THE DESIGN AND IMPLEMENTATION OF A PROLOG PARSER USING JAVACC

Pankaj Gupta, B.S.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2002

APPROVED:

Paul Tarau, Major Professor
Armin Mikler, Committee Member
Roy Jacob, Committee Member
Robert Brazille, Graduate Coordinator
Krishna Kavi, Chair of the Department of Computer Sciences
C. Neal Tate, Dean of the Robert B. Toulouse School of Graduate Studies

Gupta, Pankaj, The Design and Implementation of a Prolog parser using javacc. Master of Science (Computer Science), August 2002, 66 pp., 6 tables, 11 figures, 19 references.

Operatorless Prolog text is LL(1) in nature and any standard LL parser generator tool can be used to parse it. However, the Prolog text that conforms to the ISO Prolog standard allows the definition of dynamic operators. Since Prolog operators can be defined at run-time, operator symbols are not present in the grammar rules of the language. Unless the parser generator allows for some flexibility in the specification of the grammar rules, it is very difficult to generate a parser for such text.

In this thesis we discuss the existing parsing methods and their modified versions to parse languages with dynamic operator capabilities. Implementation details of a parser using Javacc as a parser generator tool to parse standard Prolog text is provided. The output of the parser is an "Abstract Syntax Tree" that reflects the correct precedence and associativity rules among the various operators (static and dynamic) of the language. Empirical results are provided that show that a Prolog parser that is generated by a parser generator like Javacc is comparable in efficiency to a hand-coded parser.

CONTENTS

1 INTRODUCTION 1

2 JAVACC 6

2.1 JAVACC OVERVIEW 6

2.2 JAVACC GRAMMAR FILE STRUCTURE 7

2.3 LOOKAHEAD MECHANISM 11

2.4 TOKEN MANAGER 13

3 BUILDING PROLOG PARSER USING JAVACC 14

3.1 PROLOG GRAMMAR 14

3.2 ABSTRACT SYNTAX TREE 24

3.3 DYNAMIC OPERATORS IN PROLOG 32

3.4 LL PARSING 33

3.5 ERROR REPORTING 47

3.6 EMPIRICAL RESULTS 48

4 OTHER PARSING TECHNIQUES AND THEIR ADAPTABILITY TO DYNAMIC SYNTAX 56

4.1 OPERATOR PRECEDENCE PARSING 56

4.2 LR PARSING 58

4.3 DEFERRED DECISION PARSING 60

5 CONCLUSION AND FUTURE WORK 64

BIBLIOGRAPHY 65

LIST OF TABLES

3.1 Lexical tokens 18
3.2 Operator fixity determination table 38
3.3 Timing results for Javacc's generated parser 50
3.4 Comparison of Javacc's generated parser with Jinni for clauses that do not contain dynamic operators 52
3.5 Comparison of Javacc's generated parser with Jinni for clauses that contain dynamic operators 53
3.6 Comparison of Javacc's generated parser with Jinni for clauses that have embedded Prolog compound terms 55

LIST OF FIGURES

1.1 Taxonomy 3

3.1 Prolog syntax for Javacc specification file 17

3.2 Parser 22

3.3 Hierarchy of Classes for Building an Abstract Syntax Tree 25

3.4 Step by step building of the Operator tree based on their precedences 46

3.5 General structure of a Prolog clause 49

3.6 Graph showing the parsing time curves of Javacc's generated parser for clauses with (dynamic curve) and without (static curve) dynamic operators 51

3.7 Graph that compares Javacc generated parser with Jinni's hand-coded parser for Prolog terms that do not contain dynamic operators 52

3.8 Graph that compares the Javacc generated parser with that of Jinni's hand-coded parser for clauses that contain dynamic operator 54

3.9 Graph showing the comparison of Javacc generated parser with Jinni's hand-coded parser for Prolog clauses that have embedded Prolog compound terms 55

4.1 LR parser 58

CHAPTER 1

INTRODUCTION

Every programming language has a set grammar rules associated with it. These grammar rules define the syntax of the language. Based on these grammar rules, a parser for the programming language can be constructed. There are two approaches that could be taken for the construction of the parser. The parser could either be hand-coded or a parser generator tool could be used for its construction. There are a variety of parser generator tools (bottom-up and top-down) available that take a grammar specification in the Backus-Naur format as their input and generate code that would parse the language defined by the grammar. The use of a parser generator provides us with the advantage of faster development time and efficient code that could be easily maintained. However these parser generators require that the language grammar is fixed and does not change during program execution.

In programming languages like Prolog and ML, new operators with arbitrary precedence and associativity rules could be defined at run-time. Since these operators are defined at run-time, they are not present in the grammar specification of the language. Due to this, it becomes difficult to use standard parser generator tools to generate a parser for these languages. In programming languages like C, C++ and Java, there are fixed sets of operators that are hard-coded in the grammar specification of the language. Thus it is favorable to use a parser generator to generate the parser for these languages.

There are various public domain implementations of hand coded Prolog parsers. In this thesis, an implementation scenario to parse standard Prolog text using Javacc, which is a parser generator tool, is presented. The syntax definition of Prolog used in this thesis is a subset of the ISO Prolog standard. The choice to use Javacc as the parser generator tool is based on the following reasons: Javacc produces a top-down (recursive-descent) parser. As the Prolog grammar is LL in nature, a top-down parser would be a natural way of parsing Prolog. Since Javacc produces parser in Java, this parser can be easily integrated with Java based Prolog compilers like Jinni [1]. Javacc also provides extreme flexibility in its token definition and grammar specification structures that help in the parsing of a language like Prolog with dynamic operator capabilities.

OVERVIEW

Chapter 2 introduces Javacc[5] and discusses its advantages and capabilities as compared to the other parser generators. It briefly describes the structure of the grammar file that Javacc expects as its input. The look-ahead mechanism that is provided by Javacc that helps in the resolution of ambiguities at choice points in the grammar specification is explained next. This chapter concludes with the brief discussion of the "Token Manager" (lexer) in Javacc. The lexical states and lexical actions that can be specified in the "Token Manager" section of the grammar specification file are well suited for tokenizing standard Prolog text.

Chapter 3 describes the syntax of Prolog as adapted for the Javacc specification file. Though the tokens of the grammar remain the same as in the ISO Prolog standard, the grammar itself has been modified to incorporate the parsing of dynamic Prolog constructs. The inheritance hierarchy of classes that are used to build the "Abstract Syntax Tree" (AST) is discussed next followed by an introduction to dynamic operators in Prolog. For an operator both the syntax and semantics associated with it can be overloaded. This leads to a taxonomy along two axes, syntax and semantics [2] as seen in the following figure.

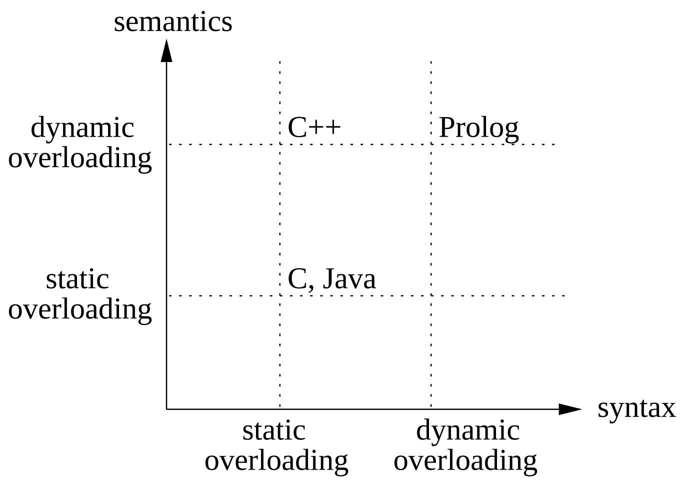

Figure 1.1: Taxonomy

Syntax overloading implies that an operator can be overloaded as unary and binary. Semantics overloading implies that the operator can be applied to different data types. Programming languages like C and Java have static overloading of operators. Therefore the number of operators in such languages are fixed. Also the data types on which these operators can be applied is also fixed by the language specification. C++ provides more flexibility as compared to the languages like C and Java. In C++, operators can be semantically overloaded for new data types. Operator definition in Prolog provides for the maximum flexibility in this taxonomy. Thus in Prolog new operators can be introduced and these can be overloaded syntactically and semantically. This flexibility leads to added complexity in the generation of the parser. However there are certain restrictions put forth by the ISO standard that help in the deterministic parsing of Prolog text with dynamic operators.

Chapter 4 overviews the existing parsing methods and briefly discusses the favorableness of these methods for parsing languages with dynamic syntax. A variation of LR parsing technique called the “Deferred Decision Parsing” [4] is discussed that is used to parse languages with dynamic syntax.

The key contribution of this thesis is the adaptation of Javacc’s syntax tree generator to support dynamic operators. This can be seen as a transformational component which reorganizes a list of terms into a new term incorporating operators as functional components. An algorithm to do the above task is presented. This module is a plug-in to the Javacc generated parser and is called by the parser when a Prolog compound term in the operator notation is identified by the parser. This module returns the term (compound term in the functional notation) to the main parser which then continues parsing further. The output of the parser is the term tree that represents the AST according to the precedence and associativity rules of the given operators.

Following is a listing of the most common terms and phrases used in this thesis.

1. LL: Left to right parse, left most derivation

2. LR: Left to right parse, right most derivation

3. Operator Notation: Infix, prefix or postfix style of representing expressions with operators.

   For example, representing $a + b$ as opposed to *+(a, b)*.

4. Functional notation: Representation of operators in the functional style.

   For example, representing *+(a ,b)* as opposed to $a + b$.

5. Fixity: Fixity of an operator can be infix, prefix or postfix. Although these fixities can apply to both unary and binary operators, in this text infix is applied only to binary operators, postfix is applied to unary operators that occur after the operand and prefix is applied only to unary operators that occur before the operand.

   For example,

   In $a + b$ "+" is an infix operator.

   In $+a$ "+" is a prefix operator.

   In $a{+}{+}$ "++" is a postfix operator.

CHAPTER 2

JAVACC

2.1 JAVACC OVERVIEW

Javacc is a java based parser generator that generates a top-down parser. Top-down parsers or recursive decent parsers allow the use of more general grammars. The only limitation of these parsers is that left recursion is not allowed because this could lead to infinite recursion. The top-down parsers have a structure that is identical to the grammar specification and are thus easier to debug. Embedding code to build abstract syntax tree in a top down parser is simpler because of the ease of passing arguments and values across the nodes of the parse tree.

In a javacc specification file since the lexical specifications and the grammar specifications are written in the same file, it is easier to read and maintain. Javacc provides different option settings that are used to customize the behavior of the generated parsers. As an example the option of Unicode processing can be turned on which will enable the generated parser to read Unicode characters. The number of tokens to look-ahead can be globally specified to resolve ambiguity at choice points. Javacc also provides extensive debugging capabilities. The DEBUG_PARSER, DEBUG_TOKEN_MANAGER, DEBUG_LOOKAHEAD options can be set to output extensive debug and diagnostic information at every stage of parsing.

By default javacc has a mechanism that looks ahead one token in the input stream to resolve ambiguity in the grammar. However there might be situations where a look-ahead of one token in the input stream does not resolve ambiguity. In such cases javacc provides a capability whereby for certain parts of the grammar more than one token can be looked ahead. Javacc provides both syntactic and semantic lookahead capabilities to resolve such ambiguities. Thus the generated parser is LL(k) at these points but remains LL(1) else where. This results in better performance.

Javacc also provides "lex" like lexical state and action capabilities. The "Token Manager" is the component of Javacc that is used to recognize tokens of the grammar. The "Token Manager" can be in any one of these lexical states and can execute user defined lexical actions for these states. The "Token Manager" is a component of javacc that is used to recognize and return the tokens of the grammar to the parser. It is an implementation of a non-deterministic finite automaton. The "Token Manager" can be in any one of these lexical states. Every lexical state could have user defined actions assigned. These lexical actions for the state get executed once the "Token Manager" enters that state.

2.2 JAVACC GRAMMAR FILE STRUCTURE

Javacc options

Parser_Begin(Identifier)

    Javacc Compilation Unit

Parser_End(Identifier)

Productions

Javacc options

The options section is the section where the various settings that customize the behavior of the generated parser are specified. This section is optional. It starts with the reserve word “option” followed by a list of one or more option bindings within braces. Some of the most widely used option settings and their descriptions are listed below.

1. LOOKAHEAD

   The default value of this option is 1. This option specifies the number of tokens to lookahead before making a decision at the choice point. Setting this option here affects the parser globally. However the local LOOKAHEAD option overrides the global one.

2. STATIC

   This is a boolean option whose default value is true. This implies that all methods and class variables are specified as static in the generated Token Manager and parser. To perform multiple passes during one run of the Java program, a call to ReInit() must be made to re-initialize the parser. If the parser is non-static, we could construct many objects with the new operator which could execute simultaneously from different threads.

3. DEBUG_PARSER,DEBUG_TOKEN_MANAGER, DEBUG_LOOKAHEAD

   The default values of these boolean options is false. When set to true, these options because the parser to provide debug information at runtime.

4. JAVA_UNICODE_ESCAPE

   The default value of this boolean option is false. When set to true, the generated parser uses an input stream object that processes Java Unicode escapes before sending them to the Token Manager.

Java Compilation Unit

The Java Compilation Unit is enclosed between the PARSER_BEGIN(*Identifier*) and PARSER_END(*Identifier*). The *Identifier* that follows the PARSER_BEGIN and PARSER_END must be the same and this identifies the name of the generated parser. The java compilation unit can contain java code so long as it contains a class declaration whose name is the same as the name of the generated parser. Javacc does not perform any detailed checks on the compilation unit. Thus the Javacc generated parser might not compile. The generated parser contains a public method corresponding to each non-terminal in the grammar file. Unlike Yacc [6] there is no single start symbol and one may parse with respect to any non-terminal in the grammar.

Productions

A grammar that is used to specify the syntax of a programming language consists of a set of productions. A production is a rule by which a sequence of terminals and non-terminals get reduced to a non-terminal. In Javacc, one can define four kinds of productions as explained below:

1. Javacode production: This is a way to write Java code for some productions instead of the usual EBNF productions. This is specially useful when it becomes necessary to recognize something that is not context free or is difficult to write a grammar for.

2. Regular Expression production: These productions are used to define the lexical entities or tokens for the grammar. These tokens get processed by the Token

Manager. In javacc, all regular expressions belong to one or many lexical states. A lexical state list can be explicitly defined for a particular regular expression or if there is not lexical state defined for a regular expression, then that regular expression belongs to the DEFAULT lexical state. In javacc there can be four kinds of regular expressions: TOKEN, SPECIAL_TOKEN, SKIP and MORE. The TOKEN type regular expressions are used to describe the tokens in the grammar. SPECIAL_TOKEN type tokens are tokens that are simply ignored by the parser. This is useful when identifying certain constructs of the language that have no significance during parsing. For example, commented code within a program. SKIP type tokens are simply ignored by the parser. The difference between SKIP and SPECIAL_TOKEN regular expression is that the later is available at parse time for extra processing, however the former is not. MORE type regular expressions are used to gradually build up a token to be passed to the parser.

3. BNF production: This is the standard way of specifying Javacc grammars. The BNF production has a format:

$$NT \rightarrow \alpha$$

where,
NT is a single non-terminal and
α is a sequence of zero or more terminals/non-terminals.
In javacc a non-terminal is written exactly like a method declaration. Since each non-terminal is translated into a method in the generated parser, this style of

writing the non-terminals makes the association obvious. The name of the non-terminal is the name of the method and the parameters and the return values declared are the means of passing values up and down the parse tree.

4. Token Manager declarations: The declarations and statements in this section are written into the generated Token Manager and are accessible from within lexical actions. See the Section "Working of Token Manager" for complete details.

2.3 LOOKAHEAD MECHANISM

The job of the parser is to read an input stream of characters and determine if these sequence of characters conform to the grammar. However there are situations when there could be multiple productions of the grammar that could match up with the sequence of characters read. If the parser had backtracking capabilities, then it would choose the first production and if that failed then if would try the second production and so on. The process of backtracking is very time consuming and the performance hit from such backtracking is unacceptable for most systems including a parser. Parsers generated by Javacc make decisions at choice points based on some exploration of tokens further ahead in the input stream and once they make such a decision, they commit to it. The process of exploring tokens further ahead in the input stream is termed as "look ahead".

A grammar should not have left recursion in it for Javacc to produce a parser. There are known methods by which left recursion could be eliminated. Also whenever there is ambiguity in the grammar there can be two things that could be done. Either we modify the grammar to make it unambiguous or we insert certain lookahead hints that would enable the parser to make the right choice at the choice points. For simpler grammars changing it to make it unambiguous is certainly a better choice. However when the grammars get complicated the second choice of introducing a lookahead mechanism is a better choice because it makes the grammar readable and easier to maintain without any serious performance hit.

By default Javacc looks ahead one token in the input stream to resolve ambiguity. However more than one token could be specified for look ahead purpose. Javacc provides syntactic and semantic look ahead mechanisms besides the one mentioned above. In syntactic lookahead one particular choice at the choice point is tried out. If that choice does not succeed then the other choices are tried out. With semantic lookahead one could specify any arbitrary boolean expression whose evaluation determines which choice to take at the choice point. For example, consider the following grammar

$$A \rightarrow aBc$$
$$B \rightarrow b\ [c]$$

The above grammar recognizes two strings "abc" and "abcc". The default lookahead mechanism would choose "[c]" from the second production every time it sees a "b" followed by a "c". Thus the Javacc parser with the default lookahead mechanism would not recognize "abc" as a valid string. The second choice should be taken if the next token is "c" and the token after that is not "c". Since this is a negative statement, syntactic lookahead cannot be used in this case. This could easily be expressed using semantic lookahead.

One could specify the global look-ahead in the options section or one could specify local look ahead at the choice points in the grammar file. The former should be avoided as it would hit the performance of the parser. The later is better as the grammar remains LL(1) for most parts and has a better performance.

2.4 TOKEN MANAGER

The “Token Manager” is used to manage the tokens specified in the grammar. It returns the tokens found in the input stream to the parser. Just like the finite automata has a finite set of states, similarly the javacc specification file is organized into a set of lexical states. The “Token Manager” at any moment is in one of the lexical states. Each lexical state has an ordered list of regular expressions. This order is determined from the order in the input file. All the regular expressions in the current lexical state are considered as potential match candidates. The “Token Manager” prefers the longest match possible. If there are multiple longest matches (same length), then the regular expression that is matched is the one with the earliest order of occurrence in the grammar file. Once the regular expression is matched, lexical action associated with that lexical state is executed. In the lexical action one could change the characters thus matched or perform any other processing.

CHAPTER 3

BUILDING PROLOG PARSER USING JAVACC

Prolog which stands for *PROgramming in LOGic* is one of the most widely used language for programming in logic. Prolog is a declarative, relational programming language. It differs from the procedural languages like C in the fact that it is used to describe problems rather than describe algorithms to solve the problem. Prolog describes “what” rather than “how”.

A Prolog program does not contain statements or instructions, rather it contains facts and rules. Facts state the properties that are true of the system we are describing. Rules give us ways of deducing new facts from existing ones. Since the Prolog program gives us information about a system, it is often called as a knowledge base. Working from the knowledge base the Prolog program will then answer “yes” or “no” to our query and provides bindings to variables of the query.

Procedural programming languages contain functions that return a particular answer for a given set of inputs. Relational languages define relations that can return many different answers for one set of inputs. Prolog is relational in the fact that it not only tells us if the relation is true but also lists all the situations that make the relation true.

3.1 PROLOG GRAMMAR

Subset of the Prolog grammar as defined by the ISO standard for Prolog language [7] has been used. The complete Prolog language syntax can be found in [7, 8]. The basic data object of the language is called a “Term”. A “Term” can either be a “Constant”, “Variable” or a “Compound Term”. A “Constant” includes “Numbers” and “Atoms”. A “Number” can either an “Integer” or a “Real”. “Atoms” can be any of the following:

1. Quoted item : An arbitrary sequence of characters enclosed in single quotes. For example ‘USA’, ‘India’, ‘12345’, ‘The Logic’, ‘ !@#$ˆ&*’.
2. Word : A string of characters made up of upper-case letters, lower-case letters, digits, and the underscore character, that begins with the lower-case character. For example _, aBigProgram, a1, java_bean.
3. Special characters : These are defined in the table 3.1 for the token SPECIAL_CHAR.

Variables may be written as any sequence of alpha-numeric characters (including “_”) starting with either a capital letter or an underscore.

Structured data objects of the language are “Compound Terms”. A “Compound Term” comprises of a functor (called the principal functor) of the term and a sequence of one or more terms called the arguments. A “Functor” is characterized by its name which is an “Atom” and its arity or the number of arguments. Thus, an “Atom” on its own is a “Functor” of arity 0.

A logic program simply consists of a sequence of statements called sentences. A “Sentence” comprises of a “Head” and a “Body”. A “Head” consists of a single “Term” or can be empty. The “Body” consists of a sequence of zero or more goals. If the “Head” is not empty then the sentence is called a “Clause”. If the “Body” of the clause is empty then the “Clause” is called a “Unit clause”. If the “Body” of the clause is not empty then the clause is called a “Non- unit clause”.

JAVACC IMPLEMENTATION OF PROLOG SYNTAX

The Prolog language syntax that is implemented is not the entire prolog language set as described by the ISO Prolog standard [7]. The representations of numbers as octal and hexadecimal values has been omitted in the current version. The listing in fig 3.1 uses the Extended Backus Naur format (EBNF) for representing grammar productions which uses the following notation:

1. [...] or (...)? implies 0 or 1 occurrence of anything within the brackets.
2. (...)+ implies 1 or more occurrence of anything within the brackets.
3. (...)* implies 0 or more occurrence of anything within the brackets.

The words with all letters in uppercase denote terminals. Any word with the mixed case letters denotes non-terminals. The brackets (square, curly or parenthesis) when enclosed within double quotes denote grammar tokens otherwise they are a part of the EBNF grammar notation.

Figure 3.1 describes the syntax of Prolog as used for the Javacc input in the EBNF format.

```
program  -> skip_spaces prog EOF

prog  ->  ( sentence skip_spaces )+

sentence  ->  clause  skip_spaces eoc

clause  ->  head  skip_spaces  (  ":-"  skip_spaces  body  )?

head  ->  term1

body -> conjSeparatedTerms

conjSeparatedTerms  ->  term1 skip_spaces  ( "," skip_spaces term1 skip_spaces )*

term1  -> ( term skip_spaces )+

term  ->  compoundTerm  |  list  |  variable  |  atom |  "("  skip_spaces body skip_spaces ")"

variable  ->  VARIABLE

compoundTerm  ->  functor  "("  arguments  ")"

functor  ->  atom

arguments  ->  term1  skip_spaces  (  ","  skip_spaces  term1  skip_spaces  )*

atom  ->  UQ_CONSTANT_STRING  |  Q_CONSTANT_STRING  |  (  SPECIAL_CHAR )+

list  ->  "["  skip_spaces  (  term1 skip_spaces  (  ","  skip_spaces  term1  skip_spaces  ) *

            (  "|" skip_spaces  term1  skip_spaces  )?  )*  "]"

eoc  ->  EOC
```

Figure 3.1: Prolog syntax for Javacc specification file

All the entities in the above productions in capital letters are terminals. These are defined in the “Token Manager” [1] section of the Javacc specification file and their definition is as follows:-

Token	*Definition*
EOF	end of file token
EOC	end of clause “.” (LAYOUT_CHAR)+
LAYOUT_CHAR	“\n” | “\r” | “\t” | “ ”
FLOAT	([“0”-“9”])* “.” ([“0”-“9”])+
INTEGER	[“1”-“9”]
SMALL_LETTER	[“a”-“z”]
CAPITAL_LETTER	[“A”-“Z”]
UNDERSCORE	“_”
TRAIL_STRING	CAPITAL_LETTER |
	SMALL_LETTER |
	INTEGER |
	FLOAT |
	UNDERSCORE
UQ_CONSTANT_STRING	SMALL_LETTER
	(TRAIL_STRING)* |
	(INTEGER | FLOAT)
SPECIAL_CHAR	“+” | “-” | “*” | “/” |
	“\” | “^” | “[” | “]” |
	“=” | “∼” | “:” | “.” |
	“?” | “@” | “#” | “$” |
	“&” | “!” | “[ ]” | “{ }” | “;”

Table 3.1: Lexical tokens

[1]This is the section where the regular expressions that describe the tokens of the language are defined.

The Q_CONSTANT_STRING terminal is represented as a MORE regular expression in Javacc's terminology and its definition is as follows:-

```
MORE:
{
    " ' " : QUOTED
}
<QUOTED>
TOKEN
{
    <Q_CONSTANT_STRING : (~ ["  ' "])* " ' " > : DEFAULT
}
```

If a single quote is encountered in the input stream, it is matched against the single quote in the "MORE" section and enters into a "QUOTED" lexical state. The Token Manager keeps appending all the characters that are matched in this state. The Token Manager exists this state when an ending single quote is encountered in the input stream and enters the "DEFAULT" lexical state. All the appended characters (including the quotes) are returned to the parser as a "Q_CONSTANT_STRING" token.

In the present implementation, both the single line and multi line comments can be recognized. The single line comments start with a "%" sign and multi line comments start with a "*" and end in a "*\". These are matched by the Token Manager as "SPECIAL_TOKEN" regular expression. The special tokens are passed to the parser but are not matched against any production rules.

The following paragraphs explains some of the important details about the grammar productions in the figure 3.1.

skip_spaces :

This is a Javacode production which in Javacc terminology is a black box production.

In the Javacc specification file we could globally specify to skip the white spaces in the "SKIP" section. However, there are certain places in the grammar where the white space decides the kind of token to be matched by the Token manager. The dot "." token when followed by a white space becomes an "end of clause" (EOC) token. For example "f(X). ". However when the dot is not followed by a white space there might be different interpretations for it. The dot might be a decimal point in a floating point number or it might be a functor in the list definition. For example in the number 4.5 the dot is the decimal point and in .(4,5) it is the functor. The skip_spaces non-terminal is used in productions where the white spaces must be ignored.

As stated in the ISO Prolog standard [7] if, in the compound term definition, there is a white space between the atom and the left parenthesis then it would not be interpreted as a compound term. Thus we must not skip spaces here because then +(x,y) and + (x,y) would mean the same thing. In the former term "+" is a functor of arity 2 while in the later it is a functor of arity 1. In further discussion, the skip_spaces non-terminal will be omitted for reasons of clarity.

Lookahead :

Although the Prolog grammar is predominantly LL(1) in nature, there are certain constructs that are not LL(1) in nature. Consider the following set of productions.

$$term \rightarrow compoundTerm \mid atom \mid \ldots$$
$$compoundTerm \rightarrow functor \text{ “(” } term \text{ “)”}$$
$$functor \rightarrow atom$$

From the above set of productions the parser would not be able to decide between an atom and a compound term without looking ahead until it finds a left parenthesis token ("("). Also productions like "$atom \rightarrow (SPECIAL\_CHAR)+$"

require a lookahead of 2. This helps the parser decide whether to match another SPECIAL_CHAR token or to return an atom. Similar argument can be given for the production "$term \rightarrow term(term)*$" where a look-ahead of 2 is introduced. Thus these look-ahead values help in resolving the ambiguities that arise at the various choice points.

Term' :

The production "$term1 \rightarrow term(term)*$" has been introduced to identify the Prolog constructs that contain dynamic operators in operator notation. The introduction of new non-terminal term1 also avoids the left-recursion which is not valid in a recursive descent parser. Consider the valid Prolog sentence "*joe is_a boy.*." Assume that *is_a* is an infix operator with a precedence of 800. Each of the words in the previous sentence is an atom and thus a form of a term. Thus all the Prolog constructs that are in the operator notation is a sequence of terms. The above listed rule with non-terminal *term1* on the left hand side is introduced that identifies a sequence of terms. This sequence of terms is then passed on to the dynamic operator parser module that builds a term tree with the help of operator table that contains the precedence and fixity information of all the operators. In the above example the dynamic operator parser would return *is_a(joe,boy)* to the main parser. This is interpreted by the main parser as a compound term.

Parenthesis have the highest precedence irrespective of the operators contained within them. This situation is handled by the Javacc generated parser. It recognizes the sequence within the parenthesis first and passes them to the dynamic operator parser. After this sequence is transformed into the functional notation it is returned by the dynamic parser to the Javacc generated parser. For example, consider the following Prolog construct *joe is_a (boy is_not tall).* Assume that *is_not* and *is_a* are dynamically defined infix operators with valid precedence values. The sequence of terms that would be passed to the dynamic operator parser first would be *boy is_not tall.* The dynamic operator parser will return *is_not(boy,tall)* to the parser. Thus the parser will have *joe is_a is_not(boy, tall).* This sequence of terms would again be passed on to the dynamic operator parser which would finally return *is_a(joe, is_not(boy, tall))* to the main parser. Thus the parser flips back and forth from the dynamic parser and itself as shown in figure 3.2.

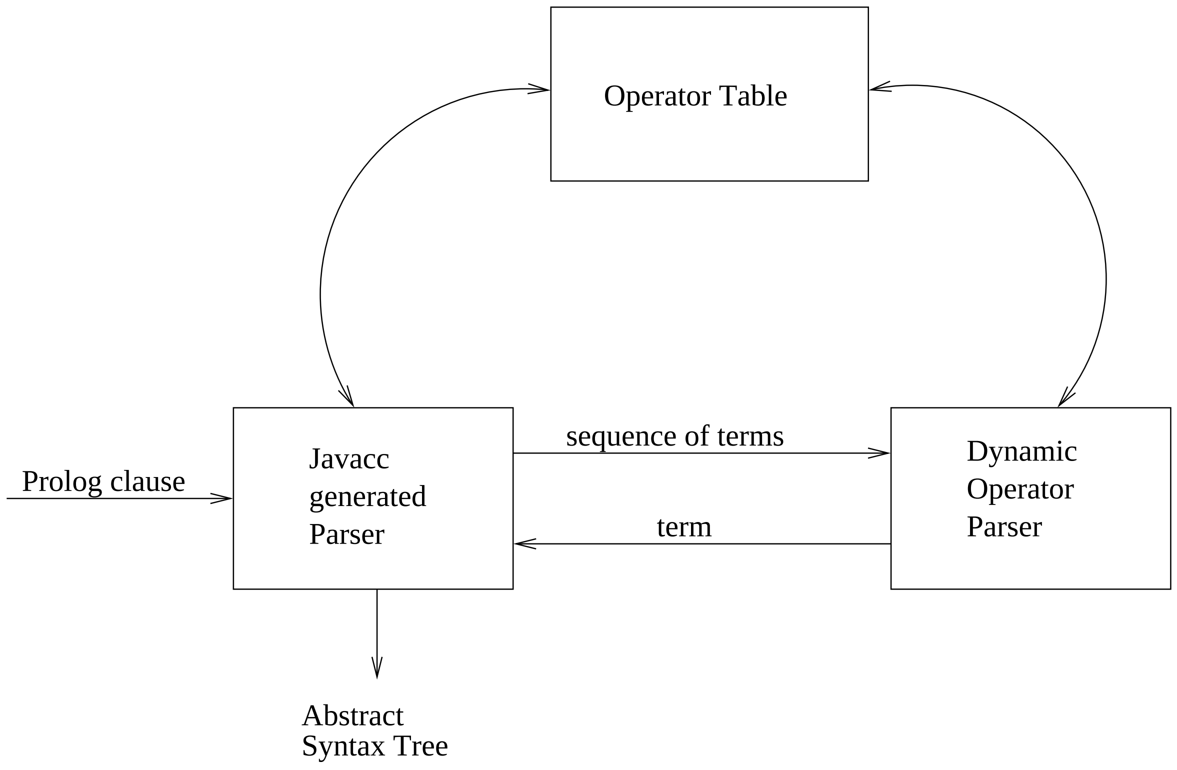

Figure 3.2: Parser

The Javacc generated parser referred to as parser in the remainder of this section, parses and constructs the term tree for operator-less Prolog terms. For constructs that contain dynamic operators which is a sequence of terms as we have seen earlier, the parser passes this sequence to the dynamic operator parser. This term creation process by the dynamic operator parser requires the precedence and associativity rules of the operators contained in it. This information is acquired via the Operator table. The Operator table contains the precedence and associativity values of the predefined operators along with any dynamically defined operators. The Operator table is implemented as a hashtable. The key in the hashtable is the string representation of the operator and the value is an array of 3 operator objects. Each operator object contains the name, fixity and precedence value of the operator. The value of the hashtable is an array of 3 objects for the case of operator overloading, as a single operator can be overloaded as an infix, prefix and a postfix operator.[2] The terms passed on to the dynamic operator parser retain their type information. This helps the dynamic parser in recognizing the dynamic operators in the term sequence. This will be explained further in section 3.2 which deals with building the Abstract Syntax Tree.

The following example shows the sequence of reductions. The part in bold is the part that is getting reduced to the functional notation.

$f(X) :- g(a + \textbf{(b - d / e)}, Y).$

$f(X) :- g(\textbf{a + -(b, /(d,e))}, Y).$

f(X) :- g(+(a, -(b, /(d, e)), Y).

[2]Though ISO Prolog standard allows an operator to be overloaded only as an (infix, prefix) or a (postfix, prefix) combination but not as a (infix, postfix) combination, we still allow an operator to be defined in all 3 fixities. An error is generated when an ambiguity arises during parsing.

$: -(f(X), g(+(a, -(b, /(d, e)), Y)$.

Thus the parser identifies the innermost sequence of terms to be reduced. The dynamic operator parser simply accepts this sequence of terms and returns the term tree to the parser or reports an error if the term tree cannot be constructed.

3.2 ABSTRACT SYNTAX TREE

The context free grammar listed in the previous section is used to verify if a program conforms to the Prolog syntax. In addition to this the parser must also build an internal tree representation of the program. This internal tree representation is called the *Abstract syntax tree*(AST) and is used by the later stages of the compiler. In a recursive descent parser like the one generated by Javacc, the parsing and the generation of the abstract syntax tree are done simultaneously. This means that when a certain group of tokens is reduced by a certain rule, the code associated with that production for building the AST is executed.

The object oriented style of building an abstract syntax tree is to make an abstract class for each of the grammar symbols as stated in [9]. A concrete class is made for each of the grammar rules which extends the abstract class. This rule is more or less followed for designing an abstract syntax tree for Prolog. In Prolog since every data structure is a kind of "term", all the abstract classes due to the grammar symbols get inherited from a universal abstract class "Term". The hierarchy of classes to represent the AST is shown in figure 3.3.

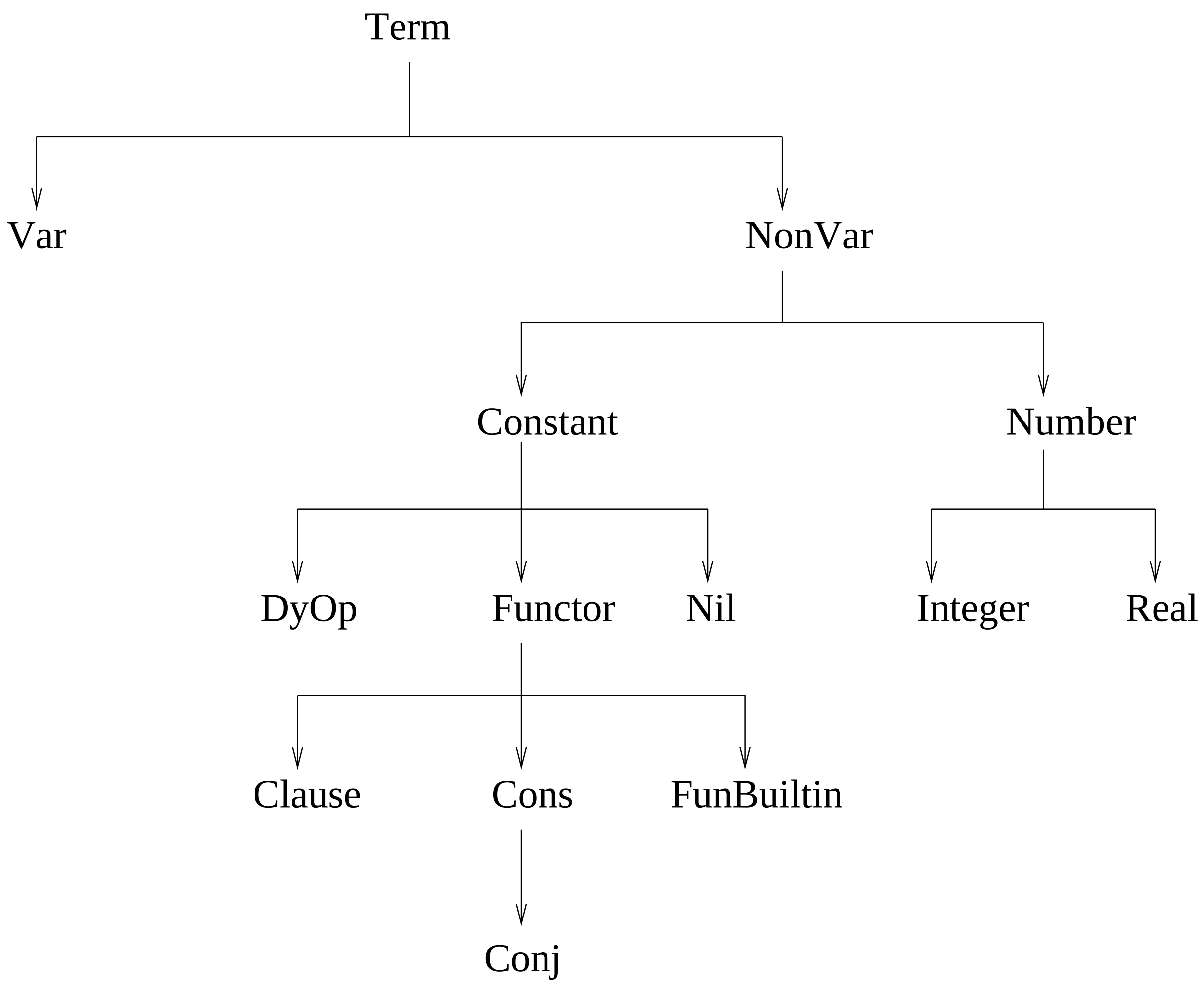

Figure 3.3: Hierarchy of Classes for Building an Abstract Syntax Tree

A *term* can be a *variable*, *constant* or a *compound term*. A *compound term* is represented as *f/n* where *f* is the name of the principle functor and *n* is it's arity (the number of arguments). A *constant* is a functor of arity 0. In other words a *constant* (if not a number) is a special kind of *compound term*. Therefore it is more appropriate to have a more generic *var* and *nonvar* classes which represent the variables and the non-variables respectively, as a subclass of the *term* class. The *nonvar* forms the base class of *constant* and *compound term* classes. As stated earlier, though *numbers* are constants, they are not *functors* of arity zero. Thus instead of having *constant* and a *compound term* as subclasses of the *nonvar* object, *number* and a *constant* form the subclasses of the *nonvar* class. The compound terms are represented by the generic *Functor* class, which inherits from the *Constant* class. There is a special constant term ([ ]) in Prolog that represents the end of the list. This special term is represented by the *nil* class. Thus the *nil* and the *fun* class form the sub-classes of the Const class. The Prolog clauses are special cases of the compound terms whose functor name is ":-" and its arity is the number of body terms in the clause. The lists in Prolog are also special cases of the compound term. The functor name for list is represented by "." and its arity is the number of items in the lists. All the Prolog built-ins are also functors. Thus, there are three classes: *clauses*, *cons* and *FunBuiltin* representing the clauses, lists and built-ins respectively that form the sub-class of the *Functor* class.

Every structure in Prolog is of the form *f(a,b,...)* where *f* is the name of the functor and a, b and so on are the terms. Though the structure of the clause is of the format *head :- body.*, it could also be written as *:-(head,body).*. Thus *:-* is a functor of arity 2 with head and body as its arguments. *cons* is a class that represents a list. Lists are usually represented within square brackets as *[a,b,...]*. They too have an alternate representation using the "." (dot) functor. Thus the list above could also be represented as *.(a,.(b,.(c,nil)))..* The class *cons* takes care of the internal representation of the list. Notice that the inner most dot functor has as its second argument "nil" which represents an empty list ([ ]). All the built-in predicates derive from the class *funbuiltin*.

Following are the functions that are used to generate the abstract syntax tree.

1. makeVar

   Return value: Term

   Argument: String s

   This function is responsible for making a *Var* object out of the string passed to it. This function is called from the production *variable* → *VARIABLE*. This *Var* object is added to a global dictionary that acts like the symbol table. This dictionary holds the identifier name along with the number of occurrences of it in a clause. In the makeVar function, the string s is searched in the dictionary. If the string is present in the dictionary, its occurrence count is increased by 1. If the string is not present in the dictionary, then a new entry is created with an occurrence count of 1. Anonymous variables represented by "_" are treated specially and are not entered in the dictionary.

2. makeConst

   Return value: Term

   Arguments: String s, Boolean checkDynamic

   The boolean value of checkDynamic indicates to the function whether or not to consult the Operator table. If the Boolean argument *checkDynamic* is true, this function first consults the Operator table to find out if string *s* is an operator. If it is an operator then an instance of class *DyOp* is created. If *checkDynamic* is false, this function creates an instance of a *Constant* class. This function is called from the code (Procedure 3.1) that gets executed once the production *functor* → *atom* is matched. The *checkDynamic* boolean variable is a global variable and is set to true when any of the following reductions occur:-

Term → *"(" Term ")"*

Atom → *UQ_CONSTANT_STRING*

Atom → *(SPECIAL_CHAR)+*

3. makeFun

   Return value: Term

   Arguments: String s

   This function is responsible for creating the Functor object out of the string passed to it. This function is called from the production *functor* → *atom.* As stated earlier the *Functor* object represents the functor and thus has an arity associated with it. The arity of the functor is the number of arguments passed to it. The arity of the *Functor* object is set from the production
   argument → *term1(,term1)*.*

4. makeInt

   Return value: Term

   Arguments: String s

   This function is responsible for creating the *Integer* object out of the string passed to it. This function is called from the production *term* → *atom.* As seen from figure 3.1 an *atom* can be a *UQ_CONSTANT_STRING*. This token can be matched against an integer, real or a constant string starting with a lower case letter. The pseudo-code listed in Procedure 3.1 gets executed once the above production is matched. This procedure is used to decide the type of *UQ_CONSTANT_STRING* (Integer, Real or a Const).

Procedure 3.1 Pseudo-code for deciding whether *UQ_CONSTANT_STRING* is an integer, real of a constant string.

```
try {
int is = Integer.parseInt(s);
term = makeInt(is);
} catch(NumberFormatException e) {
isConst = true;
}
if isConst then
   try {
   float fs = Float.parseFloat(s);
   term = makeReal(fs);
   isConst = false;
   } catch(NumberFormatException e) {
   isConst = true;
   }
end if
if isConst then
   term = makeConst(s, checkDynamic);
end if
return term;
```

5. makeReal

   Return value: Term

   Argument: String s

   This function is responsible for creating the Real object out of the string passed to it. This function gets called from the code (Procedure 3.1) that gets executed

Procedure 3.2 Pseudo-code for generating a term tree out of comma separated terms

```
Term curr = (Term)v.elementAt(i);
Term t = null;
i++;
if i >= size then
  t = curr;
  return t;
end if
Term next = (Term)v.elementAt(i);
if next instanceof Const then
  String s = ((Const)next).name();
  if s.equals(",") then
    t = new Conj(curr,makeConjCont(v,++i,size));
  end if
end if
return t;
```

once the production *term* → *atom* is matched. If the string is not an integer then it is checked to see if it is real. If it is found to be real, then the *makeReal* function gets called an a *Real* object is created.

6. makeConjCont

   Return value: Term

   Argument: Vector v, int i, int size

   This function is responsible for creating a tree structure out of the comma separated terms as is present in the body section of the clause. The vector v contains the comma separated terms, size is number of elements in the vector. This is a recursive function and the argument i represents the current index in vector. This function is called from the production *body* → *conjSeparatedTerms*. The pseudo-code listed under Procedure 3.2 is executed once the above rule is matched.

7. makeListCont

   Return value: Term

   Arguments: Vector v, int i, int size

   This function is responsible for creating the tree structure out of the list. The vector contains all the terms of the list, i is the index of the current term in the list and size is the number of elements in the list. This function is called from the production *list* → *[(term (,term)* | term)*].* The pseudo-code listed under procedure 3.3 is executed once the above rule is matched.

Procedure 3.3 Pseudo-code for generating term tree out of the list definition.

```
Term curr = (Term)v.elementAt(i);
Term t = null;
i++;
Term next = (Term)v.elementAt(i);
if next instanceof Const) then
   String s = ((Const)next).name();
   if s.equals("|") then
      t = new Cons(curr,(Term)v.elementAt(++i));
   else if s.equals(",") then
      t = new Cons(curr, makeListCont(v, ++i, size));
   else if s.equals("]") then
      t = new Cons(curr,Const.aNil);
   end if
end if
return t;
```

8. buildDynamicTerms

   Return value: Term

   Arguments: Vector v

   Exception: ParseException

   This function is responsible for creating a term tree out of the Prolog constructs that contains dynamic operators. This function is called from the production *term'* $\rightarrow$ *(term)+*. This term tree building process out of a prolog construct containing dynamic operators will be discussed in detail in section 3.3.

3.3 DYNAMIC OPERATORS IN PROLOG

Prolog language provides us with the ability to define and use dynamic operators. This ability allows us to have a syntax that is more like Natural Language than a programming language. In Prolog an operator could be defined at run-time with its precedence and associativity information. The built-in that is used to define dynamic operator is "op". For example,
we define two arbitrary operators "is_a" and "is_not" as follows:

$$op(500, yfx, is\_a).$$
$$op(600, yfx, is\_not).$$

Then the following statements are valid Prolog rules.

joe is_a boy. and
joe is_not tall.

These two operators can be combined in a single statement as

joe is_a boy is_not tall.

This built-in functor takes three arguments: precedence, associativity and the operator name. The precedence of a dynamic operator is an integer value from 1 to 1200. The higher the numeric value for precedence of an operator, the lower is its precedence. Thus an operator with a precedence value of 500 has a higher precedence than an operator with the precedence value of 600. In the above example, the operator *is_a* will bind more tightly than the operator *is_not*. The statement *joe is_a boy is_not tall.* will bind as *((joe is_a boy) is_not tall.)*. The second argument is the associativity of the operator. The associativity of an operator can have the following values:

1. fx : Unary non-associative prefix operator.
2. fy : Unary right associative prefix operator.
3. xf : Unary non-associative postfix operator.
4. yf : Unary left-associative postfix operator.
5. xfx : Binary non-associative infix operator.
6. xfy : Binary right-associative infix operator.
7. yfx : Binary left-associative infix operator.

Associativity of "yfy" is not a permissible value because an operator cannot be left and right associative at the same time.

A Prolog term in operator notation can be interchangeably used for a Prolog term in the functional notation. Thus the statement "*joe is_a man*" can also be written as "*is_a(joe, man)*". The arity of terms containing operators can be either 1 or 2 as operators can only be unary or binary.

3.4 LL PARSING

LL(k) parsers are top-down or recursive descent parsers which require 1 to k token look-ahead. "LL" stands for left to right parse, leftmost derivation. These parsers scan the input stream of tokens from left to right and try to match them against the terminals of the grammar. LL parsers are easier to understand than the LR parsers (discussed in section 4.2) since the grammar rules get translated to recursive function calls in these parsers, unlike the LR parsers where the grammar rules are encoded in a table. The LL parsers are easier to debug and have a better error recovery semantics. Javacc as discussed in Chapter 1 is an LL(k) parser generator. Left-recursion is not allowed in these parsers as they can go into an infinite loop.

ADAPTABILITY OF LL PARSERS TO DYNAMIC SYNTAX

In this section we will show that it is not feasible to have productions in a LL grammar that correctly identify the infix, prefix or postfix form of operators. A set of production rules is presented that could be used to identify the operators with different fixities. These productions have left recursion in them and so they cannot be used as an input to the LL parser generator. Therefore to make these productions LL grammar compatible, they are modified to eliminate the left recursion in them by the introduction of a new non-terminal. These LL grammar compatible productions will still not be able to identify the operator fixities as will be shown with the help of an example. After this discussion we will present our solution to this problem.

THE PROBLEM

They set of grammar rules that can be used to identify the infix, prefix and postfix operators are as follows:-

1. $term \rightarrow term\ OP\ term$
2. $term \rightarrow OP\ term$
3. $term \rightarrow term\ OP$

In the above productions "OP" represents the dynamic operator. These productions are used to identify the infix, prefix and the postfix operators respectively. There is left recursion in the first and the third productions and thus these productions need to be modified. A new non-terminal term' is introduced for this purpose and the rules are rewritten as follows:-

1. term' → term OP term'

2. term' → OP term'

3. term' → term OP

In the above productions though the left recursion is eliminated, there is ambiguity in the grammar. The right hand side of productions 1 and 3 both start with the same symbols and thus the parser will not be able to choose the correct production. The first and the third productions are combined together as production 4. After this reduction we have the following productions:-

*4. term' → term OP (term')**

5. term' → OP term'

This process is called "left factoring" and is done so that the grammar is unambiguous at the choice point. However with the above reduction we loose the ability to determine the fixity of the matched operator.

Consider the following Prolog construct:-

a && + b.

Here we assume that *&&* is a postfix operator and + is an infix operator. *a* would be returned as a term. *&&* would be matched against the *OP* in rule 4. Next the + symbol would be matched against the OP in rule 5 and *b* would be matched against term' of rule 5. Thus the *&&* operator would be treated as an infix operator when it is actually a postfix operator and + would be treated as an prefix operator when it is actually an infix operator. Given just the productions 4 and 5, it is not possible to correctly identify the fixities of operators. Also in order to apply the correct productions to the input string we must know before hand the fixities of operators. The next section explains a solution to this problem.

THE SOLUTION

Thus to correctly identify the fixities of operators in clauses with operator notation and build a tree, we do the following 3 sub-tasks.

1. Recognize the clauses in the operator notation in the Prolog text.
2. Identify the fixities of the operators in case of overloaded operators.
3. Construct the term tree based on the operators fixities determined in step 2.

SUB-TASK 1:

The sub-task 1 is done by the parser and the sub-tasks 2 and 3 are done by the dynamic parser. Sub-task 1 is done by introducing a new production rule *term'* $\rightarrow$ *term (term)**. Since all operators are atoms by definition which is again a form of term, the production above can recognize the clauses in operator notation. This sequence is passed on to the dynamic operator for further processing.

SUB-TASK 2:

There are certain restrictions that are put forth by the ISO Prolog standard [7] on the overloading of dynamic operators in Prolog. The restrictions are as follows :-

1. Two operators cannot have the same name and fixity.

2. An operator cannot be overloaded as an infix and a postfix operator. This restriction is necessary for the parser to decide the fixity of the operator by just looking ahead 1 token in the input stream. For example, consider the following sequence of terms and operators: *t1 op1 op2 op3 t2.*

   Here t1 and t2 are terms and op1, op2 and op3 are operators. Assume op1 is overloaded as infix and postfix operator and op2, op3 are prefix operators. The fixity of op1 as an infix operator can be determined only until the term t2 is read which is looking ahead more than one token.

The table 3.2 helps us determine the type of fixity of the dynamic operator given the type of adjacent terms. In this table, the columns represent the type of the next term and the rows indicate the type of the previous term. Since the ISO Prolog standard disallows overloading an operator as infix and postfix, the infix and the postfix forms are together referred to as non-prefix.

	Term	DyOp	null
Term	infix	non-prefix	postfix
prefix	prefix	prefix	postfix
infix	prefix	prefix	postfix
postfix	infix	non-prefix	postfix
null	prefix	prefix	postfix

Table 3.2: Operator fixity determination table

The algorithm to decide the fixity of the operator is shown in the algorithm listing 3.4. The variables *prev*, *curr* and *next* point to the previous, current and the next element in the sequence of terms.

Algorithm 3.4 Operator fixity determination algorithm

```
Require: Vector of Terms (vTerms), Operator table (opTable), Fixity table (opFix).
Ensure: Vector of Terms with fixities of dynamic operator's decided.
  n = size of vTerms
  Term prev = null
  Term next = null
  Term curr = null
  for $i$ = 0 to n-1 do
    curr = vTerms[i]
    next = (i+1 <= n−1) ? vTerms[i+1] : null
    if curr is not an instance of DyOp then
      prev = curr
      continue;
    else
      if  DyOp is not overloaded then
        vTerms[i] = corresponding Operator object from the opTable
      else
        fixity = opFix[previous][next]
        vTerms[i] = corresponding Operator object from the opTable
      end if
    end if
    prev = curr
  end for
```

SUB-TASK 3:

The algorithm listing 3.5 describes the algorithm to build a term tree from a sequence of terms. This algorithm is recursive in nature. It finds the operator with the widest scope (highest precedence value) and splits the sequence of terms from that point. This procedure is again applied to the split sequences. Thus the complexity of this algorithm is nlogn.

The algorithm to find the operator with the widest scope is listed in algorithm listing 3.6. This algorithm takes into account the case when there is more than one operator with the same precedence and fixity adjacent to each other. Two operators

Algorithm 3.5 Algorithm to create a term tree that reflects the correct operator associativity and precedence

```
Require: Vector of Terms (vTerms) with correct fixities of operators, sIdx (start
  index of the sub-vector), eIdx (end index of sub-vector)
Ensure: Term tree representing the correct precedence of the operators.
  if sIdx >= eIdx then
    return term
  end if
  int idx = findRootIndex(sIdx, eIdx)
  Operator o = vTerms[idx]
  if o is Prefix then
    right subtree of o = buildTreeBySplitting(idx+1, eIdx)
  end if
  if o is Infix then
    left subtree of o = buildTreeBySplitting(sIdx, index−1)
    right subtree of o = buildTreeBySplitting(idx+1, eIdx)
  end if
  if o is Postfix then
    left subtree of o = buildTreeBySplitting(sIdx, index−1)
  end if
```

are said to be adjacent to each other if there is not a single operator with a different precedence value and fixity separating the two. If there is a sequence of operators adjacent to each other then the determination of the operator with the widest scope depends on the associativity of the operators.

The following examples describe the operators that are adjacent to each other and the operators that are not. The precedence values and fixity of the operators are shown in brackets. For example,

1. op1(N,infix) t1 op2(N,infix)

   op1 and op2 are adjacent to each other.

2. op1(N,prefix) op2(N,prefix) op3(N,prefix)

   All the three operators are adjacent to each other.

3. op1(N,prefix) t1 op2(N-1,prefix) op3(N-1,prefix) t2

   op1 and op2 are not adjacent to each other because they have different precedence values.

4. op1(N,postfix) op2(N-1,infix) op3(N,prefix)

   None of the three operators are adjacent to each other.

5. t1 op1(N,infix) op2(N,prefix) t2

   Although op1 and op2 have the same precedence values, they are not adjacent to each other as one is infix and the other is prefix.

Algorithm 3.6 is described as follows:-

If there is just one operator with the highest precedence then the index of that operator is returned. However, if there is more than one operator with the same precedence value and these operators are adjacent to each other, then the index returned would depend on the associativity of the operators. If they are left associative, then the index returned is the one of the right most operator in the sequence of adjacent operators. If they are right associative then the index returned is of the left most operator in the sequence of adjacent operators. If they are non-associative then an error is reported. If the operators are not adjacent to each other, then the index of the first operator with the highest precedence is returned.

Algorithm 3.6 Agorithm to find the operator with the widest scope

```
Require: Sub-vector of Terms (vTerms) from the buildTreeBySplitting function, sIdx
   (start index of the sub-vector), eIdx (end index of sub-vector)
Ensure: index of the operator with the widest scope (lowest precedence)
  int indexOfRoot = -1
  int maxPrecedence = -1 {The precedence value of the operators are stored as (1200
  - value + 1). This is done so that the precedence of an operator does not have an
  inverse relation with it's precedence value as is the case while defining the operator.}
  for i = sIdx to eIdx do
    curr = vTerms[i]
    if curr is not an instance of Operator then
      continue
    else
      if maxPrecedence < precedence of curr then
        maxPrecedence = precedence of curr
        indexOfRoot = i
        continue
      end if
      if maxPrecedence = precedence of curr then
        rootOperator = vTerms[indexOfRoot]
        if rootOperator and curr are adjacent to each other then
          if associativity of curr = left associative then
            indexOfRoot = i
          end if
          if associativity of curr = right associative then
            continue
          end if
          if associativity of curr = non- associative then
            throw ParseException {Non-associative operators cannot be adjacent
            to each other}
          end if
        end if
      end if
    end if
  end for
```

Example for algorithm 3.4

Input : *a + - b $ * c / e.*

Curr	Operator	Prev	Next	opFix[Prev][Next]
a	No			
+	Yes	Term	Op	non-prefix (Infix)
−	Yes	Infix	Term	Prefix
b	No			
$	Yes	Term	Op	non-prefix (Postfix)
∗	Yes	Postfix	Term	Infix
c	No			
/	Yes	Term	Term	Infix
e	No			

In the table above we use the precedence values of the predefined operators as listed in the ISO Prolog standard [7]. An extra operator "$" has been dynamically added with a precedence value of 100. The final column in the table above contains the evaluated fixity of the operators present in the input string. As discussed before this is obtained by indexing into the table 3.2 with the previous and next types of terms. The fixity information got from the previous query is used to retrieve the correct Operator object from the Operator table. The name of the Operator is used as the key and the fixity is used as an index into the Operator array. If the Operator object cannot be retrieved because an operator with the determined fixity was not defined, then a parse exception is thrown. Thus at this point the fixity and the precedence values of the operators are known. After this step, it is trivial to build a tree representation which will represent the correct precedences of the operators.

In the following line the dynamic operator symbols are annotated with their evaluated fixity and precedence values.

a +(yfx, 500) -(fy, 200) b $(xf,100) *(yfx, 400) c /(yfx,400) e.

The following figure shows the step by step formation of the tree based on the precedence values of the operators. As stated in the algorithm 3.5 we find the operator with the highest precedence value and then split the terms from that point. The operator becomes the functor and the left and right sequences become it's arguments. Note in this case the left or the right split could be empty denoting a prefix or a postfix operator. This process is repeated until no more splits are possible.

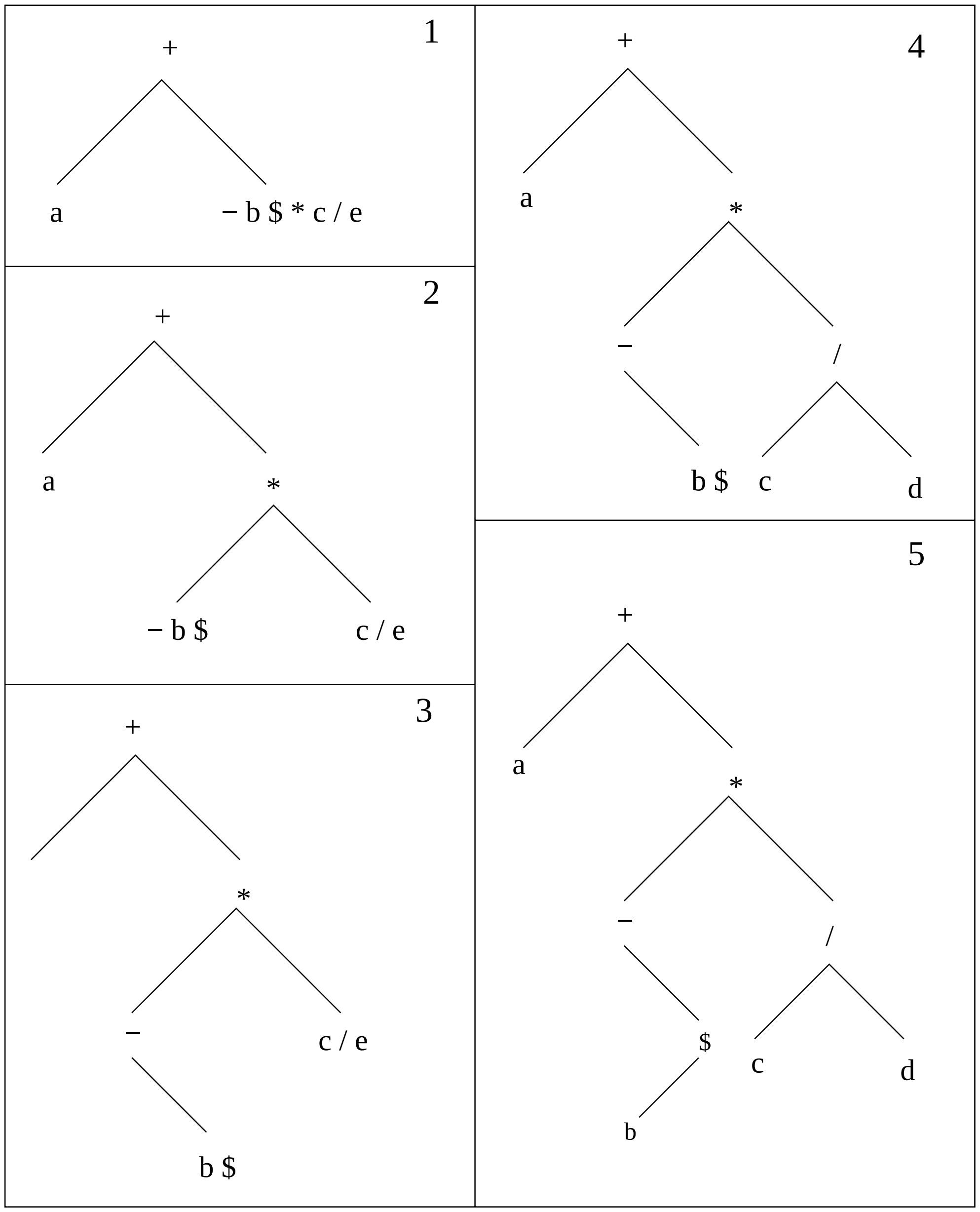

Figure 3.4: Step by step building of the Operator tree based on their precedences

3.5 ERROR REPORTING

Whenever there is a syntax error encountered in the Prolog text, an exception is thrown. This exception gets propagated to the top level production "prog $\rightarrow$ (sentence)+". This production occurs within the try catch block and the exception gets caught here. This exception is called "ParseException" and is provided by Javacc. It contains information about the type of token expected instead of the token encountered at the place of syntax error. It also contains information about the line and column number where the error occurred. Once the exception is caught, the function "skip_to_eoc" is called which as the name suggests skips all the tokens until the end of clause token is reached. Thus the parser recovers from the error, displays the syntax error, skips to the end of the clause and starts parsing the next clause. Following is a listing of some typical errors and messages printed out by the parser.

1. Prolog sentence: *add(X :- a + b.*
   Encountered ":-" at line 1, column 10.
   Was expecting one of:
   ")" ...
   "," ...
   <UQ_CONSTANT_STRING> ...
   <Q_CONSTANT_STRING> ...
   <SPECIAL_CHAR> ...
   "[" ...
   <VARIABLE> ...
   "(" ...

2. Prolog sentence: *a b c d e f.*
   LINE: 1
   SYNTAX ERROR: OPERATOR EXPECTED
   a
   *** HERE ****

b c d e f

3. Prolog sentence: a * ? c.
   LINE: 1
   SYNTAX ERROR: OPERATOR PRIORITY CLASH
   a '*'
   *** HERE ****
   '?' c

4. Prolog sentence: ? ? a
   LINE: 1
   SYNTAX ERROR: NON-ASSOCIATIVE OPERATORS FOUND TOGETHER
   '?'
   *** HERE ****
   '?' a

5. Prolog sentence: a + .
   LINE: 7
   SYNTAX ERROR: UNBALANCED OPERATOR
   a '+'
   *** HERE ****

3.6 EMPIRICAL RESULTS

This section measures the parsing time for the Javacc generated parser for parsing clauses that with and without dynamic operators. These results are then compared to that of Jinni's parser that is hand coded. The expected time complexity of Javacc's generated parser depends on the number of dynamic operators contained in the clause. The general structure of a Prolog clause can be visualized from the following figure.

The dotted line is the part of the clause that contains dynamic operators and the solid line indicates regular prolog terms, that is,. terms in the functional notation

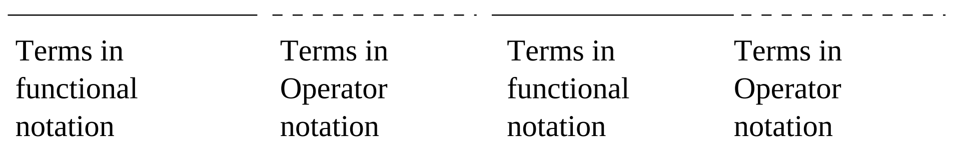
Terms in functional notation
Terms in Operator notation
Terms in functional notation
Terms in Operator notation

Figure 3.5: General structure of a Prolog clause

. The complexity for the parser to parse regular prolog terms is N where N is the number of terms and is k log k for terms that contain dynamic operators where k is the number of operators present in the clause. For small k, k log k tend to k and the overall complexity of the parser tends towards N. However for large k, k log k is significant and the complexity of the parser tends towards N log N. The worst case is when the entire clause is made up of dynamic operators. However, long operator sequences are unlikely and the frequent operators like the "," are hard-coded in the syntax. In the Javacc generated parser, the price for dynamic operator constructs is paid only when they are present and not other wise. This is not the case for some of the hand-coded parsers. There is always an extra overhead introduced in these parsers if they had to account for dynamic operators.

Table 4.1 shows the parsing time information for the Javacc-generated parser. The nTerms column represents the number of terms for the case where the clause contains terms without dynamic operators and it represents the number of operators for the case where the clause contains terms with dynamic operators. The format of the clauses without dynamic operators is a(_) :- b(_) where b(_) is repeated 1 to nTerms times. The format of the clauses with dynamic operators is "a(A) :- A * A + 1". In this case the part "* A + 1" is repeated 1 to nTerms times.

In the following text the parsing time for clauses without operators is referred to as static time and the parsing time for clauses with dynamic operators is referred to as dynamic time.

Number of terms/Operators	*Without operators*	*With dynamic operators*	*nlog n curve*
1	0.979	1.384	0
2	1.098	1.777	2
4	1.248	2.767	8
8	1.924	4.932	24
16	3.128	12.277	64
32	4.904	29.157	160
64	10.612	79.898	384
128	22.016	250.81	896
256	47.125	830.5	2048
512	107.4	3312	4608

Table 3.3: Timing results for Javacc's generated parser

The static time curve as seen in figure 3.6 is bounded by n. This is the time complexity of the Javacc generated parser alone without the dynamic operator parser involved during parsing. The time complexity when there are dynamic operators present in the clause is bounded by nlogn where n is the number of dynamic operators in the clause. This is the expected time complexity as the dynamic operator first goes through all the operators to decide the operator with the widest scope and then divided the clause from that point onwards and again repeats the same process on the divided terms. If we used a stack-based approach for the evaluation of dynamic operators, the complexity would have been n. However, this is an experimental parser, generated from a parser generator tool. A module that has a better time complexity can always replace the dynamic operator module. The focus here was to get dynamic operator capabilities out of a parser that was not hand-coded and get a performance that was closer to or better than that of a hand-coded parser.

The time information in every row is an average of a 1000 iterations of the same

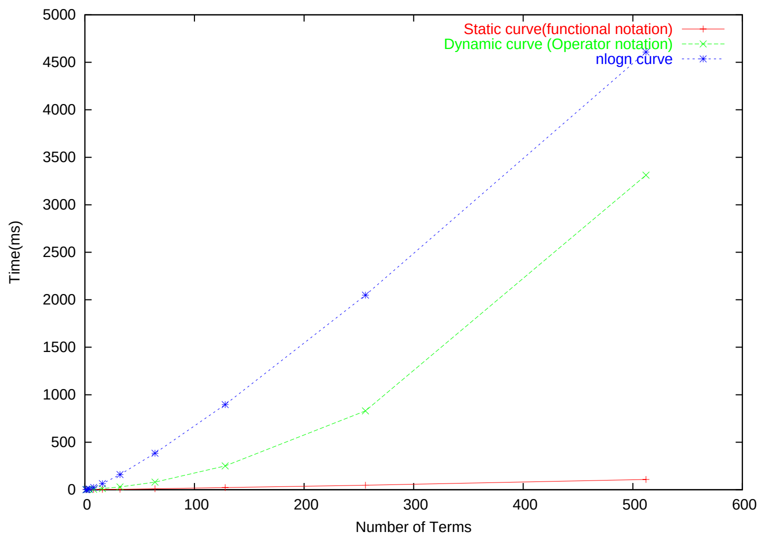

Figure 3.6: Graph showing the parsing time curves of Javacc's generated parser for clauses with (dynamic curve) and without (static curve) dynamic operators

clause. This is to get a more accurate time information so that the operating system factors such as process scheduling, available memory cache, etc are reduced. In calculating the average timings, the first parse time was ignored. This is because it took about 10 times more time to parse the clause first time as compared to the subsequent times. This is attributed to the Java JIT (Just In Time) compiler that introduces an extra overhead of compiling the class files to the native code the first time.

COMPARISON OF JAVACC GENERATED PARSER WITH JINNI'S PARSER

The results in this section compare the Javacc generated parser with that of Jinni's parser. First we compare the two parser's in their ability to parse Prolog clauses that do not contain dynamic operators. Then we compare them in their ability to parse Prolog clauses that contain dynamic operators.

Number of terms/Operators	*Javacc*	*Jinni*
1	0.609	10.5
2	0.834	10.01
4	0.884	10.19
8	1.152	10.38
16	1.668	12.22
32	2.998	12.49
64	4.086	14.83
128	10.934	19.25
256	25.018	30.36
512	61.07	56.78

Table 3.4: Comparison of Javacc's generated parser with Jinni for clauses that do not contain dynamic operators

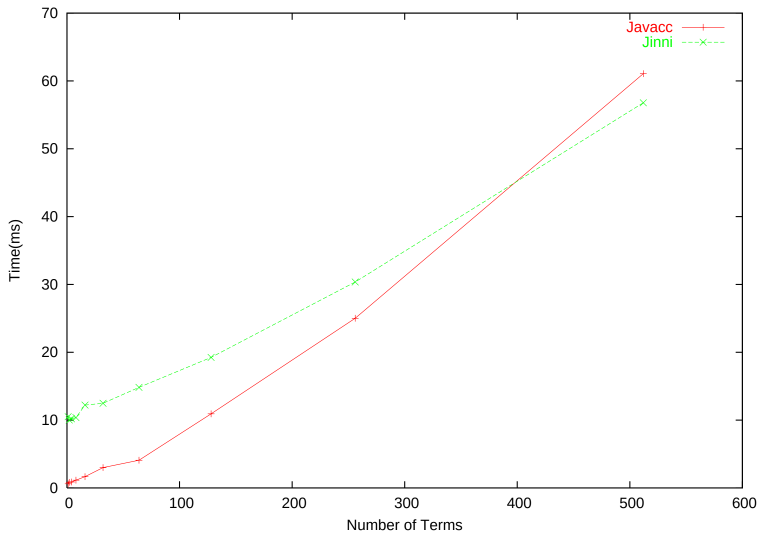

Figure 3.7: Graph that compares Javacc generated parser with Jinni's hand-coded parser for Prolog terms that do not contain dynamic operators

As seen from figure 3.6 the Javacc curve and the Jinni curve are both bounded by n where n is the number of terms. However Javacc takes more time as compared to Jinni at higher nTerms values. This is because Jinni has higher initialization cost as compared to Javacc which are visible at small data sets. Thus the Javacc generated parser performs better than Jinni's parser at smaller data sets.

Number of terms/Operators	*Javacc*	*Jinni*
1	0.979	9.84
2	1.098	9.97
4	1.248	10.00
8	1.924	10.16
16	3.128	10.33
32	4.904	10.94
64	10.612	12.64
128	22.016	15.55
256	47.125	22.22
512	107.4	40.04

Table 3.5: Comparison of Javacc's generated parser with Jinni for clauses that contain dynamic operators

Jinni does not provide support for dynamic operators. However it does have a fixed set of operators that are hard-coded in the grammar syntax. Thus there is no overhead of looking up the operator table, determining the operator's fixity and finally constructing the operator tree based on these results. Hence for Jinni, the static and the dynamic curve are similar as compared to the static and dynamic curves of javacc generated parser.

The comparison's between the generated and hand coded parser were done on flat clause structures of the form "a:-b,c,d,c ...". However the way certain parsers are written may affect their performance when it comes to parsing embedded terms of the format "$a : -b(b(c(d(..)))...$" . The table 3.6 lists the timing information for

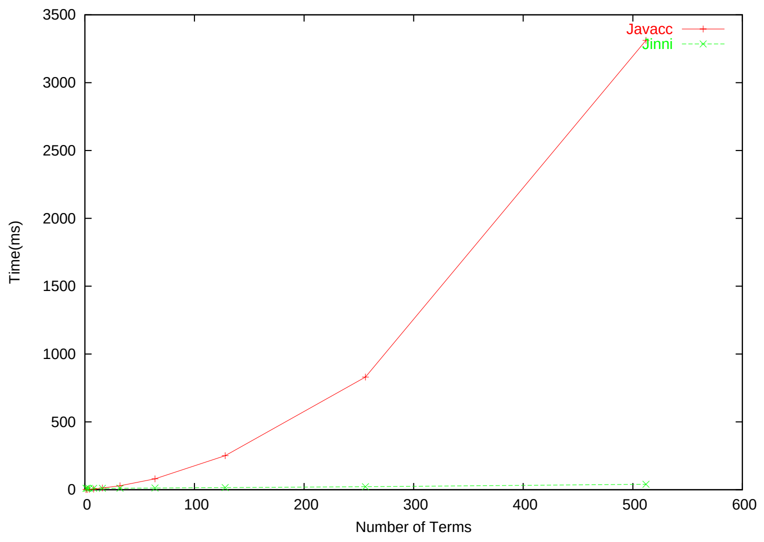

Figure 3.8: Graph that compares the Javacc generated parser with that of Jinni's hand-coded parser for clauses that contain dynamic operator

clause structure having the above form.

As seen from the figure 3.6 the hand coded parser actually does worse than a tool generated parser.

The parser generated with Javacc and enhanced for Dynamic operator capabilities is simple. It is open ended in the sense that a new module that can parse the dynamic operator text more efficiently can easily replace the current one without any changes to the existing parser.

Number of embedded terms	*Javacc*	*Jinni*
1	0.6	6
2	0.7	6
4	0.7	6
8	0.7	7
16	0.8	8
32	0.8	10
64	1.1	12
128	2.3	15
256	4.48	23
512	12.06	48

Table 3.6: Comparison of Javacc's generated parser with Jinni for clauses that have embedded Prolog compound terms

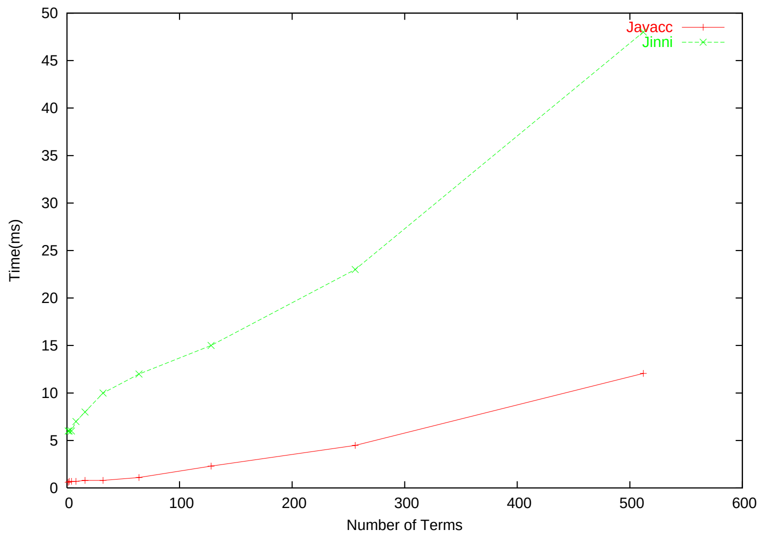

Figure 3.9: Graph showing the comparison of Javacc generated parser with Jinni's hand-coded parser for Prolog clauses that have embedded Prolog compound terms

CHAPTER 4

OTHER PARSING TECHNIQUES AND THEIR ADAPTABILITY TO DYNAMIC SYNTAX

4.1 OPERATOR PRECEDENCE PARSING

Operator precedence parsing is a bottom up parsing technique [10, 9]. The set of grammars for which the operator precedence parsing can be applied have the property that no productions right hand side is empty and no productions can contain adjacent non-terminals.

In operator precedence parsing, there are three disjoint precedence relations: $\doteq$, $<$, $>$ between any pairs of terminals. These relations have the following meaning.

Relation	Meaning
$a > b$	a yields precedence to b
$a \doteq b$	a has the same precedence as b
$a < b$	a takes precedence over b

There are two ways of determining the operator relations between the pair of terminals. The first one is based on the fact that the precedence relations between all the pair of terminals is known. For example, if "$*$" has higher precedence than the "$+$" operator, then we can write "$* > +$" and "$+ < *$" in the parse table. The second method is to construct an unambiguous grammar for the language. This grammar automatically reflects the correct associativity and precedence relationships in the parse tree.

For example, consider the following two representations of a grammar,

Representation 1:

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

From the above grammar representation there is no information about the precedence and associativity of the operators "+" and "$*$" and hence we have to supply to the parser the associativity and precedence information about these operators.

Representation 2:

$$E \rightarrow E + T$$

$$T \rightarrow T * F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

This representation of grammar is identical to representation 1 in the sense that both of them would recognize the same set of strings. However, representation 2 has the precedence and associativity information about the operators "+" and "$*$" embedded in it.

Operator precedence parsing technique is not very conducive to dynamic operator parsing. The parse table can grow very large as new operators can be introduced at run-time. The parse table would have to be rebuild each time a precedence or associativity of an operator is changed. Operator overloading cannot be easily represented with this parsing method [10]. The operator precedence parser would have to rely on the lexical analyzer to return a different token for every overloaded operator [10].

4.2 LR PARSING

LR parser is the most general bottom up parsing method used to construct shift-reduce parsers. “LR” stands for left to right parse, right most derivation in reverse. The following figure explains the LR parsing method. These parsers encode the grammatical knowledge in tables.

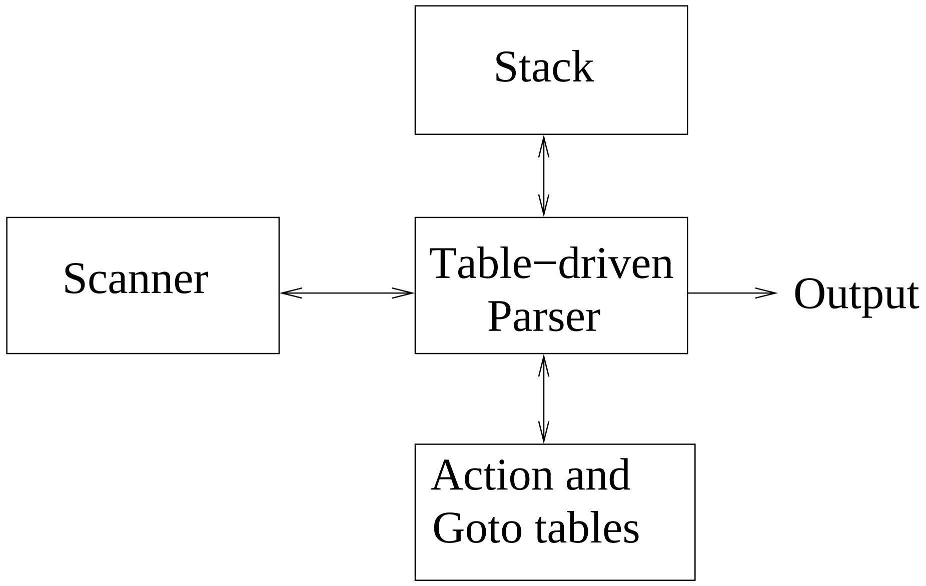

Figure 4.1: LR parser

The scanner provides the parser with the tokens of the language. Every item on the stack has two entities: the state and the symbol. The state represents the current state of the of the non-deterministic finite automata that is used to recognize the viable prefixes of the right sentential form. The symbol can either be a terminal of a non-terminal of the grammar. The parser reads the next token from the scanner and also reads the state on top of the stack. These two values are used as index into the two dimensional parse table. The parse table has two parts: action and goto. There can be four possible actions:

1. *shift n*: “n” denotes the state of the NFA. This action shifts the next input token from the scanner onto the stack and the top of the stack becomes the state n.

2. *reduce n*: “n” denotes the rule by which to reduce. This action means that right hand of a handle is on top of the stack. Thus “2n” symbols are popped from the top of the stack. Let m be the current state on top of the stack. The new state on top of the stack is determined by by goto(m, X). “X” is the left hand side non-terminal of production “n” in the grammar.

3. *accept*: This action signals successful parsing of the string.

4. *error*: This action indicates that the string does not conform to the grammar.

There can be shift-reduce and reduce-reduce type conflicts in an LR parser. A shift-reduce conflict arises when a parse table entry has both shift and reduce actions assigned to it. A reduce-reduce conflict arises when a parse table entry has two different reduce entries in it. The shift-reduce conflict can be eliminated by modifying the grammar or one could favor the “shift” action over the “reduce” action. There is no simple solution for the reduce-reduce conflict.

The shift-reduce type of conflicts can also be resolved if there were precedence and associativity rules between the terminals. The parser would prefer shifting if the precedence of the symbol on the stack was less than that of the next symbol to be read and it would prefer reducing if the precedence of the symbol on the stack was more than that of the next symbol to be read. In case the precedences of the stack symbol and the input symbol are the same then the parser would check the associativities of the symbols. If the symbol is left associative, then the parser would reduce and if it is right associative then the parser would shift.

The above discussion assumed that the precedence and associativity values of symbols are present at compile-time. However, for the case of dynamic operator parsing such information is not present at compile-time. In Prolog, the operators with different precedence and associativities can be defined at run-time. Thus, the shift-reduce conflicts that arise in an LR parser table cannot be resolved. However, for dynamic parsing these conflicts could be left as it is during compile-time and resolved only at run-time. Such a variation of LR parser is called as the "Deferred Decision Parser" and is discussed in the next section.

4.3 DEFERRED DECISION PARSING

Deferred decision parsing [4] postpones the resolution of conflicts involving dynamic operators until run-time. This parser consists of an operator table that stores the operators defined during run-time along with their precedence and associativity. This operator table is available to the parser at run-time to resolve conflicts involving dynamic operators. The scanner returns a special token for dynamic operators. This token is declared to the parser generator and appears in the production rules of the grammar. Given below is a subset of the prolog grammar involving dynamic operators. The first production is used to match prefix operators, the second to match infix and the third to match postfix operators.

$$term(T) \rightarrow op(Name), term(T1)$$

$$term(T) \rightarrow term(T1), op(Name), term(T2)$$

$$term(T) \rightarrow term(T1), op(Name)$$

$$term(T) \rightarrow op(Name)$$

The scanner can identify a dynamic operator token due to the fact that there could be at most one symbol on either side of it. For the above grammar, the parse table has 11 states out of which 4 have shift reduce conflicts.

The shift-reduce conflicts that arise due to the dynamic operators are turned into a new type of action called *resolve* which has two arguments: the state to enter if the conflict is resolved in favor of shift and the rule by which to reduce if a reduction is selected. Conflicts between an operator and a non-operator can be resolved at table construction time because non-operators have a precedence value of 0 and operators have positive precedence values. Thus non-operators take precedence over operators.

The parser driver for *Deferred Decision parser* is similar to that of LR parser except that the entries in the parse table are accessed through the procedure parse_action(S, X) where S is the current state of the parser and X is the look-ahead token. This returns any one of the following actions: shift, reduce, accept or error. The pseudo code for parse_action is as follows:

```
action parse_action {
If parse_table(S,X) = resolve(S', $A \rightarrow \alpha\ op_A\ \beta$)
  then return do_resolve($A \rightarrow \alpha\ op_A\ \beta, X$)
  else return parse_table(S,X)
}
```

The *do_resolve* function can return a shift action or a reduce action. If this function returns a shift action, then the parser would shift the look-ahead token 'X'and if it returns a reduce action then the parser would reduce using the rule $A \rightarrow \alpha\ op_A\ \beta$.

Operator Overloading Issues

Operator overloading poses serious difficulties in the parser design. There could be explicit overloading of operators or there could be implicit overloading of operators. Explicit overloading occurs when the same operator is defined for different fixities. For example, "+" can be defined as a unary and a binary operator. Implicit overloading occurs between the operators and the nullary operators. Nullary operators are nothing but constants.

The nullary operators are defined to have a precedence value greater than the maximum possible value. This guarantees a deterministic grammar if there is no declared overloading and retains the flexibility of allowing operators to appear as terms.

If there is declared overloading then there could be multiple interpretations of a string that contains overloaded operators. Consider the string "$\cdots\ \alpha\ op_A\ \beta\ op_B\ \cdots$" where α and β are symbols of the grammar and op$_A$ and op$_B$ are operators. Since the Prolog grammar does not generate sentential forms with two adjacent expressions, there are only four fixity combinations to consider.

Form of $\alpha\ op_A\ \beta$	*Possible fixity combinations (op_A, op_B)*
$\alpha \neq \epsilon, \beta \neq \epsilon$	(infix,infix), (infix, postfix)
$\alpha \neq \epsilon, \beta = \epsilon$	(infix, prefix), (infix,null), (postfix, infix), (postfix,postfix)
$\alpha = \epsilon, \beta \neq \epsilon$	(prefix,infix), (prefix,postfix)
$\alpha = \epsilon, \beta = \epsilon$	(prefix,prefix), (prefix, null), (null,infix), (null,postfix)

The following rules are evaluated for each fixity combinations and collected in a set.

1. If op_A and op_B have equal scope then

    if op_A is right-associative, shift

    if op_B is left-associative, reduce.

2. if op_A is either infix or prefix, with wider scope than op_B, shift

3. if op_B is either infix or postfix with wider scope than op_B, reduce.

The parser enters an error state if the set is empty - signifying a precedence error - or contains both shift and reduce actions - signifying an ambiguous input. If there is a unique action, then the conflict is resolved.

As stated in [2] this modified version of the LR parser is too powerful to parse Prolog text that conforms to the ISO Prolog standard. Many of the situations that result in ambiguity have been removed from the standard. A parser that is less complex as the deferred decision parser would still be able to parse Prolog text.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The process of building a Prolog parser which is LL in nature out of a parser generator like Javacc is fairly simple. However to get dynamic operator capabilities out of the generated parser is not simple due to the fact that these operators are not present in the grammar of the language.

A Javacc based Prolog parser that can also handle dynamic operators has been designed and implemented. The syntax used was subset of the ISO Prolog grammar. This parser can successfully parse and build abstract syntax tree for Prolog clauses that are either in the functional notation or in the operator notation. Our parser compares well with the hand-coded parsers as it is seen in the section on empirical results. It can be enhanced to cover the complete ISO Prolog syntax. This parser can be used as a front end for any Java based Prolog compiler. The Dynamic operator parser has been designed in a modular way such that a different implementation can be plugged in with ease. Thus by using a parser generator to build a parser for Prolog we have exploited simplicity without compromising much on the efficiency of the parser.

Javacc like parser generators could use our technique to cover languages with dynamic operator capabilities. The Javacc grammar specification could be extended to provide inbuilt dynamic operator support. The functionality of dynamic operator parser could be abstracted such that it becomes language independent.

BIBLIOGRAPHY

[1] P. Tarau, "Inference and Computation Mobility with Jinni", Available on the Internet at http://www.binnetcorp.com/Jinni/index.html.

[2] Koen De Bosschere, "An Operator Precedence Parser for Standard PrologText", *Software - Practice and Experience,* 26(7):763-779, July 1996.

[3] Kjell Post and Allen Van Gelder. Parsing Prolog. Technical Report UCSC-CRL-93-22, University of California, Santa Cruz, 1993.

[4] Kjell Post, Allen Van Gelder, and James Kerr. Deterministic Parsing of Languages with Dynamic Operators. Technical Report UCSC-CRL-91-31, University of California, Santa Cruz, 1991.

[5] Complete documentation for Javacc can be found on the Internet at http://www.webgain.com/products/java_cc/documentation.html

[6] Yacc - Yet another compiler compiler. Technical Report CSTR 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[7] R. S. Scowen, 'Draft prolog standard', *Technical Report ISO/IEC JTC1 SC22 WG17 N110,* International Organization for Standardization, 1993

[8] http://www.cs.bham.ac.uk/~pjh/prolog_course/sicstus_manual_v3_5/sicstus_42.html

[9] Andrew W. Appel. *Modern Compiler Implementation in Java*. Published by Cambridge University Press, ISBN 0-521-588388-8, 1998

[10] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addixon Wesley, ISBN 0-201-10088-6, 1985

[11] A. V. Aho and S. C. Johnson. LR parsing. *Computing Surveys*, June 1974.

[12] A. V. Aho, S.C. Johnson, and J. D. Ullman. Deterministic Parsing of Ambiguous grammars. *Communications of the ACM*, 18(8):441-52, 1975

[13] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2), 1970.

[14] Charley N. Fishcher and Richard J. LeBlanc Jr. *Crafting a Compiler*. Benjamin/Cummings Publishing Company, Inc, 1988.

[15] Jan Heering, Paul Klint, and Jan Rekers. Incremental Generation of Parsers. *IEEE Transactions on Software Engineering*, 16(12):1344-1351, Dec 1990.

[16] R. Nigel Horspool. Incremental Generation of LR parsers. *Computer Languages*, 15(4):205-233, 1990.

[17] James Kerr. On LR Parsing of Languages with Dynamic Operators. Technical Report UCSC-CRL-89-13, UC Santa Cruz, 1989.

[18] W. R. LaLonde and J. des Rivieres. Handling Operator Precedence in Arithmetic Expressions with Tree Transformations. *ACM TOPLAS*, 3(1), 1981.

[19] Masaru Tomita. *Efficient Parsing of Natural Languages.* Kluwer Academic Publishers, Boston, 1986.