  non terminal symbol "sym" in this grammar
        sym · CollectFollows():
    for every  non terminal symbol "sym" in this grammar
        sym · Fit_for_LL1(this . 0):
end

void Aggregate::Fit_for_LL1()
begin
    for every  production "prod" on the right hand side of this aggregate
        this · FIRST();
        if   (this · nullable) and  (not prod · nullable) and
             (prod ·FIRST() ∩ FOLLOW(this))
                 Error
    for every  production "prod" on the RHS of this aggregate
        for every  regular expr "reg_exp_sym" in any of the list of:
        optionals. repetitions. and alternations that belong to prod
            reg_exp_sym ·Fit_for_LL1();
    return  ;
end
```

Algorithm 1: Algorithms for verifying if a grammar is an LL1

FIRST Sets

Let $FIRST(\alpha)$ be the set of terminals that begin the strings derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in $FIRST(\alpha)$. The following rules show how to calculate $FIRST(\alpha)$

1. If α is a terminal, then $FIRST(\alpha)$ is $\{\alpha\}$.
2. If α is ϵ, then $FIRST(\alpha) = \{\epsilon\}$.
3. If α is a nonterminal and $\alpha \rightarrow \beta 1 \mid \beta 2 \mid \beta 3 \mid \ldots$ then $FIRST(\alpha) = \cup_k FIRST(\beta_k)$
4. If $\alpha = X_1 \ldots X_n$ where $X_1 \ldots . X_n$ are all symbols, and $n >= 2$
   then if $X_1 \stackrel{*}{\Rightarrow} \epsilon$
   then $FIRST(\alpha) = (FIRST(X_1) - \{\epsilon\}) \cup FIRST(\beta)$, where $\beta = X_2 X_3 \ldots X_n$
   else $FIRST(\alpha) = FIRST(X_1)$

We do not include ϵ in the first set, rather we set the data member "nullable" of Aggregate to 1 when the symbol is nullable.

Let class Construct contain a virtual function :

```
virtual Set<Terminal> * FIRST( ... ) = 0 ;
```

This means this function has to be be redefined in the non abstract classes that inherit Construct.

Obviously, the first of a terminal is the terminal itself, and the first of an action is nothing or null. Algorithm 2 shows the algorithms for calculating symbols' First sets. The algorithm does not show the First function of classes Terminal nor Action since they are trivial. Aggregate::First is used to calculate first sets of non terminals, optionals, repetitions, and alternations.

Line 6 of function "Aggregate::First" is supposed to issue an error statement if the grammar is left recursive. A grammar is left recursive if for some nonterminal symbol "A", directly or indirectly, $A \stackrel{*}{\Rightarrow} A\alpha$. Such a re-occurence of A on the left side of the new derivation is detected using the list of non terminals "*levels*" which is an argument of Aggregate's function "First".

"levels" can be thought of as a mirror of the language's stack which holds the frames of recursive functions calls to Aggregate::First. For example, consider the following grammar rules:

$$
\begin{array}{llll}
A & \rightarrow & B \quad C & \cdots \\
B & \rightarrow & B_1 & \cdots \\
B_1 & \rightarrow & B_2 & \cdots \\
C & \rightarrow & C_1 & \cdots \\
C_1 & \rightarrow & C_2 & \cdots \\
C_2 & \rightarrow & D_1 \quad D_2 & \cdots \\
\cdots & & &
\end{array}
$$

Where B, C, B_1, B_2, C_1, C_2, D_1, and D_2 are all nullable. Figure 53 shows the variations of levels during the calculation of the First set of A.

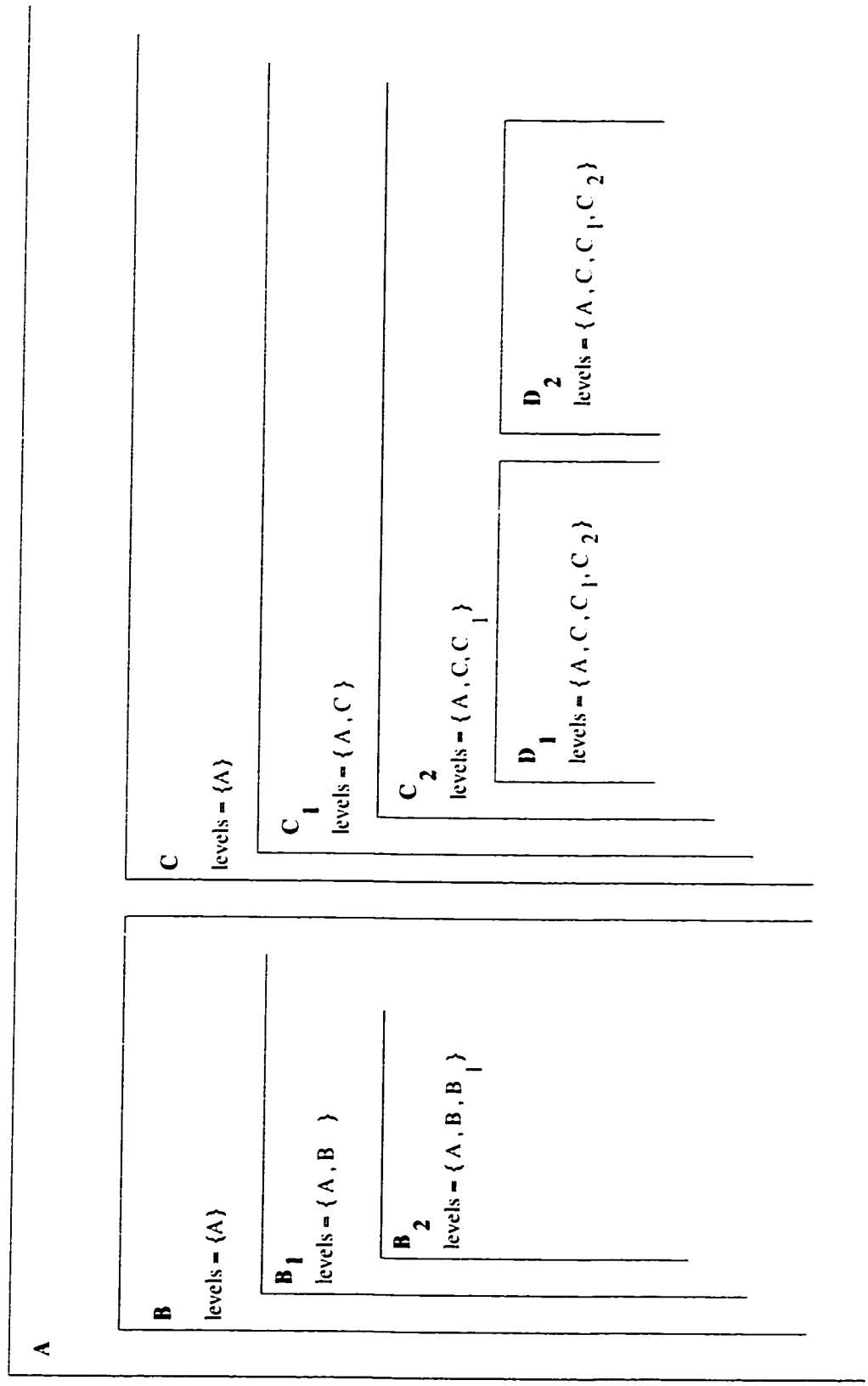


Figure 53: The variations of "levels" during the calculation of the FIRST set of A

```
Set<Terminal> Production::FIRST(Set<NonTerminal> levels)
begin
    for every symbol "sym" in the body of this production
        First = First + sym · FIRST(levels)
        if (sym is not nullable) break
        if ( sym is the last symbol in prod )
            if (owner · nullable) and (owner is NT or ALTERNATION)
                Error /* not LL(1). more than one production of symbol are nullable */
            this· nullable = TRUE ;
            this· owner · nullable = TRUE
    return First
end

Set<Terminal> Aggregate::FIRST(Set<NonTerminal> levels)
begin
1:  if (this aggregate was not defined by means of any other symbols)
2:      this· nullable = TRUE ;
3:      return NULL ;
4:  if ( this is NT)
5:      if (this ∈ levels )
6:          Error /* Grammar is left recursive */
7:      Set <NonTerminal> levels1 = levels + this ;
8:  this· First = FRST(this· rule , levels1) ;
9:  return this· First ;
end

Set<Terminal> FRST(Set<Production> ProdList , Set<NonTerminal> levels)
begin
    if (not ProdList or ProdList is empty ) return NULL ;
    Set< Terminal> Result = 0;
    for every production "prod" in ProdList
        Set<Terminal> TermList = prod · FIRST(levels);
        if (TermList ∩ Result)
            Error ; /* more than 1 production of owner start with same terminal */
        Result = Result + TermList
    return Result ;
end
```

Algorithm 2: Functions for calculating symbols' first sets

FOLLOW Sets

The following steps are needed to compute non terminals and regular expressions follow sets:

1. if there is a production $A \rightarrow \alpha B \beta$ where B is either a non terminal or a regular expression
   if $(FIRST(\beta) - \{\epsilon\}) \not\subset FOLLOW(B)$
   $FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) - \{\epsilon\})$

2. if there is a production $A \rightarrow \alpha B.$ or a production $A \rightarrow \alpha B.\beta$ where $\beta \stackrel{*}{\Rightarrow} \epsilon$
   if $FOLLOW(A) \not\subset FOLLOW(B)$
   $FOLLOW(B) = FOLLOW(A) \cup FOLLOW(B)$

3. if there is an optional or a repetiton O containing αB or $\alpha B.\beta$ where $\beta \stackrel{*}{\Rightarrow} \epsilon$
   and B is either a non terminal or a regular expression
   if $FOLLOW(O) \not\subset FOLLOW(B)$
   $FOLLOW(B) = FOLLOW(O) \cup FOLLOW(B)$

4. if there is a repetiton R containing αB or $\alpha B.\beta$ where $\beta \stackrel{*}{\Rightarrow} \epsilon$
   and B is either a non terminal or a regular expression
   if $(FIRST(R) - \{\epsilon\}) \not\subset FOLLOW(B)$
   $FOLLOW(B) = (FIRST(R) - \{\epsilon\}) \cup FOLLOW(B)$

5. if there is an alternation $A = (\cdots \mid B \mid \cdots)$ where B is a non terminal
   if $(FOLLOW(A) \not\subset FOLLOW(B)$
   $FOLLOW(B) = FIRST(A) \cup FOLLOW(B)$

6. repeat steps 1, 2, 3, 4, and 5 until nothing can be added to any follow set.

The algorithms for calculating the follow sets are shown in algorithm 3.

On line 1 of function NonTerminal::CollectFollows in algorithm 3 the word trace is used for the first time. Trace is a class that looks like this:

```
class Trace
{
   private :
      Node<Construct> * node ;
      Production * prod ;
   public :
     Trace(Node<Construct> * n , Production * p)
     {
       node = n ;
       prod = p ;
```

```
        };
        ....
    };
```

The calculation of the follow set of a non terminal involves tracking every position in which this non terminal occurred on the right hand side of a production and checking what follows it. This search is time consuming. If a non terminal has a list of all places in which it has occurred some time can be saved. Trace with its two data members "prod" and "node" provide all information needed about the places in which the non terminal occured: the production and the specific node in the body of that production.

The construction of this traces list for every non terminal is an easy task that can be done during parsing. If we go back to figure 48 and consider which rule in the grammar specifies the introduction of a non terminal on the right hand side of a grammar, we will find it to be in rules 15, 18, and 19 :

SymOrRegExp → *id* {**action17**}
Alternation → {**action22**} "(" *Actions* *id*{**action17**} *Actions* " |" *SubAlternation* {**action23**} ")"
SubAlternation → {**action24**} [*Actions* [*id* {**action17**} *Actions* [" |" *SubAlternation* {**action25**}]]]

As a review, Action17 in rule 15 is :

1: **if** ($\exists$ *symbol* : *symbol* $\in$ *Grammar* **and** *symbol* · *name* == < *id* · *lexeme* >)
2: *prod*· body = *prod*· body + *sym*
3: **else**
4: Error

On line 2, of action17, the symbol is appended to the production's body. Adding the following statement between lines 2 and 3 will save the trace of a non terminal:

if (*sym* **is** NT)
    Trace *t* = new Trace(*prod*· body · end , *prod*);
    sym·traces = *sym*·traces + *t* ;

5.3.4 Building the Inherited List

The concept of inheritance between non terminal symbols was introduced in chapter 4. Definitely, this inheritance between symbols affect the generated code. C++ requires that if a class A inherits classes B and C, that the class declaration of A indicates this as: class A : public B, public C. Thus, it is necessary, when generating the code for a non terminal A, to have at hand a list of all non terminals that A inherits. Such a list can be described with such a data member in class NonTerminal:

```
void NonTerminal :: CollectFollows()
begin
1:  for every trace "t" in the trace list of this non terminal
2:      CalculateFollow(t·prod , t·node);
3:  for every production "prod" on the right hand side of this non terminal
4:      prod · RegExpFollows();
end

void Aggregate :: CalculateFollow(Production p , Node<Construct> node)
begin
    Node<Construct> next = node · next;
    if (next)
        Set<Construct> tmp = new Set containing elements from next to p·body·end.
        Production phoney = new Production whose body is tmp ;
        Set<Terminal> result1 = phoney · FIRST(0) ;
        Set<Terminal> result2 = result1 + ;
    if ("this" is REP)
        result2 = result1 + FIRST();
    if (result2 ⊄ Follow)
        UpdateDependantFollows[1](result2);
    Follow = Follow + result1 ;
end

void Aggregate::UpdateDependantFollows(Set<Terminal> s)
begin
    for every production "prod" in the rule defining this aggregate
        for every symbol "sym" in the body of prod from the last backward
            if (s ⊆ Follow) return ;
            sym·Follow = sym·Follow + s ;
            sym · UpdateDependantFollows(s);
            if (sym is not nullable) break ;
end

void Production::RegExpFollows()
begin
    for every regular expression "sym" in any of the optionals,
    repetitions or alternations lists of "this" production.
        sym · FIRST() ;
        sym · CalculateFollow(this , sym · position) ;
        for every production "prod" in the set of rules defining sym
            prod · RegExpFollows();
end
```

1. *UpdateDependantFollow: A virtual function of class construct. This function is redefined in classes Terminal, ActionInfo, and Aggregate. Classes NonTerminal, Optional, Repetition, and Alternation inherit class Aggregate and need not redefine this function. We have seen earlier when we introduced the steps necessary for the calculation of the follow sets, in 2, 3, and 5 that the follow set of one aggregate can be dependent on the follow of another. This function takes care of these cases.*

Algorithm 3: Functions for calculating symbols' follow sets

```
Set<NonTerminal> * inherited ;
```

The following algorithm, if inserted in a member function of NonTerminal, would be sufficient for initializing "inherited" :

for every trace t in the list ''*traces*'' of *this* non terminal
  Aggregate sym = t-prod-owner:
  if (sym **is** NT)
    if (t-prod has only a non terminal symbol in its body and zero or many actions)
      this-inherited = *this*-inherited + t-node-item :

We chose to include it in algorithm 3 inside CollectFollow that will be invoked by every non terminal.

5.3.5 Checking the Validity of References in Actions

The process of code generation for actions is very important. If the user's specified actions are not carefully checked before generating code (function calls) for them, they could cause the generated parser to crash. The risks stem from the fact that actions contain references to member functions and data members of symbols. For example, the user could write in a rule :

$$E \rightarrow ID \quad "+" \quad F \quad \{ \quad E.Val(\cdots) \quad \} \quad :$$

When writing the action, the user mistakenly typed "E" instead of "F". If we just go ahead and transform this action we would have introduced a bug into the generated program by our own hands since Val is not a member function of E.

The same logic applies also to the arguments of a function.

Another point that has to be checked carefully also, is the validity of references used in an action. For example, the user could write:

$$E \rightarrow ID \quad "+" \quad F \quad \{ \quad G.Val(\cdots) \quad \} \quad :$$

Obviously, G does not occur anywhere in this production. Therefore, the action has to be ignored with a warning message issued to the user. Similarly, the action has to be ignored also when the symbol referenced occurs textually preceding the action. The same logic applies to arguments.

When it is decided that an action or an argument ought to be ignored, the flag "ignored" (an integer) which is a data member of "Trio" (inherited by classes ActionInfo and Argument) is set to 1. While the code is being generated, an action will be skipped (no code will be generated for it) and an argument will be replaced by 0 upon finding this field to be 1.

CheckActionInfo is a member function of class ActionInfo, that checks the validity of the action. This function has to be invoked by every action in the grammar before code is being generated for it. CheckActionInfo is presented in algorithm 4. CheckActionInfo also matches the references to symbols in the actions specified by the users to the correct references in the genrated code. For example, in the rule above F will be represented in the generated program by "s3".

```
void ActionInfo::CheckActionInfo(Grammar grammar , Production prod)
begin
    int counter = 0, index = 0 ;
    SearchTrioResult[1] Result = NOT_FOUND ;
    if (∃ sym : sym ∈ grammar and sym·name == caller_name)
        HasAttrib has = sym ;
        if (! has·CheckValidFuncOwner[2](this))
            Error
        prod·SearchTrio[3](this, this, index, counter, Result);
        if (Result == FOUND)
            for every argument arg in the list of agruments arguments of this action
                Trio * trio = arg ;
                counter = index = 0 ;
                Result = NOT_FOUND ;
                if (∃ sym : sym ∈ grammar and sym·name == arg·caller_name)
                    HasAttrib has = sym;
                    if (! has·CheckValidAttribOwner[3](arg))
                        Error
                    prod·SearchTrio(this, trio, index, counter, Result);
                else
                    Error
    else
        Error
end
```

1. *SearchTrioResult: An enumerate that contains three values : NOT_FOUND, FOUND, and ACTION_ENCOUNTERED*

2. *CheckValidFuncOwner: A virtual function of class HasAttrib. Redefined in classes WithInfoTerm and NonTerm. This function checks whether the symbol has a member function such as the argument passed. For non terminals, the action has to be looked up also in the list of functions of inherited non terminals. The function returns one if the symbol has such a function, and zero otherwise.*

3. *CheckValidAttribOwner: A virtual function of class HasAttrib. Redefined in classes WithInfoTerm and NonTerm. This function checks whether the symbol has a data member such as the argument passed. For non terminals, the attribute has to be looked up also in the list of attributes of inherited non terminals too. The function returns one if the symbol has such an attribute, and zero otherwise.*

Algorithm 4: ActionInfo :: CheckActionInfo

```
void Production::SearchTrio(ActionInfo a_info, Trio trio, int index,
                            int &counter, SearchTrioResult &Result)
begin
    int val = 0 :
    if (trio-caller_name == this-owner-name) and (! trio-number)
        Result = FOUND :
        return :
    for every symbol Sym in the body of this production
        val = index :
        index+ = Sym-SearchTrio[1](a_info.trio. index. counter. Result):
        if (Result == FOUND)
            if (Sym is NT) or (Sym is WIT)
                trio-index[2] = "s" + val :
            return :
        else
            if (Result == ACTION_ENCOUNTERED)
                return :
    if (Result == NOT_FOUND)
        Warning("Sym does not occur to the left of the action") :
    return :
end
```

1. *SearchTrio: A pure virtual member function of class Construct. SearchTrio has the following prototype :*

   ```
   SearchTrio(ActionInfo, Trio, int, int &, SearchTrioResult &)
   ```

   The & means the argument is passed by reference. SearchTrio is redefined in all classes that inherit class Construct:

   - *Terminal: The function just returns 1.*
   - *WithInfoTerm and NonTerminal: The body of their functions is the same:*

     ```
     if (name == trio-caller_name)
         counter + + :
         if (counter == trio-number)
             Result = FOUND:
     return 1 :
     ```

   - *Optional, Repetition, and Alternation: Due to the similarities between their three functions we combined them in algorithm 6*
   - *Actions: The algorithm is the following:*

     ```
     if (this == a_info)
         Warning("symbol referenced does not occur to the left of the action") :
         trio-ignore = 1:
         Result = ACTION_ENCOUNTERED:
     return 0 :
     ```

2. *index : member function of class Trio. Contains the string that when printed gives the correct match to the symbol in the generated code with the symbol specified in the action.*

Algorithm 5: Production :: SearchTrio

```
for every production prod in the list of productions of this regular expression
    int dummy = 0 . tmp = 0 . val = 0;
    for every symbol Sym in the body of prod
        val = tmp + index ;
        tmp += Sym->SearchTrio(a_info . trio . index . counter . Result);
        if (Result == FOUND)
            if (Sym is NT or Sym is WIT)
                if (this is CHOICE)
                    trio->index = "s" + val ;
                    return tmp ;
                else
                if (this is ALTER)
                    if (a_info->owner <> prod)
                        Warning("Sym is not in the same production as action")
                        trio->ignore = 1;
                        return 1 ;
                    trio->index = "tmp" + index ;
                    return 1;
                else
                if (this is REP)
                    if (a_info->owner <> prod)
                        Warning("Sym is not in the same production as action")
                        trio->ignore = 1;
                        return 1 ;
                    trio->index = "tmp_rep_" + dummy ;
                    return 1;
        else
        if (Result == ACTION_ENCOUNTERED)
            return 1 ;
        dummy + + ;
```

Algorithm 6: The algorithms of SeachTrio for optionals, repetitions, and alternations combined in one algorithm

5.3.6 Code Generation

The entire process of code generation hinges around the grammar described by the user's input file and transformed later by our parser generator into the data structure held by class Grammar. Therefore, it would be natural to enable Grammar to generate the code by adding to it a member function :

```
void Grammar::GenCode(...);
```

Without worrying about small details we can divide the code generated mainly into four different categories, as has been understood from chapter 4 :

1. A part used by the scanner:

   (a) As was shown in figure 42, the class for the scanner contains a member function TokenTextVal which can be generated automatically.

   (b) In order for the scanner to work, the name of every terminal in the grammar must be assigned an integer value. This can be automatically produced by generating C language define commands.

   Let's call the function for doing this :

   ```
   char ** Grammar::GenTokenTextVal(...);
   ```

   which is also a member function of Grammar. This function returns an array containing the names of NITs symbols. This array is useful for function GenTermCode introduced bellow. GenTokenTextVal is also responsible for generating function GetTokenTextVal which is a member function of class Scanner. This function takes as an argument the token value of a terminal and return its name. GenTokenTextVal is shown in algorithm 8.

2. NoInfoTerminals: A class NITCollection has to be declared and defined. Such a class acts as a container for NITs. Let's call the function for doing this :

   ```
   void Grammar::GenTermCode(...);
   ```

   which is also a member function of Grammar. GenTermCode is shown in algorithm 9.

3. WithInfoTerminals: A special class must be created for every terminal on which attributes or protoypes have been declared. Let's call the function for doing this :

   ```
   void Grammar::GenWithInfoTermCode(...);
   ```

   which is also a member function of Grammar. GenWithInfoTernCode simply calls the member function GenClass of every WIT symbol in Grammar. GenClass is shown in algorithm 10.

4. NonTerminals: A complete class declaration and definition must be generated for every non terminal symbol. Let's call the function for doing this :

```
NonTerminal * Grammar::GenNonTermCode(...);
```

which is also a member function of Grammar. This function returns a pointer to a non terminal symbol which is meant as a pointer to the start symbol of the grammar and which can be used in the generated main program to start the parsing process. GenNonTermCode simply calls the member function GenClass of every non terminal symbol in Grammar. GenClass is shown in algorithm 11.

Therefore the body of function GenCode should look as shown in algorithm 7.
In the algorithms will the encounter four names of files :

1. Hppfd: Contains all the declarations of classes generated.
2. Cppfd: Contains the definitions of classes members functions.
3. Declfd: Contains the skeletons definitions of actions member functions. These definitions are meant to be in a seperate file to avoid regenerating the whole parser every time the user modifies the actions.
4. Hfd: Contains values assigned to terminals that are necessary for scanning.

The algorithms described below, were meant to give an overall understanding of how the code is generated. A C++ style was used in the description of the algorithms.

5.4 Summary

This chapter presents our approach to building an object-oriented parser generator. A grammar that represents the set of all acceptable input files is given, Based on that grammar classes were created to represent every construct in a program. These classes can each generated code about itself. The four phases of parser generation, scanning, parsing, grammar validation, and code generation are detailed. Many algorithms are given and explained.

```
void Grammar::GenCode()
begin
    ...
    char ** Names = GenTokenTextVal[1]():
    GenTermCode[2](Names):
    GenWithInfoTermCode[3]():
    NonTerminal NTsym = GenNonTermCode[1]():
    if (! NTsym) /* No non terminals were found */
        Error
    ...
end
```

1. *GenTokenTextVal (alg 8): Generates member function TokenTextVal of class Scanner. This function takes as an argument the token value of a terminal and return its name. It also generates "define" macros for Token values in the header file Hfd that will be used by the scanner. In other words the name of a Terminal will be given a value.*

2. *GenTermCode (alg 9): Generates class NITCollection declaration and definition. The two member functions IniSymbols and GetSymbol of class NITCollection are also generated. Class NITCollection works as a container for NITs. So that an NIT will be instantiated only once. IniSymbol contains an array of all the NITs. And GetSymbol takes as an argument the token val of a NIT and return a pointer to this NIT. The array and these two fuctions are all static. Class NITCollection was described in chapter 4.*

3. *GenWithInfoTermCode: Calls the member function GenClass (alg 10) of every WIT symbol in Grammar.*

4. *GenNonTermCode: This function calls the symbol's member function "GenClass" (alg 11) of every non terminal symbol. The function returns the first non terminal encountered.*

Algorithm 7: GenCode

```
char ** Grammar::GenTokenTextVal()
begin
    Cppfd   << "char * Scanner::TokenTextVal(int val) \n { \n" << ...
            << "char * TextVal[ ] = { \"(Blank)\"" :
    char ** Names1 = PrintAllNames[1](NITs);
    char ** Names2 = PrintAllNames(WITs);
    Cppfd << ...
    ...
    if (Names1 or Names2)
        char * s1 = Names1 ? Names1[0] : Names2[0] :
        Cppfd   << "if ((val >= _" << s1 << ") && (val <= _UNKNOWN)) \n"
                << "return TextVal[val - START + 1] : \n":
    ...
    return Names :
end
```

1. *PrintAllNames: will take as an argument a list of terminals and will print the names of terminals in the list seperated by commas in file Cppfd. This will be part of TokenTextVal that the program started generating. It will also print a define statement for every terminal in the list in Hppfd. For example for terminal sym the following statement will be printed:*

   " *# define _" + sym · name + '\t' + sym·TokenVal + endl*

   It also returns an array of all the names of terminals in the list.

Algorithm 8: GenTokenTextVal

```
void Grammar::GenTermCode(char **Names)
begin
    int nbr_nodes = NITs · nbr_nodes :
    if (! nbr_nodes) return :
    Hppfd   << "class NITCollection \n" << "{ \n" << "private : \n"
            << "static NoInfoTerminal * symbol[" << nbr_nodes << "] : \n\n"
            << ...
    ...
    Cppfd   << "NoInfoTerminal * NITCollection::symbol[" << nbr_nodes << "]: \n\n"
            << "void NITCollection :: IniSymbols() \n { \n" :
    int counter = 0 :
    for every NIT symbol "sym" in NITs
        Cppfd   << "symbol[" << counter << "] = new NoInfoTerminal(_"
                << Names[counter++] << "):\n":
    Cppfd   << "} \n \n"
            << "NoInfoTerminal * NITCollection :: GetSymbol(int val)" << "\n { \n"
            << "if ((val >= _" << Names[0] << ") && "
            << "(val <= _" << Names[nbr_nodes - 1] << "))\n"
            << "return symbol[val - _" << Names[0] << "]: \n"
            << ...
end
```

Algorithm 9: GenTermCode

```
void WithInfoTerm::GenClass()
begin
    if (this symbol has no attributes nor actions) return ;
    Hppfd << "class " << name << ": public WithInfoTerminal \n { \n protected: \n" ;
    GenAttribCode[1](name);
    GenConstructor[2]();
    Hppfd << "\n public : \n" << name << "(int , char *); \n" ;
    GenAttribFuncCode[3](name);
    GenPrototypesCode[4](name);
    Hppfd << "} \n ;" ;
end
```

1. *GenAttribCode: A member function of class HasAttrib. For every attribute of the symbol which can be either a non terminal or a terminal (not a keyword), and which inherits class HasAttrib. generate a declaration of a data member in that symbol's class declaration.*

2. *GenConstructor: a member function of WithInfoTerm this function will generate the constructor for the symbol on which this function is invoked. In the constructor for every attribute of the symbol a statement initialzing it to zero will be printed. For example if attrib name is "val" and the symbol name is "id", then the statement printed should be : "id_val = 0 ;"*

3. *GenAttribFuncCode: A member function of class HasAttrib. For every attribute "attrib of this symbol generate a member function in "this" symbol's class to access since it is a private data member. For example if "this" symbol's name is "id" and attrib name is "val" and its type is "int" , then the printed statement is: "int _Get_id_val() { return id_val; } "*

4. *GenPrototypesCode: a member function of class HasAttrib. This function generates the declarations and definition for every "prototype" owned by the symbol that inherits HasAttrib and that has called GenPrototypesCode. The declaration of the prototype corresponds to printing a line in Hppfd about the type, name, and function's arguments types. Such information can be obtained from "prototype". It was mentioned in chapter 4, that the name of the symbol owning the attributes and the prototypes will be added to their names when generating the code. The definition of the prototype will be printed in Declfd to avoid regenerating the program every time the user changes the body of the functions. Since the body of these functions are supposed to be written by the user, a skeleton (a pair of bracelets) is generated to make the program more readable and easier to complete. The arguments are given dummy names such as* a1, a2, ...

Algorithm 10: Member function GenClass of class WithInfoTerm

```
NonTerminal * NonTerminal::GenClass(int SubClass)
begin
    if (generated[1]) return NULL :
    GenClassesInOrder[2]():
    Hppfd << "class " << name << " : public virtual NonTerminal" :
    GenInheritance[3]():
    Hppfd << ···
    GenAttribCode(name): /* introduced in alg 10 */
    int Multi = IsItMulti[4]() :
    GenConstructor(Multi): /* see alg 12 */
    Hppfd << "\n public :" << ···
    GenAttribFuncCode(name): /* introduced in alg 10 */
    GenPrototypesCode(name): /* introduced in alg 10 */
    if (SubClass)[5]
        GenInheritedAttribIni[6]():
    GenError[7]():
    Hppfd << "} :" << endl :
    generated = 1 :
    if (Multi)
        GenSubClasses(): /* see alg 15 */
    return this :
end
```

1. *generated: is a data member of class NonTerminal. It is set to one when the class for the non terminal was already generated. This is useful to prevent multiple generations of the same class. It is necessary because of function GenClassesInOrder*

2. *GenClassesInOrder: On every inherited symbol in the list of inherited symbols "inherited" of "this" non terminal GenClass is called. This indirect recursion will make sure no class is generated before a class that inherits it. This is needed for the C++ GNU compiler.*

3. *C++ requires that if a class A inherits classes B and C, that the class declaration of A indicates this as: class A : public B, public C. GenInheritance: for every inherited symbol: "NTsym" in the list of inherited symbols "inherited" of this NT it prints: ", public " + NTsym-name.*

4. *IsItMulti: returns "0" if the non terminal symbol has only one production. It also returns "0" if it has two productions and the second productions either contains the null symbol or semantic actions. IsItMulti returns "1" otherwise.*

5. *subclass: a data member of class NonTerminal. Set to one when "this" non terminal is a generated subclass. The concept of subclasses was introduced in chapter 4.*

6. *GenInheritedAttribIni (alg 16): When a subclass of a non terminal is created. all the attributes of the parent non terminal, even its inherited ones have to be mirrored into the subclass. GenInheritedAttribIni generates assignments statements to mirror attributes.*

7. *GenError: a member function of NonTerminal. GenError generates the declaration and definition of function "_Error". This function contains a prompt to the user that the symbol expected to be found by the scanner is one of such and such. These symbols are obtained from the First set of the non terminal. It contains also a call the exit function which terminates the program. The function "_Error" will be generated only if the non terminal is not nullable*

Algorithm 11: Member function GenClass of class NonTerminal

```
void NonTerminal::GenConstructor(int Multi)
begin
    Cppfd   << name << "::" << name
            << "(Scanner *MyScanner) : NonTerminal(MyScanner)" :
    GenInheritedCons[1]():
    Cppfd << "\n { \n char * lexeme : \n" :
    IniAttribToZero[2](name):
    if there are no productions on the RHS of this non terminal
        Cppfd << "} \n":
        return :
    if (Multi)
        Hppfd << "Construct * s0 : \n"
        Cppfd << "s0 = NULL : \n" :
    else
        Production prod = the production on the RHS of this non terminal
        int nbr = CountProdSyms[3](prod, 0, 0):
        for i from 0 to nbr
            Cppfd << "s" << i << " = NULL : \n":
    int i = 1 :
    for every production prod on the RHS of this non terminal
        GenProdSymsInfo(prod, name , i , 0, Multi): /* see alg 14*/
        i ++ :
    if (! nullable)
        Cppfd << "_Error( ··· ) : \n " :
    Cppfd << "} \n" :
end
```

1. *GenInheritedCons: Member function of class non terminal. For every inherited symbol: "NTsym" in the list of inherited symbols: inherited of this NT it prints: ", " + NTsym.name + "(MyScanner)" . This is necessary because C++ requires that if A inherits B, the constructor of A calls that of B.*

2. *IniAttribToZero: a member function of HasAttrib. this function will generate for every attribute of the symbol a statement initialzing it to zero. For example if attrib name is "val" and the symbol name is "id", then the statement printed should be : "id_val = 0 ;"*

3. *CountProdSyms: returns an integer representing the number of symbols in a production for which data members were created. This function works by adding up the values obtained by calling the member function CountSyms of every symbol inside prod. CountSyms is a virtual function of Construct redefined in heir classes. CountSyms of NT, WIT, NIT, Alternation, and Repetition just return "1". CountSyms of an Action return "0". CountSyms of an Optional is shown in algorithm 13.*

Algorithm 12: Member function GenConstructor of class NonTerminal

```
int Optional::CountSyms(int index , int Multi)
begin
    int counter = 0 :
    for every symbol "Sym" in the production "Prod" representing "this" optional
        counter += Sym-CountSyms(counter + index, Multi):
    return counter :
end
```

Algorithm 13: Optional::CountSyms

```
void GenProdSymsInfo(Production Prod, char * name , int prod_nbr,
                     int index, int Multi)
begin
    if (!Prod-nullable)
        GenInFirst[1](Prod-FIRST() , "if"):
        Cppfd << "{ \n":
    if (IsSubClassNeeded[2](Prod , Multi))
        Cppfd   << name << "_" << prod_nbr << " * tmp0 = new "
                << name << "_" << prod_nbr << "(MyScanner): \n"
                << "tmp0→InitialiseInheritedAttrib(this): \n"
                << "s0 = tmp0 : \n" :
    else
        int counter = 0 :
        for every symbol Sym in the body of this production
            counter += Sym-GenSymInfo[3](index + counter, Multi):
    if (!Prod-nullable)
        Cppfd << "return : \n } \n" :
end
```

1. *GenInFirst: this function has two arguments: a set of terminals "First" and a string "keyword". GenInFirst generates a conditional statement which can be either an "if" or a "while" statement depending on "keyword". The conditional statement compares the current token found by the scanner with each element of "First" if none of them is equal, the user is prompted that the expected token is one of such and such. The conditional statement contains also a call the "exit" function which terminates the program.*

2. *IsSubClassNeeded: This function returns "1" if a subclass of the non terminal owning "Prod" needs to be created to represent "Prod". A subclass is needed if IsMulti() returns 1 (see alg 11) when the non terminal is checked and if Prod contains more than one symbol (non actions) or even one symbol but that is either a repetition or an alternation.*

3. *GenSymInfo: A virtual member function of class Construct. This function will be inherited and redefined by all classes that inherit class Construct, such as NonTerminal, Terminal, WithInfoTerm, Repetition, Optional, Alternation, Action. The function is used to generate code about the symbol. The different variants of GenSymInfo are shown in algorithms: 17, 18, 19, 20, 21.*

Algorithm 14: GenProdSymsInfo

```
void NonTerminal::GenSubClasses()
begin
    int prod_nbr = 1 ;
    for every production prod on the right hand side of this non terminal
        if (IsSubClassNeeded(prod . 1)) /* introduced in alg 14*/
            ...
            NonTerminal sub_class[1] = new NonTerminal(name + prod_nbr. prod. this);
            sub_class·GenClass(1); /* 1: means it is a subclass. see alg 11*/
        prod_nbr + + ;
end
```

1. *Create a new non terminal whose name is name + prod_nbr. right hand side is prod. and which inherits "this" non terminal.*

Algorithm 15: NonTerminal::GenSubClasses

```
void NonTerminal::GenInheritedAttribIni()
begin
    NonTerminal * parent = the NT inherited when this was created as a subclass
    Hppfd << "void InitialiseInheritedAttrib(" << parent·name << " *); \n" ;
    Cppfd   << "void " << name << ":: InitialiseInheritedAttrib("
            << parent·name << " * s) \n { \n" ;
    StartInheritedAttribIni();
    Cppfd << "} \n" ;
end

void NonTerminal::StartInheritedAttribIni()
begin
    for every inherited symbol "parent" in set of inherited symbols of this non terminal
        for every attribute "attrib" of parent
            Cppfd   << parent·name << "_" << attrib·name << " = s→"
                    << parent·name << "_" << attrib·name << ":\n" ;
            parent·StartInheritedAttribIni();
end
```

Algorithm 16: NonTerminal::GenInheritedAttribIni

```
int Terminal::GenSymInfo(int index, int Multi)
begin
    if (! Multi )
        Hppfd << "NoInfoTerminal * s" << index << " : \n" :
    Cppfd   << "MyScanner → match(_" << name << ") : \n"
            << "s" << index << " = NITCollection::GetSymbol(_" << name << "): \n" :
    return 1 :
end

int WithInfoTerm :: GenSymInfo(int index, int Multi)
begin
    if (! Multi )
        if (this has one or more actions or attributes)
            Hppfd << name :
        else
            Hppfd << "WithInfoTerminal " :
        Hppfd << "* s" << index << " : \n" :
        Cppfd   << "lexeme = MyScanner → Lexeme(): \n"
                << "MyScanner → match(_" << name << "): \n" :
                << "s" << index << " = new ":
        if (this has or or more actions or attributes )
            Cppfd << name << "(_" << name << ", lexeme): \n"
        else
            Cppfd << "WithInfoTerminal" << "(_" << name << ", lexeme): \n ":
    else
        Cppfd   << "lexeme = MyScanner → Lexeme(): \n"
                << "MyScanner → match(_" << name << "): \n":
        if (this has any actions or attributes)
            Cppfd   << name << " * tmp" << index << " = new "
                    << name << "(_" << name << ", lexeme): \n"
                    << "s" << index << " = tmp" << index << " : \n" :
        else
            Cppfd   << "s" << index << " = new WithInfoTerminal(_" << name
                    << ", lexeme): \n" :
    return 1 :
end

int NonTerminal::GenSymInfo(int index, int Multi)
begin
    if (! Multi)
        Hppfd << name << " * s" << index << " : \n" :
        Cppfd << "s" << index << " = new " << name << "(MyScanner) : \n" :
    else
        Cppfd   << "tmp" << index << " = new " << name << "(MyScanner) : \n"
                << "s" << index << " = tmp" << index << " :\n"
    return 1 :
end
```

Algorithm 17: Virtual function GenSymInfo of classes Terminal, WithInfoTerm, and NonTerminal

```
int Alternation::GenSymInfo(int index, int Multi)
begin
    if (!Multi)
        Hppfd << "Construct * s" << index << ": \n":
    int counter = 0 , nullable_found = 0 ;
    for every production "prod" in the body of "this" alternation
        if (prod · nullable)
            nullable_found = 1 ;
        else
            GenInFirst(prod · FIRST() , "if"): /* described in alg 14 */
        Cppfd << "{ \n" :
        for every symbol "Sym" in the body of prod
                Sym · GenSymInfo(index, Sym is ACTION ? Multi : 1 ):
        Cppfd << "} \n" :
        if prod is not the the last production in the body of "this" alternation)
            Cppfd << "else \n" :
    if (!nullable)
        GenError[1](Cppfd):
    else
    if (!nullable_found)
        Cppfd << "else \n s" << index << " = NULL \n " :
    return 1 :
end
```

1. *GenError: a member function of Alternation. GenError generates an else statement containing a prompt to the user that the symbol expected to be found by the scanner is one of the such and such. These symbols are obtained from the First set of the alternation. The else statement contains also a call the exit function which terminates the program.*

Algorithm 18: Alternation::GenSymInfo

```
int ActionInfo::GenSymInfo(int val, int Multi)
begin
    if (ignore) return 0 :
    char * owner_name = prototype-owner-name :
    if (! number )
        Cppfd << owner_name << "_" << called_name << "(" :
    else
        if (Multi)
            if (first three characters of index[1] are not "tmp")
                char * tmp = index + 1 : /* index starts with an 's'. Skip the 's' */
                Cppfd << "if ( tmp" << tmp << ") \n tmp" << tmp << " → " :
            else
                Cppfd << "if ( " << index << ") \n" << index << " → ":
        else
            Cppfd << "if ( " << index << ") \n" << index << " → ":
        Cppfd << owner_name << "_" << called_name << "(" :
    for every argument "arg" in the list of arguments of this ActionInfo
        arg · GenSymInfo(Multi):
        ··· /* print commas between arguments */
    Cppfd << "): \n" :
    return 0 :
end

void Argument::GenSymInfo(int Multi)
begin
    if (ignore)
        Cppfd << 0 : /* introduced in alg 14*/
        return ;
    char * owner_name = attrib-owner-name:
    if (!number)
        Cppfd << owner_name << "_" << called_name :
    else
        if (Multi) and (first three characters of index[2] are not "tmp")
            char * tmp = index + 1 : /* index starts with an 's'. Skip the 's' */
            Cppfd << "tmp" << tmp << "? tmp" << tmp:
        else
            Cppfd << index << "?" << index ;
        Cppfd << " → _Get_" << owner_name << "_" << called_name << "() : 0" :
end
```

1.2: *index values were described in algorithms 4, 5, and 6.*

Algorithm 19: Virtual function GenSymInfo of classes ActionInfo and Argument

```
int Optional::GenSymInfo(int index, int Multi)
begin
    GenInFirst(First . "if");
    Cppfd << "{ \n" ;
    Node<Construct> SymNode = WarningNested[1]();
    int counter = 0 ;
    for every symbol "Sym" in the body of the production representing
    "this" optional and which starts with SymNode
        counter += Sym · GenSymInfo(counter + index, Multi);
    Cppfd << "} \n" ;
    return counter ;
end
```

1. *WarningNested: A member function of class RegularExp. Issues a warning whenever we have an optional or a repetition enclosed inside one or many other optionals or repetitions. For example, the nesting here is unecessary:* [[[A]*]]. *WarningNested returns a pointer to the innermost nested repetition or optional, or just to the optional/repetition itself if no nesting occurs.*

Algorithm 20: Optional::GenSymInfo

```
int Repetition::GenSymInfo(int index, int Multi)
begin
    if (this·First == φ) return 0 ;
    WarningNested();
    Production prod = first production of rule that defines the body of this ;
    GenInFirst(prod·FIRST() . "if");
    Cppfd << "{ \n BinaryOp * bin_op = 0 ; \n" ;
    GenInFirst(prod·FIRST(). "while");
    Cppfd << "{ \" ;
    int counter = 0 ;
    for every symbol sym in the body of prod
        counter += sym · Rep_GenSymInfo(counter , Multi);
    if (counter)
        Cppfd << "Construct ** array = new (Construct * ) [" << counter << "]; \n" ;
    else
        Cppfd << "Construct ** array = 0 \n" ;
    for i from 0 to counter − 1
        Cppfd << "array[ " << i << " ] = tmp_rep_" << i << "; \n" ;
    Cppfd   << "bin_op = new BinaryOp(array, " << counter << ", bin_op); \n"
            << "} \n s" << index << " = bin_op ; \n"
            << "} \n else \n s" << index << " = NULL ; \n" ;
    if (! Multi)
        Hppfd << "Construct * s" << index << "; \n" ;
    return 1 ;
end
```

Algorithm 21: Repetition::GenSymInfo

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The conclusions drawn from this work are twofold. One is related to LL(1) grammars and the other is related to the use of object-oriented paradigm.

6.1.1 LL(1) Grammars

The use of LL(1) grammars has advantages and disadvantages. Without repeating the information given in Sections: 2.8 and 4.1. the generation of a recursive descent parser from an LL(1) grammar has five main advantages:

1. An LL(1) grammar is easy to read.
2. An LL(1) grammar is easy to parse.
3. An LL(1) grammar allows the use of both inherited and synthesized attributes unlike an LALR(1) which can only have synthesized attributes.
4. The generated parser is reasonably small and fast.
5. The generated parser is easier to understand and modify because it directly reflects the grammar.
6. We came up with a clearer notation for representing semantic actions than that of yacc.

On the other hand, using an LL(1) grammar has the following main disadvantages:

1. The class of languages specifiable by LL(1) grammars is smaller than the class of languages specifiable with LR(1) grammars.

2. Some restrictions are imposed on the form of the grammar making it more difficult for the programmer to design new grammars.

We believe that the advantages of having an easy to understand grammar and parser outweight the disadvantages.

6.1.2 Use of Object-Oriented Paradigm

Some of the benefits claimed for object-oriented programming are:

- Reader Comprehension.
- Encapsulation: Can be defined as a form of information hiding. It allows changes to be made to the implementation of a system with minimal effects on the end user. It is a technique by which data is packaged together with its corresponding methods. The state data in an object is said to be encapsulated from the outside world. This means that the internal data of an object can only be accessed through the message interface for that object. The way in which the internal data is accessed is hidden from the requester, because it is neither required nor convenient that the designer of the application should be aware of the internal implementation details of the method invoked by the message.
- Modularity: This concept is closely related to the concept of encapsulation. Modularity is the degree to which a program is devided into parts (modules) with well-defined, narrow interfaces. The ability to make changes within a module without affecting other modules is essential to support the evolution of large systems. A key concept in object technology is that an application can be constructed from existing modules (also called parts or components). In object-oriented languages the basis of modularity is the class definition. The elimination of unwanted dependencies between different parts of a program (sometimes called coupling), is achieved by having each object presenting to the world its public interface; the inner works are hidden, and can be modified without rippling the effect of the modification to the other modules.
- Inheritance: Allows reuse of the behavior of a class in the definition of new classes. Subclasses of a class inherit the data structure and the operations of their parent class (also called a superclass) and may add new operations and new instance variables. Inheritance is often referred to as an "is-a" relationship if the class A inherits from class B, then it is possible to say that A "is-a" B.
- Polymorphism: A word of Greek origin that means "having multiple forms". It refers to the ability to hide different implementations behind a common interface. With polymorphism, the same message can be interpreted differently by objects of different classes and therefore produce different but appropriate results.

The use of an object-oriented programming paradigm affected this work in its two components: The generated parser and the parser generator.

The Parser Generator

One of the most important conclusions drawn from the work on the parser generator is that it is not necessary for a parser generator to include a finite state machine, a push-down automaton, or a decision table. Many parser generators that claimed to be object-oriented in the past wrapped up those components in classes. Having those components wrapped up in classes does not offer any boost to the comprehensibility of the parser generator, nor does it reflect the relation between the parser generator and its product: the generated paser, which are of the most important objectives of object-oriented programming.

An object-oriented program is supposed to be data-oriented. This is how we started. A parser generator is supposed to generate a parser from a grammar. A grammar is nothing but a set of components: rules, terminal, non terminals, regular expressions, semantic actions. Classes represent very well a grammar's components. By representing every component with a class, the set of objects obtained can cooperate together on the generation of the parser by producing each the code about itself. The set of objects representing the grammar relate directly to the generated parser since there is a clear match between every object and a class of the generated parser.

Our parser generator took advantage also of the benefits of OOP mentioned earlier. The modularity provided by class definitions provided seperate naming spaces which made building each class much easier. The genericity provided by class methods simplified the coding. Inheritance also helped to reduce code size by factoring out common functionality to new superclasses.

The Generated Parser

The benefit gained by applying object-oriented programming on the generated parser is a very comprehensible object-oriented approach to parsing. Representing every symbol in the grammar with a class relates to the nodes of the parse tree. Moreover, representing semantic actions as methods in classes fits well with the object-oriented concepts of a message and a method.

The representation of attributes and semantic actions as data and function members in a class fits well with object-oriented approach where classes states are supposed to be changed by messages sent to an object.

6.2 Future Work

The are several small improvements which could be made to our parser generator. These changes do not affect the basic function of the system, but could easily be justified as convenience features.

1. Comments should be allowed in the users' grammars. This is useful and easy to add.

2. Literal strings should be allowed in the users' grammars. For example, the literal string '+' should be allowed in the users' grammars instead of writing something such as AddOp.

3. The input grammar should be expanded to allow some non-LL(1) features which could be easily handled, like common prefixes among the grammar productions can be mechanically removed.

In addition, it may be useful to allow some ambiguity, like the usual form of IF-THEN-ELSE production. This could be handled, as most LL parsers do, by providing a default rule for resolving the ambiguity.

4. The parser generator and the generated parser could be much improved by inserting some better form of syntax error handling. Currently, the parser aborts as soon as it finds a single syntax error. As the follow sets are computed anyway to generate the parser, these could be used to attempt error recovery by discarding input tokens until an element of the follow set is found.

5. Distribute the scanner in the generated parser into the classes for terminals. This way the construct for terminal ID, for example, when called will scan for ID.

6. Providing a graphical user interface for the generated parser where the user can watch the construction of the parse tree during parsing. This can be made by defining a class tree node that all generated classes inherit and which contains all the necessary functions to plot a tree node.

6.3 Summary

This chapter names the contributions of this thesis and suggests future work. The use of object-oriented techniques is very suitable for building parsers and paser generators. The clarity and simplicity gained is great. Further studies into using O-O techniques in all phases of a compiler is encouraged.

Bibliography

[AG98] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press. 1998.

[AN96] Henk Alblas and Albert Nymeyer. *Practice and Principles of Compiler Building with C.* Prentice Hall. 1996.

[ASK90] H. Alblas and J. Schaap-Kruseman. An attributed ELL(1)-parser generator. In D. Hammer, editor, *Compiler Compilers*, number 477 in Lectures Notes in Computer Science, pages 208–209. Berlin: Springer-Verlag, 1990.

[ASU86] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AU72] A. V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing.* Prentice Hall International, 1972.

[AU78] A. V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, second edition, 1978.

[Bau]

[BBG+60] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, Perlis A.J., H. Rutishauser, K. Samelson, B. Vauquois, H. Wegstein, J. A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[BBGC86] William A. Barett, Rodney M. Bates, David A. Gustafson, and John D. Couch. *Compiler Construction: Theory and Practice.* Englewood Cliffs, New Jersey: Science Research Associates, Inc, second edition, 1986.

[Cho56] N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

[Cho59] N. Chomsky. On certain formal properties of grammars. *Information and Control*, (2):137–167, 1959.

[Cho62] N. Chomsky. *Context-Free Grammars and Pushdown Storage.* RLE Quart. Prog. Report No. 65. MIT. Cambridge. Mass.. 1962.

[Chr99] Thomas Christopher. A strong LL(k) parser generator that accepts non-LL grammars and generates LL(1) tables. Technical Report 1999-3-#2-TC. Tools of Computing. March 1999.

[DeR69] F. DeRemer. *Practical Translators for LR(k) Languages.* PhD thesis. MIT. 1969.

[DP82] Frank DeRemer and Thomas J. Pennello. Efficient computation of LALR(1) lookahead sets. *ACM Trans. Prog. Lang. Syst.*. 4(4):615–649. October 1982.

[FL88] Charles N. Fischer and Richard J. LeBlanc. Jr. *Crafting a Compiler.* The Benjamin Cummings Publishing Company. 1988.

[Gál92] Jóse Fortes Gálvez. Generating LR(1) parsers of small size. In U. Kastens and P. Pfahler. editors. *Compiler Construction.* number 641 in Lectures Notes in Computer Science. pages 16–29. Berlin: Springer-Verlag. 1992.

[GJ88] Dick Grune and Ceriel J. H. Jacobs. A programmer-friendly LL(1) parser generator. *Software-Practice and Experience.* 18(1):29–38. January 1988.

[GJ90] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide.* Ellis Horwood. 1990.

[Gro88] J. Grosch. *Generators for High-Speed Front-Ends.* pages 81–92. Number 371 in Lectures Notes in Computer Science. Berlin: Springer-Verlag. 1988.

[Gro90a] J. Grosch. Efficient and comfortable error recovery in recursive descent parsers. *Structured Programming.* (11):129–140. 1990.

[Gro90b] J. Grosch. Object-oriented attribute grammars. In A. E. Harmanci and E. Gelenbe. editors. *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V).* pages 807–816. Oct 1990.

[Hol90] Allen I. Holub. *Compiler Design in C.* Prentice Hall. 1990.

[Hor88] R.N. Horspool. ILALR: An incremental generator of LALR(1) parsers. In D. Hammer. editor. *Compiler Compilers and High Speed Compilation.* number 371 in Lectures Notes in Computer Science. pages 128–136. Berlin: Springer-Verlag. 1988.

[Hor90] R.N. Horspool. Recursive ascent-descent parsers. In D. Hammer. editor. *Compiler Compilers.* number 477 in Lectures Notes in Computer Science. pages 1–10. Berlin: Springer-Verlag. 1990.

[HU79] J. E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages. and Computation.* Reading. Mass.: Addison-Wesley. 1979.

[Joh75] S. C. Johnson. Yacc - yet another compiler compiler. Technical Report CSTR 32. AT&T Bell Laboratories. Murray Hill. N.J. 1975.

[Joh88] S. C. Johnson. Yacc meets C++. *Computing Systems.* 1(2):159 167. Spring 1988.

[Kio97] Derek Beng Kee Kiong. *Compiler Technology: Tools. Translators and Language Implementation.* Kluwer Academic Publishers. 1997.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control.* (8):607 639. 1965.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *The American Mathematical Systems Theory.* (2):127 145. 1968.

[Knu71] Donald E. Knuth. Top-down syntax analysis. *Acta Informatica.* 1:79 110. 1971.

[Knu74] Donald E. Knuth. Computer science and its relation to mathematics. *Computers and People.* pages 8 11. September 1974.

[KR77] K. Kennedy and J. Ramanathan. Deterministic attribute grammar evaluator based on dynamic sequencing. In *Fourth ACM Symposuim on Principles of Programming Languages.* pages 72 85. 1977.

[Lee89] Peter Lee. *Realistic Compiler Generation.* Foundations of Computing Series. The MIT Press. 1989.

[Lem92a] Karen A. Lemone. *Design of Compilers: Techniques of Programming Language.* Addison-Wesley. 1992.

[Lem92b] Karen A. Lemone. *Fundamentals of Compilers: An Introduction to Computer Language Translation.* Addison-Wesley. 1992.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall. 1992.

[Mey94] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice Hall. 1994.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers. 1997.

[Nij82] Anton Nijholt. *Computers and Languages: Theory and Practice.* volume 4 of *Studies in Computer Science and Artificial Intelligence.* Elsevier Science Publishers B.V.. 1982.

[Nij93] Anton Nijholt. *Context-Free Grammars: Covers. Normal Forms. and Parsing.* Lecture Notes in Computer Science. Berlin: Springer. 1993.

[Par92] Thomas W. Parsons. *Introduction to Compiler Construction.* W.H. Freeman [and] Company. 1992.

[Par93] T.J. Parr. *Obtaining Practical Variants of LL(k) and LR(k) for K > 1 by Splitting the atomic k-tuple.* PhD thesis. Purdue University. 1993.

[PQ95] T.J. Parr and R.W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software-Practice and Experience.* 25(7):789–810. July 1995.

[Rév91] György E. Révész. *Introduction to Formal Languages.* Dover Publications. Inc. New York. 1991.

[Röh80] J. Röhrich. Methods for the automatic construction of error correcting parsers. *Acta Informatica.* (13):115–139. 1980.

[Seb96] Robert W. Sebesta. *Concepts of Programming Languages.* Addison-Wesley. third edition. 1996.

[SHGF85] Axel T. Schreiner and Jr. H. George Friedman. *Introduction to Compiler Construction With UNIX.* Englewood Cliffs. New Jersey: Printce-Hall. Inc.. 1985.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley. 1995.

[Sti85] Colin Stirling. Follow set error recovery. *Software-Practice and Experience.* 15(3):239 257. March 1985.

[Wat95] Bruce W Watson. Trends in compiler construction - an invited address. In *Proceedings of the Symposium of the South African Institute for Computer Scientists and Information Technologists.* Pretoria: South Africa. May 1995.

[WC93] William M. Waite and Lynn R. Carter. *An Introduction to Compiler Construction.* Harper Collins College Publishers. 1993.

[Wir96] Nicklaus Wirth. *Compiler Construction.* Addison-Wesley. 1996.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* Addison-Wesley. 1995.

[Zin91] Barbara Zino. The domino effect. parsing and lexing objects with Yacc++. *SunWorld.* 4(5):86–92. May 1991.

Appendix A

Samples Output from the Parser Generator

In this appendix we show the parser generated for the following grammar which is a subset of the Oberon0 grammar shown in figure 43:

```
% NonTerminal module alpha declarations2 IdentList statement IfStatement
        IfStatement0 expression term operator1 operator2 factor
% NoInfoTerminal MODULE SEMICOLON BEGIN END DOT VAR COLON COMMA IF ELSIF
        ELSE THEN EQUAL LESS GREATER PLUS MINUS LP RP STAR DIV
% WithInfoTerminal letter digit
% Attributes
        int factor.first ;
        char letter.second ;
        int declarations2.third ;
        float statement.five ;
% Actions
        int    module.First();
        int    letter.Second();
        float  alpha.Third();
        char   IdentList.Fourth(int , char);
        double statement.Five();
```

```
        void    IfStatement.Six(float);
        int     expression.Seven();
        int     factor.Eight();
% Rules
        module : MODULE letter [ alpha {alpha.Third();} ]*  SEMICOLON
                 [ VAR   [declarations2]* ]
                 [BEGIN statement [ SEMICOLON statement ]* ] END
                 letter [ alpha {alpha:2.Third();} ]* DOT
                 {First(); letter.Second(); }
               ;
        declarations2 : IdentList COLON letter
                        [ alpha {alpha.Third();} ]*
                        { IdentList.Fourth(third , letter.second); }
                        SEMICOLON
                      ;
        IdentList : letter [ alpha {alpha.Third();} ]*
                    [ COMMA statement { statement.Five(); } ]*
                  ;
        statement : ( IfStatement { IfStatement.Six(five); } |
                      { statement.Seven(); } )
                  ;
        IfStatement : IF IfStatement0 END [ ELSIF IfStatement0 ]*
                      [ ELSE statement [ SEMICOLON statement ]* ]
                      END
                    ;
        IfStatement0 : expression THEN statement
                       [ SEMICOLON statement ]*
                     ;
        expression : ( PLUS | MINUS | )  term [operator2 term]*
                     [ ( EQUAL | LESS | GREATER ) ( PLUS | MINUS |   )
                     term [operator2 term]*
                   ;
```

```
term : factor [ operator1 factor ]*
                    ;
               factor : letter [ alpha {alpha.Third();} ]*
                      | digit [digit]*
                      | LP expression RP {expression.Seven(); Eight();}
                      ;
               alpha : letter | digit       ;
               operator1 : ( STAR | DIV )   ;
               operator2 : ( PLUS | MINUS ) ;
```

The parser generated is distributed in four files: OOParser.h, OOParser.hpp, OOParser.cpp, OOPdecl.cpp. Some parts were omitted and replaced with "..." beecause they were either shown earlier, as is the case of the scanner, or because they are too simple.

A.1 OOParser.h

```
#define _MODULE 257
#define _SEMICOLON 258
#define _BEGIN 259
#define _END 260
#define _DOT 261
#define _VAR 262
#define _COLON 263
#define _COMMA 264
#define _IF 265
#define _ELSIF 266
#define _ELSE 267
#define _THEN 268
#define _EQUAL 269
#define _LESS 270
#define _GREATER 271
#define _PLUS 272
#define _MINUS 273
#define _LP 274
#define _RP 275
#define _STAR 276
#define _DIV 277
#define _letter 278
```

```
#define _digit 279
#define _UNKNOWN 281
```

A.2 OOParser.hpp

```
#if !defined (_OOPARSER_HPP)
#define _OOPARSER_HPP 1
extern "C" {
    #include "stdio.h"
    #include "malloc.h"
    #include "string.h"
}
#include <stream.h>
#include <assert.h>
#include "OOParser.h"

// These are necessary for Lex. They will be removed when a our lexical
// analyzer defined as an object will be provided.
extern FILE * yyin ;
extern int yylex();
extern char * yytext ;
extern int yyline ;

class Scanner {
    private :
        FILE * input ;
        int lineno ;
        int token ;
        char * lexeme ;
        void NextToken() ;
        char * TokenTextVal(int);
    public :
        Scanner(char *) ;
        int Token() ;
        char *Lexeme() ;
        int LineNo() ;
        void match(int) ;
        void match(int , char * ) ;
    Scanner() ;
} ;
```

```
class Construct {
} ;

class NonTerminal : public Construct {
    protected :
        Scanner *scanner ;
    public :
        NonTerminal(Scanner *scan) {
            assert(scan) ;
            scanner = scan ;
        } ;
} ;

class NoInfoTerminal : public Construct {
    protected :
        int TokenVal ;
    public :
        NoInfoTerminal(int val ) {
            TokenVal = val;
        } ;
} ;

class WithInfoTerminal : public Construct {
    protected :
        int TokenVal ;
        char *lexeme ;
    public :
        WithInfoTerminal(int val , char *text) {
            TokenVal = val;
            lexeme = strdup(text);
        } ;
} ;

class BinaryOp : public Construct {
    private :
        Construct **array;
        int array_size ;
        BinaryOp * previous ;
    public :
```

```
        BinaryOp(Construct ** arr = 0. int size = 0 . BinaryOp *p=0) {
            array = arr ;
            array_size = size ;
            previous = p ;
        } ;
} ;

class module;
class alpha;
class declarations2;
class IdentList;
class statement;
class IfStatement;
class IfStatement0;
class expression;
class term;
class operator1
class operator2;
class factor;
class letter;

class NITCollection {
    private :
        static NoInfoTerminal * symbol[21] ;
    public :
        static void IniSymbols() ;
        static NoInfoTerminal *GetSymbol(int);
};

class letter : public WithInfoTerminal {
    protected :
        char letter_second;
    public :
        letter(int , char *);
        char _Get_letter_second() { return letter_second; };
        int letter_Second();
} ;

class module : public virtual NonTerminal {
    protected :
```

```
            NoInfoTerminal * s0 :
            letter* s1 :
            Construct * s2:
            NoInfoTerminal * s3 :
            NoInfoTerminal * s4 :
            Construct * s5:
            NoInfoTerminal * s6 :
            statement * s7 :
            Construct * s8:
            NoInfoTerminal * s9 :
            letter* s10 :
            Construct * s11:
            NoInfoTerminal * s12 :
        public :
            module(Scanner *):
            int module_First():
            static void _Error(int line = -1):
};

class declarations2 : public virtual NonTerminal {
    protected :
            int declarations2_third:
            IdentList * s0 :
            NoInfoTerminal * s1 :
            letter* s2 :
            Construct * s3:
            NoInfoTerminal * s4 :
    public :
            declarations2(Scanner *):
            int _Get_declarations2_third() { return declarations2_third: };
            static void _Error(int line = -1);
};

class IdentList : public virtual NonTerminal {
    protected :
            letter* s0 :
            Construct * s1:
            Construct * s2;
    public :
            IdentList(Scanner *):
```

```
        char IdentList_Fourth(int, char);
        static void _Error(int line = -1);
};

class statement : public virtual NonTerminal {
    protected :
        float statement_five;
        Construct * s0;
    public :
        statement(Scanner *);
        float _Get_statement_five() { return statement_five; };
        double statement_Five();
};

class IfStatement : public virtual NonTerminal {
    protected :
        NoInfoTerminal * s0 ;
        IfStatement0 * s1 ;
        NoInfoTerminal * s2 ;
        Construct * s3;
        NoInfoTerminal * s4 ;
        statement * s5 ;
        Construct * s6;
        NoInfoTerminal * s7 ;
    public :
        IfStatement(Scanner *);
        void IfStatement_Six(float);
        static void _Error(int line = -1);
};

class IfStatement0 : public virtual NonTerminal {
    protected :
        expression * s0 ;
        NoInfoTerminal * s1 ;
        statement * s2 ;
        Construct * s3;
    public :
        IfStatement0(Scanner *);
        static void _Error(int line = -1);
};
```

```
class expression : public virtual NonTerminal {
    protected :
        Construct * s0;
        term * s1 :
        Construct * s2;
        Construct * s3;
        Construct * s4;
        term * s5 :
        Construct * s6;
    public :
        expression(Scanner *);
        int expression_Seven();
        static void _Error(int line = -1);
} ;

class term : public virtual NonTerminal {
    protected :
        factor * s0 :
        Construct * s1;
    public :
        term(Scanner *);
        static void _Error(int line = -1);
} ;

class factor : public virtual NonTerminal {
    protected :
        int factor_first;
        Construct * s0 :
    public :
        factor(Scanner *);
        int _Get_factor_first() { return factor_first; };
        int factor_Eight();
        static void _Error(int line = -1);
} ;

class factor_1 : public virtual NonTerminal, public factor {
    protected :
        letter* s0 :
        Construct * s1;
```

```
    public :
        factor_1(Scanner *);
        void InitialiseInheritedAttrib(factor *);
        static void _Error(int line = -1);
};

class factor_2 : public virtual NonTerminal, public factor {
    protected :
        WithInfoTerminal * s0 ;
        Construct * s1;
    public :
        factor_2(Scanner *);
        void InitialiseInheritedAttrib(factor *);
        static void _Error(int line = -1);
};

class factor_3 : public virtual NonTerminal, public factor {
    protected :
        NoInfoTerminal * s0 ;
        expression * s1 ;
        NoInfoTerminal * s2 ;
    public :
        factor_3(Scanner *);
        void InitialiseInheritedAttrib(factor *);
        static void _Error(int line = -1);
};

class alpha : public virtual NonTerminal {
    ...
};

class operator1 : public virtual NonTerminal {
    ...
};

class operator2 : public virtual NonTerminal {
    ...
};

#endif
```

A.3 OOPdecl.cpp

```
#include "OOParser.hpp"

int letter::letter_Second() {
}

int module::module_First() {
}

float alpha::alpha_Third() {
}

char IdentList::IdentList_Fourth(int a1, char a2) {
}

double statement::statement_Five() {
}

void IfStatement::IfStatement_Six(float a1) {
}

int expression::expression_Seven() {
}

int factor::factor_Eight() {
}
```

A.4 OOParser.cpp

```
#include "OOParser.hpp"

Scanner::Scanner(char * gram) {
... }

int Scanner::Token() {
   return token;
}

char * Scanner::Lexeme() {
```

```
    ... }

int Scanner::LineNo() {
    return lineno;
}

void Scanner::match(int val) {
    ...
}

void Scanner::match(int val, char * string) {
    ...
}

void Scanner::NextToken() {
    ...
}

Scanner:: Scanner() {
    ...
}
char * Scanner::TokenTextVal(int val)
{
    const int START = 257 ;
    char * TextVal[] = {
        "(Blank)", "MODULE", "SEMICOLON", "BEGIN", "END", "DOT", "VAR", "COLON",
        "COMMA", "IF", "ELSIF", "ELSE", "THEN", "EQUAL", "LESS", "GREATER",
        "PLUS", "MINUS", "LP", "RP", "STAR", "DIV", "letter", "digit", "UNKNOWN"
    } ;
    if (!val)
        return TextVal[0];
    else
    if ((val >= _MODULE) && (val <= _UNKNOWN))
        return TextVal[val - START + 1] ;
    else {
        cout << "Undefined token value: " << val << endl ;
        exit(0);
    }
}
```

```
NoInfoTerminal * NITCollection::symbol[21];

void NITCollection :: IniSymbols() {
    symbol[0] = new NoInfoTerminal(_MODULE);
    symbol[1] = new NoInfoTerminal(_SEMICOLON);
    symbol[2] = new NoInfoTerminal(_BEGIN);
    symbol[3] = new NoInfoTerminal(_END);
    symbol[4] = new NoInfoTerminal(_DOT);
    symbol[5] = new NoInfoTerminal(_VAR);
    symbol[6] = new NoInfoTerminal(_COLON);
    symbol[7] = new NoInfoTerminal(_COMMA);
    symbol[8] = new NoInfoTerminal(_IF);
    symbol[9] = new NoInfoTerminal(_ELSIF);
    symbol[10] = new NoInfoTerminal(_ELSE);
    symbol[11] = new NoInfoTerminal(_THEN);
    symbol[12] = new NoInfoTerminal(_EQUAL);
    symbol[13] = new NoInfoTerminal(_LESS);
    symbol[14] = new NoInfoTerminal(_GREATER);
    symbol[15] = new NoInfoTerminal(_PLUS);
    symbol[16] = new NoInfoTerminal(_MINUS);
    symbol[17] = new NoInfoTerminal(_LP);
    symbol[18] = new NoInfoTerminal(_RP);
    symbol[19] = new NoInfoTerminal(_STAR);
    symbol[20] = new NoInfoTerminal(_DIV);
}

NoInfoTerminal * NITCollection :: GetSymbol(int val) {
    if ((val >= _MODULE) && (val <= _DIV))
        return symbol[val - _MODULE];
    else {
        cout << "Unknown value for a NoInfoTerminal \n ";
        exit(0);
    }
}

letter::letter(int val , char *text) : WithInfoTerminal(val , text) {
    letter_second = 0 ;
}

module::module(Scanner *MyScanner) : NonTerminal(MyScanner) {
```

```
char * lexeme ;
s0 = NULL : s1 = NULL : s2 = NULL : s3 = NULL : s4 = NULL : s5 = NULL :
s6 = NULL : s7 = NULL : s8 = NULL : s9 = NULL : s10 = NULL : s11 = NULL :
s12 = NULL :
if ((MyScanner→Token() == _MODULE)) {
   MyScanner→match(_MODULE) :
   s0 = NITCollection::GetSymbol(_MODULE):
   lexeme = MyScanner→Lexeme():
   MyScanner→match(_letter):
   s1 = new letter(_letter, lexeme):
   if ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit)) {
      BinaryOp * bin_op = 0 ;
      while ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit) {
         alpha * tmp_rep_0 = new alpha(MyScanner):
         if ( tmp_rep_0)
            tmp_rep_0→alpha_Third():
         Construct ** array = new (Construct * ) [1];
         array[ 0 ] = tmp_rep_0:
         bin_op = new BinaryOp(array, 1, bin_op):
      }
      s2 = bin_op ;
   }
   else
      s2 = NULL ;
   MyScanner→match(_SEMICOLON) :
   s3 = NITCollection::GetSymbol(_SEMICOLON):
   if ((MyScanner→Token() == _VAR)) {
      MyScanner→match(_VAR) :
      s4 = NITCollection::GetSymbol(_VAR);
      if ((MyScanner→Token() == _letter)) {
         BinaryOp * bin_op = 0 ;
         while ((MyScanner→Token() == _letter)) {
            declarations2 * tmp_rep_0 = new declarations2(MyScanner):
            Construct ** array = new (Construct * ) [1];
            array[ 0 ] = tmp_rep_0:
            bin_op = new BinaryOp(array, 1, bin_op);
         }
         s5 = bin_op ;
      }
      else
```

```
        s5 = NULL ;
    }
    if ((MyScanner→Token() == _BEGIN)) {
        MyScanner→match(_BEGIN) ;
        s6 = NITCollection::GetSymbol(_BEGIN);
        s7 = new statement(MyScanner) ;
        if ((MyScanner→Token() == _SEMICOLON)) {
            BinaryOp * bin_op = 0 ;
            while ((MyScanner→Token() == _SEMICOLON)) {
                MyScanner→match(_SEMICOLON);
                NoInfoTerminal * tmp_rep_0 = NITCollection::GetSymbol(_SEMICOLON);
                statement * tmp_rep_1 = new statement(MyScanner);
                Construct ** array = new (Construct * ) [2];
                array[ 0 ] = tmp_rep_0;
                array[ 1 ] = tmp_rep_1;
                bin_op = new BinaryOp(array, 2, bin_op);
            }
            s8 = bin_op ;
        }
        else
            s8 = NULL ;
    }
    MyScanner→match(_END) ;
    s9 = NITCollection::GetSymbol(_END);
    lexeme = MyScanner→Lexeme();
    MyScanner→match(_letter);
    s10 = new letter(_letter, lexeme);
    if ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit)) {
        BinaryOp * bin_op = 0 ;
        while ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit)) {
            alpha * tmp_rep_0 = new alpha(MyScanner);
            if ( tmp_rep_0)
                tmp_rep_0→alpha_Third();
            Construct ** array = new (Construct * ) [1];
            array[ 0 ] = tmp_rep_0;
            bin_op = new BinaryOp(array, 1, bin_op);
        }
        s11 = bin_op ;
    }
    else
```

```
        s11 = NULL ;
      MyScanner→match(_DOT) ;
      s12 = NITCollection::GetSymbol(_DOT);
      module_First();
      if ( s1)
         s1→letter_Second();
      return ;
   }
   _Error( MyScanner→LineNo() ) ;
}

void module::_Error( int line ) {
   if (line != -1)
      cout << line << ":" ;
   cout << " An MODULE was expected. \n" ;
   exit(0);
}

declarations2::declarations2(Scanner *MyScanner) : NonTerminal(MyScanner) {
   char * lexeme ;
   declarations2_third = 0 ;
   s0 = NULL ; s1 = NULL ; s2 = NULL ; s3 = NULL ; s4 = NULL ;
   if ((MyScanner→Token() == _letter)) {
      s0 = new IdentList(MyScanner) ;
      MyScanner→match(_COLON) ;
      s1 = NITCollection::GetSymbol(_COLON);
      lexeme = MyScanner→Lexeme();
      MyScanner→match(_letter);
      s2 = new letter(_letter, lexeme);
      if ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit)) {
         BinaryOp * bin_op = 0 ;
         while ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit)){
            alpha * tmp_rep_0 = new alpha(MyScanner);
            if ( tmp_rep_0)
               tmp_rep_0→alpha_Third();
            Construct ** array = new (Construct * ) [1];
            array[ 0 ] = tmp_rep_0;
            bin_op = new BinaryOp(array, 1, bin_op);
         }
         s3 = bin_op ;
```

```
        }
        else
            s3 = NULL ;
        if ( s0)
            s0→IdentList_Fourth(declarations2_third, s2?s2→_Get_letter_second(): 0);
        MyScanner→match(_SEMICOLON) ;
        s4 = NITCollection::GetSymbol(_SEMICOLON);
        return ;
    }
    _Error( MyScanner→LineNo() ) ;
}

void declarations2::_Error( int line ) {
    if (line != -1)
        cout << line << ":" ;
    cout << " An letter was expected. \n" ;
    exit(0);
}

IdentList::IdentList(Scanner *MyScanner) : NonTerminal(MyScanner) {
    char * lexeme ;
    s0 = NULL ; s1 = NULL ; s2 = NULL ;
    if ((MyScanner→Token() == _letter)) {
        lexeme = MyScanner→Lexeme();
        MyScanner→match(_letter);
        s0 = new letter(_letter, lexeme);
        if ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit)) {
            BinaryOp * bin_op = 0 ;
            while ((MyScanner→Token() == _letter) || (MyScanner→Token() == _digit)) {
                alpha * tmp_rep_0 = new alpha(MyScanner);
                if ( tmp_rep_0)
                    tmp_rep_0→alpha_Third();
                Construct ** array = new (Construct * ) [1];
                array[ 0 ] = tmp_rep_0;
                bin_op = new BinaryOp(array, 1, bin_op);
            }
            s1 = bin_op ;
        }
        else
            s1 = NULL ;
```

```
        if ((MyScanner→Token() == _COMMA)) {
            BinaryOp * bin_op = 0 ;
            while ((MyScanner→Token() == _COMMA)) {
                MyScanner→match(_COMMA);
                NoInfoTerminal * tmp_rep_0 = NITCollection::GetSymbol(_COMMA);
                statement * tmp_rep_1 = new statement(MyScanner);
                if ( tmp_rep_1)
                    tmp_rep_1→statement_Five();
                Construct ** array = new (Construct * ) [2];
                array[ 0 ] = tmp_rep_0;
                array[ 1 ] = tmp_rep_1;
                bin_op = new BinaryOp(array, 2, bin_op);
            }
            s2 = bin_op ;
        }
        else
            s2 = NULL ;
        return ;
    }
    _Error( MyScanner→LineNo() ) ;
}

void IdentList::_Error( int line ) {
    if (line != -1)
        cout << line << ":" ;
    cout << " An letter was expected. \n" ;
    exit(0);
}

statement::statement(Scanner *MyScanner) : NonTerminal(MyScanner) {
    char * lexeme ;
    statement_five = 0 ;
    s0 = NULL ;
    if ((MyScanner→Token() == _IF)) {
        IfStatement * tmp0 = new IfStatement(MyScanner) ;
        s0 = tmp0 ;
        if ( tmp0)
            tmp0→IfStatement_Six(statement_five);
    }
    else
```

```
    s0 = NULL ;
}

IfStatement::IfStatement(Scanner *MyScanner) : NonTerminal(MyScanner) {
   char * lexeme ;
   s0 = NULL ; s1 = NULL ; s2 = NULL ; s3 = NULL ; s4 = NULL ;
   s5 = NULL ; s6 = NULL ; s7 = NULL ;
   if ((MyScanner→Token() == _IF)) {
      MyScanner→match(_IF) ;
      s0 = NITCollection::GetSymbol(_IF);
      s1 = new IfStatement0(MyScanner) ;
      MyScanner→match(_END) ;
      s2 = NITCollection::GetSymbol(_END);
      if ((MyScanner→Token() == _ELSIF)) {
         BinaryOp * bin_op = 0 ;
         while ((MyScanner→Token() == _ELSIF)) {
            MyScanner→match(_ELSIF);
            NoInfoTerminal * tmp_rep_0 = NITCollection::GetSymbol(_ELSIF);
            IfStatement0 * tmp_rep_1 = new IfStatement0(MyScanner);
            Construct ** array = new (Construct * ) [2];
            array[ 0 ] = tmp_rep_0;
            array[ 1 ] = tmp_rep_1;
            bin_op = new BinaryOp(array, 2, bin_op);
         }
         s3 = bin_op ;
      }
      else
         s3 = NULL ;
      if ((MyScanner→Token() == _ELSE)) {
         MyScanner→match(_ELSE) ;
         s4 = NITCollection::GetSymbol(_ELSE);
         s5 = new statement(MyScanner) ;
         if ((MyScanner→Token() == _SEMICOLON)) {
            BinaryOp * bin_op = 0 ;
            while ((MyScanner→Token() == _SEMICOLON)) {
               MyScanner→match(_SEMICOLON);
               NoInfoTerminal * tmp_rep_0 = NITCollection::GetSymbol(_SEMICOLON);
               statement * tmp_rep_1 = new statement(MyScanner);
               Construct ** array = new (Construct * ) [2];
               array[ 0 ] = tmp_rep_0;
```

```
                array[ 1 ] = tmp_rep_1;
                bin_op = new BinaryOp(array, 2, bin_op);
            }
            s6 = bin_op ;
        }
        else
            s6 = NULL ;
    }
    MyScanner->match(_END) ;
    s7 = NITCollection::GetSymbol(_END);
    return ;
  }
  _Error( MyScanner->LineNo() ) ;
}

void IfStatement::_Error( int line ) {
  if (line != -1)
    cout << line << ":" ;
  cout << " An IF was expected. \n" ;
  exit(0);
}

IfStatement0::IfStatement0(Scanner *MyScanner) : NonTerminal(MyScanner) {
  char * lexeme ;
  s0 = NULL ; s1 = NULL ; s2 = NULL ; s3 = NULL ;
  if ((MyScanner->Token() == _PLUS) || (MyScanner->Token() == _MINUS) ||
      (MyScanner->Token() == _LP) || (MyScanner->Token() == _letter) ||
      (MyScanner->Token() == _digit)) {
    s0 = new expression(MyScanner) ;
    MyScanner->match(_THEN) ;
    s1 = NITCollection::GetSymbol(_THEN);
    s2 = new statement(MyScanner) ;
    if ((MyScanner->Token() == _SEMICOLON)) {
      BinaryOp * bin_op = 0 ;
      while ((MyScanner->Token() == _SEMICOLON)) {
        MyScanner->match(_SEMICOLON);
        NoInfoTerminal * tmp_rep_0 = NITCollection::GetSymbol(_SEMICOLON);
        statement * tmp_rep_1 = new statement(MyScanner);
        Construct ** array = new (Construct * ) [2];
        array[ 0 ] = tmp_rep_0;
```

```
            array[ 1 ] = tmp_rep_1;
            bin_op = new BinaryOp(array, 2, bin_op);
          }
          s3 = bin_op ;
        }
        else
          s3 = NULL ;
        return ;
    }
    _Error( MyScanner→LineNo() ) ;
}

void IfStatement0::_Error( int line ) {
    if (line != -1)
        cout << line << ":" ;
    cout << " An PLUS , MINUS , LP , letter or digit was expected. \n" ;
    exit(0);
}

expression::expression(Scanner *MyScanner) : NonTerminal(MyScanner) {
    char * lexeme ;
    s0 = NULL ; s1 = NULL ; s2 = NULL ; s3 = NULL ; s4 = NULL ; s5 = NULL ; s6 = NULL ;
    if ((MyScanner→Token() == _PLUS) || (MyScanner→Token() == _MINUS) ||
          (MyScanner→Token() == _LP) || (MyScanner→Token() == _letter) ||
          (MyScanner→Token() == _digit)) {
        if ((MyScanner→Token() == _PLUS)) {
          MyScanner→match(_PLUS) ;
          s0 = NITCollection::GetSymbol(_PLUS);
        }
        else
        if ((MyScanner→Token() == _MINUS)) {
          MyScanner→match(_MINUS) ;
          s0 = NITCollection::GetSymbol(_MINUS);
        }
        else
          s0 = NULL ;
        s1 = new term(MyScanner) ;
        if ((MyScanner→Token() == _PLUS) || (MyScanner→Token() == _MINUS)) {
          BinaryOp * bin_op = 0 ;
          while ((MyScanner→Token() == _PLUS) || (MyScanner→Token() == _MINUS)) {
```

```
        operator2 * tmp_rep_0 = new operator2(MyScanner);
        term * tmp_rep_1 = new term(MyScanner);
        Construct ** array = new (Construct * ) [2];
        array[ 0 ] = tmp_rep_0;
        array[ 1 ] = tmp_rep_1;
        bin_op = new BinaryOp(array, 2, bin_op);
    }
    s2 = bin_op ;
}
else
    s2 = NULL ;
if ((MyScanner→Token() == _EQUAL) || (MyScanner→Token() == _LESS) ||
        (MyScanner→Token() == _GREATER)) {
    if ((MyScanner→Token() == _EQUAL)) {
        MyScanner→match(_EQUAL) ;
        s3 = NITCollection::GetSymbol(_EQUAL);
    }
    else
    if ((MyScanner→Token() == _LESS)) {
        MyScanner→match(_LESS) ;
        s3 = NITCollection::GetSymbol(_LESS);
    }
    else
    if ((MyScanner→Token() == _GREATER)) {
        MyScanner→match(_GREATER) ;
        s3 = NITCollection::GetSymbol(_GREATER);
    }
    else {
        cout << MyScanner→LineNo() << ":" ;
        cout << " An EQUAL , LESS , GREATER was expected. \n" ;
        exit(0);
    }
    if ((MyScanner→Token() == _PLUS)) {
        MyScanner→match(_PLUS) ;
        s4 = NITCollection::GetSymbol(_PLUS);
    }
    else
    if ((MyScanner→Token() == _MINUS)) {
        MyScanner→match(_MINUS) ;
        s4 = NITCollection::GetSymbol(_MINUS);
```

```
            }
            else
               s4 = NULL ;
            s5 = new term(MyScanner) ;
            if ((MyScanner→Token() == _PLUS) || (MyScanner→Token() == _MINUS)) {
               BinaryOp * bin_op = 0 ;
               while ((MyScanner→Token() == _PLUS) || (MyScanner→Token() == _MINUS)) {
                  operator2 * tmp_rep_0 = new operator2(MyScanner);
                  term * tmp_rep_1 = new term(MyScanner);
                  Construct ** array = new (Construct * ) [2];
                  array[ 0 ] = tmp_rep_0;
                  array[ 1 ] = tmp_rep_1;
                  bin_op = new BinaryOp(array, 2, bin_op);
               }
               s6 = bin_op ;
            }
            else
               s6 = NULL ;
         }
         return ;
      }
      _Error( MyScanner→LineNo() ) ;
}

void expression::_Error( int line ) {
   if (line != -1)
      cout << line << ":" ;
   cout << " An PLUS , MINUS , LP , letter or digit was expected. \n" ;
   exit(0);
}

term::term(Scanner *MyScanner) : NonTerminal(MyScanner) {
   char * lexeme ;
   s0 = NULL ; s1 = NULL ;
   if ((MyScanner→Token() == _LP) || (MyScanner→Token() == _letter) ||
         (MyScanner→Token() == _digit)) {
      s0 = new factor(MyScanner) ;
      if ((MyScanner→Token() == _STAR) || (MyScanner→Token() == _DIV)) {
         BinaryOp * bin_op = 0 ;
         while ((MyScanner→Token() == _STAR) || (MyScanner→Token() == _DIV)) {
```

```
            operator1 * tmp_rep_0 = new operator1(MyScanner);
            factor * tmp_rep_1 = new factor(MyScanner);
            Construct ** array = new (Construct * ) [2];
            array[ 0 ] = tmp_rep_0;
            array[ 1 ] = tmp_rep_1;
            bin_op = new BinaryOp(array, 2, bin_op);
          }
          s1 = bin_op ;
        }
        else
          s1 = NULL ;
        return ;
    }
    _Error( MyScanner→LineNo() ) ;
}

void term::_Error( int line ) {
    if (line != -1)
        cout << line << ":" ;
    cout << " An LP . letter or digit was expected. \n" ;
    exit(0);
}

factor::factor(Scanner *MyScanner) : NonTerminal(MyScanner) {
    char * lexeme ;
    factor_first = 0 ;
    s0 = NULL ;
    if ((MyScanner→Token() == _letter)) {
        factor_1 * tmp0 = new factor_1(MyScanner);
        tmp0→InitialiseInheritedAttrib(this);
        s0 = tmp0 ;
        return ;
    }
    if ((MyScanner→Token() == _digit)) {
        factor_2 * tmp0 = new factor_2(MyScanner);
        tmp0→InitialiseInheritedAttrib(this);
        s0 = tmp0 ;
        return ;
    }
    if ((MyScanner→Token() == _LP)) {
```

```
        factor_3 * tmp0 = new factor_3(MyScanner);
        tmp0->InitialiseInheritedAttrib(this);
        s0 = tmp0 ;
        return ;
    }
    _Error( MyScanner->LineNo() ) ;
}

void factor::_Error( int line ) {
    if (line != -1)
        cout << line << ":" ;
    cout << " An LP . letter or digit was expected. \n" ;
    exit(0);
}

factor_1::factor_1(Scanner *MyScanner) : NonTerminal(MyScanner), factor(MyScanner) {
    char * lexeme ;
    s0 = NULL ; s1 = NULL ;
    if ((MyScanner->Token() == _letter)) {
        lexeme = MyScanner->Lexeme();
        MyScanner->match(_letter);
        s0 = new letter(_letter, lexeme);
        if ((MyScanner->Token() == _letter) || (MyScanner->Token() == _digit)) {
            BinaryOp * bin_op = 0 ;
            while ((MyScanner->Token() == _letter) || (MyScanner->Token() == _digit)) {
                alpha * tmp_rep_0 = new alpha(MyScanner);
                if ( tmp_rep_0)
                    tmp_rep_0->alpha_Third();
                Construct ** array = new (Construct * ) [1];
                array[ 0 ] = tmp_rep_0;
                bin_op = new BinaryOp(array, 1, bin_op);
            }
            s1 = bin_op ;
        }
        else
            s1 = NULL ;
        return ;
    }
    _Error( MyScanner->LineNo() ) ;
}
```

```
void factor_1::InitialiseInheritedAttrib(factor * s) { }

void factor_1::_Error( int line ) {
   if (line != -1)
      cout << line << ":" ;
   cout << " An letter was expected. \n" ;
   exit(0);
}

factor_2::factor_2(Scanner *MyScanner) : NonTerminal(MyScanner), factor(MyScanner) {
   char * lexeme ;
   s0 = NULL ; s1 = NULL ;
   if ((MyScanner→Token() == _digit)) {
      lexeme = MyScanner→Lexeme();
      MyScanner→match(_digit);
      s0 = new WithInfoTerminal(_digit, lexeme);
      if ((MyScanner→Token() == _digit)) {
         BinaryOp * bin_op = 0 ;
         while ((MyScanner→Token() == _digit)) {
            WithInfoTerminal * tmp_rep_0 =
                        new WithInfoTerminal(_digit , MyScanner→Lexeme()) ;
            Construct ** array = new (Construct * ) [1];
            array[ 0 ] = tmp_rep_0;
            bin_op = new BinaryOp(array, 1, bin_op);
         }
         s1 = bin_op ;
      }
      else
         s1 = NULL ;
      return ;
   }
   _Error( MyScanner→LineNo() ) ;
}

void factor_2::InitialiseInheritedAttrib(factor * s) {
}

void factor_2::_Error( int line ) {
   if (line != -1)
```

```
        cout << line << ":" ;
    cout << " An digit was expected. \n" ;
    exit(0);
}

factor_3::factor_3(Scanner *MyScanner) : NonTerminal(MyScanner), factor(MyScanner) {
    char * lexeme ;
    s0 = NULL ; s1 = NULL ; s2 = NULL ;
    if ((MyScanner→Token() == _LP)) {
        MyScanner→match(_LP) ;
        s0 = NITCollection::GetSymbol(_LP);
        s1 = new expression(MyScanner) ;
        MyScanner→match(_RP) ;
        s2 = NITCollection::GetSymbol(_RP);
        if ( s1)
            s1→expression_Seven();
        factor_Eight();
        return ;
    }
    _Error( MyScanner→LineNo() ) ;
}

void factor_3::InitialiseInheritedAttrib(factor * s) {
}

void factor_3::_Error( int line ) {
    if (line != -1)
        cout << line << ":" ;
    cout << " An LP was expected. \n" ;
    exit(0);
}

alpha::alpha(Scanner *MyScanner) : NonTerminal(MyScanner) {
    ...
}

void alpha::_Error( int line ) {
    ...
}
```

```
operator1::operator1(Scanner *MyScanner) : NonTerminal(MyScanner) {
    ...
}

void operator1::_Error( int line ) {
    ...
}

operator2::operator2(Scanner *MyScanner) : NonTerminal(MyScanner) {
    ...
}

void operator2::_Error( int line ) {
    ...
}

int main(int argc , char ** argv ) {
    if ( argc != 2 ) {
        cout << "Wrong number of arguments. should be: [Program Name] [File Name] ."
             << endl ;
        exit(0) ;
    }
    NTCollection::IniSymbols():
    Scanner * scanner = new Scanner(argv[1]):
    module * root = new module(scanner) :
    if (! scanner→Token())
        cout << "Input is accepted by grammar\n":
    else
        cout << "Input is unaccepted by grammar\n ":
}
```