

DESIGN AND DEVELOPMENT OF A GRAMMAR ORIENTED PARSING SYSTEM

Devin D. Cook

B.S., California State University, Sacramento, 1997

PROJECT

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

CALIFORNIA STATE UNIVERSITY,

SACRAMENTO

SUMMER

2004

DESIGN AND DEVELOPMENT OF A GRAMMAR ORIENTED PARSING SYSTEM

A Project

by

Devin D. Cook

Approved By:

Du Zhang

Dr. Du Zhang, Committee Chair

8/24/2004

Date

Anne-Louise Radimsky

Dr. Anne-Louise Radimsky, Second Reader

8/24/2004

Date

Student: **Devin D. Cook**

I certify that this student has met the requirements for the format contained in the University format manual, and that this Project is suitable for shelving in the Library and credit is to be awarded for the Project.

Du Zhang (for Dr. Cui Zhang)
Dr. Cui Zhang, Graduate Coordinator

8/24/2004
Date

Abstract

of

**DESIGN AND DEVELOPMENT OF A GENERALIZED
PARSING GENERATOR**

by

Devin D. Cook

Currently, the most common approach used to create parsers is by using compiler-compilers. Compiler-compilers such as YACC and ANTLR allow the developer to integrate the grammar with the source code which defines the actions of the parse. The software then, subsequently, creates a parser program.

However, since each compiler-compiler is designed for a specific implementation language, different parser generators must be written for each language. Most of the common programming languages are supported by one suite or another, but newer languages and specialized languages do not have such suites. As a result, the field consists of several different parser generators with different features, interfaces and grammars.

The goal of this project is to design and implement a parsing system that can support multiple implementation languages and, as a result, create a consistent development platform.

Du Zhang
Dr. Du Zhang, Committee Chair

8/24/2004
Date

ACKNOWLEDGEMENTS

A large number of people have contributed to the completion of this project. First off, I would like to thank my family for their continual support. In particular, I would like to thank my father, Dwight Cook, and my mother, Judy, for being constant positive influences.

I would also like to thank Dr. Du Zhang for his hard work as First Reader. His dedication and generous help with this project is deeply appreciated. I also enjoyed all the discussions we have had about politics, computer science and other various topics. The free lunches were always appreciated too!

I would like to thank Dr. Anne-Louise Radimsky for her hard work as Second Reader. Her keen eye helped me refine my terminology and understanding of parser theory. I am also grateful for the kind words and talks we have had over the years.

I would like to acknowledge Dr. Cui Zhang for her hard work as Graduate Coordinator. Her constant and persistent help - making sure that all the required forms and documents were filed on time - is deeply appreciated.

Finally, I would like to thank all the professors in the College of Engineering and Computer Science and all those professors throughout CSU, Sacramento from whom I have learned so much. Their expertise, hard work, patience and kindness serves as the backbone of an outstanding center of higher learning.

Table of Contents

TABLE OF CONTENTS.....	VI
TABLE OF FIGURES	XIII
TABLE OF TABLES	XVI
1. INTRODUCTION.....	1
1.1. STATEMENT OF PROBLEM.....	1
1.2. PROJECT NAME	2
1.3. CLARIFICATION OF TERMS	3
1.3.1. "Source String".....	3
1.3.2. "Grammar".....	3
1.3.3. "Tokenizer".....	4
1.3.4. "Parser".....	4
1.3.5. "Meta-grammar"	5
1.3.6. "Meta-language"	6
1.3.7. "Source Language".....	6
1.3.8. "Implementation Language".....	6
1.3.9. Engine "Implementation"	7
1.4. THEORETICAL FRAMEWORK.....	8
1.4.1. Regular Expressions.....	8
1.4.2. Context Free Grammars.....	10
1.4.3. Deterministic Finite Automata	14

1.4.4. Parsing Algorithms.....	15
1.4.5. Byte Ordering.....	20
1.4.6. Character Encoding	22
1.5. EXISTING APPROACHES.....	29
1.5.1. Compiler-Compilers.....	29
1.5.2. YACC.....	30
1.5.3. ANTLR.....	35
2. DESIGN	37
2.1. PRIMARY GOAL.....	37
2.2. PROGRAMMING LANGUAGE INDEPENDENCE	37
2.2.1. <i>Separate the Generator from the Actual Parser</i>	37
2.2.2. <i>Use the LALR Algorithm</i>	38
2.3. META-LANGUAGE.....	39
2.4. DESIGN FLOW	41
3. THE BUILDER MODULE	42
3.1. META-LANGUAGE.....	42
3.1.1. <i>Defining Character Sets</i>	42
3.1.2. <i>Pre-Defined Character Sets</i>	44
3.1.3. <i>Comments</i>	49
3.1.4. <i>Defining Terminals</i>	50
3.1.5. <i>Whitespace Terminal</i>	53
3.1.6. <i>Comment Terminals</i>	55
3.1.7. <i>Defining Rules</i>	56
3.1.8. <i>Defining Parameters</i>	59
3.1.9. <i>Examples</i>	65

3.2. PROGRAM TEMPLATES	78
3.2.1. Overview.....	78
3.2.2. Parameters	80
3.2.3. Lists	84
4. COMPILED GRAMMAR TABLE FILE	92
4.1. INTRODUCTION.....	92
4.1.1. Design Considerations	92
4.1.2. Why not XML?.....	93
4.2. FILE STRUCTURE	95
4.2.1. Data Structures.....	95
4.2.2. Records.....	96
4.2.3. Entries	97
4.3. FILE CONTENT.....	100
4.3.1. Parameters	100
4.3.2. Table Counts.....	101
4.3.3. Character Set Table Member	102
4.3.4. Symbol Table Member.....	103
4.3.5. Rule Table Member	105
4.3.6. Initial States.....	106
4.3.7. DFA State Table Member.....	107
4.3.8. LALR State Table Member	109
4.4. EXAMPLE COMPILED GRAMMAR TABLE FILE	111
4.4.1. Example Grammar	111
4.4.2. Table Content	112
4.4.3. File Representation	117

5. THE ENGINE MODULE.....	119
5.1. OVERVIEW	119
5.2. OBJECTS	121
5.2.1. <i>Object Hierarchy</i>	121
5.2.2. <i>Symbol Object</i>	123
5.2.3. <i>Rule Object</i>	125
5.2.4. <i>Token Object</i>	126
5.2.5. <i>Reduction Object</i>	129
5.2.6. <i>GOLDParser Object</i>	131
6. TESTING AND DEVELOPMENT	138
6.1. LALR AND DFA TABLE GENERATION	138
6.1.1. <i>Overview</i>	138
6.1.2. <i>Book: Crafting a Compiler</i>	139
6.1.3. <i>Book: Modern Compiler Implementation</i>	143
6.1.4. <i>Testing Correctness</i>	147
6.2. ENGINE DEVELOPMENT	149
6.2.1. <i>Overview</i>	149
6.2.2. <i>Book: Crafting a Compiler</i>	150
6.2.3. <i>Book: Modern Compiler Implementation</i>	153
6.3. META-LANGUAGE DEVELOPMENT	157
7. IMPLEMENTATION.....	158
7.1. BUILDER APPLICATION	158
7.1.1. <i>Overview</i>	158
7.1.2. <i>Grammar Edit Window</i>	161

7.1.3. <i>Parameter Window</i>	162
7.1.4. <i>Symbol Table</i>	162
7.1.5. <i>Rule Table Window</i>	163
7.1.6. <i>Log Window</i>	164
7.1.7. <i>DFA State Table Window</i>	165
7.1.8. <i>LALR State Table Window</i>	166
7.1.9. <i>Grammar Test Window</i>	168
7.1.10. <i>Exporting the Parse Tables</i>	173
7.2. WEBSITE	188
7.2.1. <i>Introduction</i>	188
7.2.2. <i>Contributors Section</i>	190
7.2.3. <i>Online Documentation</i>	191
7.2.4. <i>Check for Updates</i>	193
8. COMPARISON	196
8.1. INTRODUCTION.....	196
8.2. MATHEMATICAL EXPRESSIONS	198
8.2.1. <i>GOLD Meta-Language</i>	198
8.2.2. <i>YACC / Lex Meta-Language</i>	199
8.2.3. <i>ANTLR Meta-Language</i>	202
9. CONCLUSION.....	204
9.1. RESULTS.....	204
9.1.1. <i>Multiple Programming Language Support</i>	204
9.1.2. <i>Popularity</i>	206
9.2. FUTURE WORK.....	207
9.2.1. <i>Support Additional Platforms</i>	207

9.2.2. Support "Virtual" terminals.....	207
APPENDIX A. ENGINE CODE LISTING.....	209
A.1. INTRODUCTION	209
A.2. CLASSES	210
A.2.1. <i>CompiledGrammarTableFile</i>	210
A.2.2. <i>FAEdge</i>	215
A.2.3. <i>FASState</i>	215
A.2.4. <i>GOLDParser</i>	218
A.2.5. <i>LRAAction</i>	237
A.2.6. <i>LRAActionTable</i>	238
A.2.7. <i>NumberSet</i>	240
A.2.8. <i>ObjectArray</i>	243
A.2.9. <i>Reduction</i>	245
A.2.10. <i>Rule</i>	248
A.2.11. <i>RuleList</i>	250
A.2.12. <i>Stream</i>	253
A.2.13. <i>Symbol</i>	267
A.2.14. <i>SymbolList</i>	270
A.2.15. <i>Token</i>	272
A.2.16. <i>TokenStack</i>	275
A.2.17. <i>Variable</i>	278
A.2.18. <i>VariableList</i>	278
APPENDIX B. EXAMPLE PROGRAM TEMPLATE.....	281
APPENDIX C. GOLD META-GRAMMAR	284

C.1. INTRODUCTION	284
C.2. GRAMMAR	284
APPENDIX D. LALR CONSTRUCTION TEST	289
D.1. INTRODUCTION	289
D.2. ANSI C GRAMMAR IN YACC	291
D.3. SIDE-BY-SIDE COMPARISON	297
APPENDIX E. TEST GRAMMARS	429
E.1. BASIC PROGRAMMING LANGUAGE	429
E.2. ANSI C PROGRAMMING LANGUAGE	433
E.3. COBOL PROGRAMMING LANGUAGE.....	440
E.4. HTML	456
E.5. LISP PROGRAMMING LANGUAGE	460
E.6. SMALLTALK PROGRAMMING LANGUAGE.....	461
E.7. SQL	466
E.8. VISUAL BASIC .NET	471
E.9. XML.....	484
APPENDIX F. ENGINE TEST	486
F.1. TEST PROGRAM.....	486
F.2. PARSE TREE	487
APPENDIX G. CASE MAPPING.....	494
BIBLIOGRAPHY	509

Table of Figures

Figure 1-1. "Parser" Components.....	4
Figure 1-2. Example BNF Rule Defintion.....	11
Figure 1-3. Example DFA Graph	15
Figure 1-4. Rule Handle	17
Figure 1-5. LR / LALR Parse Table	18
Figure 1-6. Big Endian vs. Little Endian.....	21
Figure 1-7. Windows-1252.....	24
Figure 1-8. YACC/Lex Design Flow	31
Figure 1-9. YACC / Lex Program Structure.....	31
Figure 1-10. ANTLR Object Declaration.....	35
Figure 2-1. Two Components & Shared File	38
Figure 2-2. Data Flow	41
Figure 3-1. Set Syntax Diagram	43
Figure 3-2. Commonly Used Character Set Map	46
Figure 3-3. Terminal Syntax Diagram.....	50
Figure 3-4. Regular Expression Syntax Diagram.....	50
Figure 3-5. Rule Syntax Diagram.....	57
Figure 3-6. Parameter Syntax Diagram	59
Figure 3-7. Line Based Grammar Example.....	73
Figure 3-8. Parameter Syntax	80
Figure 3-9. Symbol List Syntax.....	84
Figure 3-10. Rule List Syntax	85

Figure 4-1. General File Structure.....	95
Figure 4-2. Record Contents	97
Figure 4-3. Empty Record Entry	98
Figure 4-4. Byte Record Entry	98
Figure 4-5. Boolean Record Entry.....	98
Figure 4-6. Integer Record Entry.....	99
Figure 4-7. String Record Entry	99
Figure 4-8. Parameter Contents	100
Figure 4-9. Table Count Contents	101
Figure 4-10. Character Set Table Member Contents	102
Figure 4-11. Symbol Table Member Contents	103
Figure 4-12. Rule Table Member Contents	105
Figure 4-13. Initial State Contents.....	106
Figure 4-14. DFA State Table Member Contents.....	107
Figure 4-15. LALR State Table Member Contents	109
Figure 4-16. File Representation of Parse Information	118
Figure 5-1. Parsing Engine Data Flow	119
Figure 5-2. Object Hierarchy.....	122
Figure 5-3. Parse Tree for "a+b*c"	134
Figure 5-4. Reductions That Can Be Trimmed	135
Figure 5-5. Parse Tree For "a+b*c" with TrimReductions	135
Figure 6-1. Diagram from Crafting a Compiler, Figure 6.22, pg 167	140
Figure 6-2. Diagram from Modern Compiler Implementation, Table 3.28 (revised), pg 66.....	144
Figure 6-3. GOLD Builder LALR State Table.....	146
Figure 6-4. Side by Side Comparison.....	146
Figure 7-1. Application Layout	159

Figure 7-2. Grammar Edit Window.....	161
Figure 7-3. Parameter Window	162
Figure 7-4. Symbol Table Window	163
Figure 7-5. Rule Table Window	164
Figure 7-6. Log Window	165
Figure 7-7. DFA State Table Window	166
Figure 7-8. LALR State Table Window	167
Figure 7-9. Grammar Test Window	169
Figure 7-10. Test Window Parse Action List	170
Figure 7-11. Test Window Parse Tree.....	171
Figure 7-12. Export Tables to a Web page	173
Figure 7-13. GOLD Parser Website.	189
Figure 7-14. Contributors Page	190
Figure 7-15. Online Documentation.....	192
Figure 7-16. Check for Updates - Update Available	194
Figure 7-17. Check for Updates - Latest Version.....	195

Table of Tables

Table 3-1. Individual Characters	44
Table 3-2. Commonly Used Character Sets	45
Table 3-3. Unicode Character Sets	47
Table 3-4. Examples of Comment Terminals.....	56
Table 3-5. Character Mapping Table.....	63
Table 4-1. Parameter Content Details	100
Table 4-2. Table Count Content Details.....	101
Table 4-3. Character Set Table Entry Record Details	102
Table 4-4. Symbol Table Member Content Details.....	103
Table 4-5. Symbol 'Kind' Constants	104
Table 4-6. Rule Table Entry Record Details	105
Table 4-7. Initial State Record Details	106
Table 4-8. DFA State Table Member Content Details	107
Table 4-9. Edge 0...n	108
Table 4-10. LALR State Table Member Content Details	109
Table 4-11. Action 0...n.....	110
Table 4-12. 'Action' Constants	110
Table G-1. Case Mapping	494

1. Introduction

1.1. Statement of Problem

When a software engineer designs and writes a program, it is often in one of the many modern programming languages available. Rather than taking on the tedious task of writing the program using the actual instructions used by the computer processor, the logic and behavior of the program are expressed using human-like and English-like terms. Before this program can be compiled or interpreted, the information must be broken down into structures of the language. This process is known as parsing.

Often the process of analyzing a source string is divided into two components which work in tandem. The scanner (also called a tokenizer), performs lexical analysis - breaking the source string into the reserved words, symbols and other atoms of the language. As the information is analyzed by the scanner/tokenizer, a sequence of tokens, which represent the atoms of the source string, are created and passed to the parser. The parser then performs syntactic analysis on the token sequence and determines if it is structurally valid.

Currently, the most common approach used to create parsers is through compiler-compilers. Compiler-compilers allow the developer to integrate the grammar directly with code being used to implement the actual compiler or interpreter. The compiler-compiler then creates a new program that can be subsequently compiled. Often the scanner and parser are integrated, but this is not always the case. The YACC compiler-compiler (Johnson 1979) uses a separate program called Lex to generate a scanner. The

scanner is later compiled with the parser generated by YACC. ANTLR (Parr 2000), on the other hand, generates both the scanner and parser simultaneously.

However, each of these tools is quite different – in both design and usage. As a result, developing a parser in different implementation languages presents a much different experience. The grammar notation used by each parser generator vary greatly in both look and behavior. In addition, how the developer interacts with each tool is different for each.

Since each compiler-compiler is designed for a specific implementation language, different parser generators must be written for each new or different implementation language. Most common implementation languages are supported by one suite or another, but newer languages and specialized languages do not have such suites.

As a result, the field consists of a myriad of different parser generators with different features, interfaces and grammars. For those learning about parsing technology, context free grammars and other language theory related subjects, the inconsistencies between parser generators can present problems.

1.2. Project Name

The project will be called "GOLD" which is an acronym for Grammar Oriented Language Developer. Admittedly, this is not a particularly clever acronym, but it does (in part) represent the history of the greater Sacramento Area.

1.3. Clarification of Terms

1.3.1. "Source String"

The term "source string" will be used abstractly within this document to refer to the text that is analyzed by the parser. This text can be located in a local buffer, string object, file or any other programming language structure available to developers. Hence, the actual "location" of the string will be considered nebulous and not relevant within this document.

1.3.2. "Grammar"

The term "grammar" will be used broadly to refer to the syntax of a language. The grammar of tokens is typically described by regular expressions while the grammar of a programming language can be represented in different forms such as Backus-Naur Form, syntax charts, etc... Even though the format used to describe a grammar can be different, the "grammar" for that language is still the same. The grammar for LISP, for instance, is constant no matter how it is written.

Within this text, the term will also be used to describe the files used to represent the syntax on different development platforms such as GOLD, YACC or ANTLR. Developers on these platforms write "grammars" using the platform's particular notation.

1.3.3. "Tokenizer"

The scanner is used to perform lexical analysis of the source string by dividing it into various pieces of information known as "tokens". These are consequently passed to the parsing algorithm which, using the rules defined in the grammar, performs syntactic analysis and determines when rules are complete.

The scanner is commonly referred to as a "lexical analyzer", "tokenizer" and "lexer". All three of these terms are equivalent and used interchangeably by different texts. This text will use the term "tokenizer" given that emphasis will be placed on the creation of tokens by the system.

Figure 1-1 contains the data flow of a "parser". The unrounded boxes represent different components while the rounded boxes represent input/output flowing through these components.

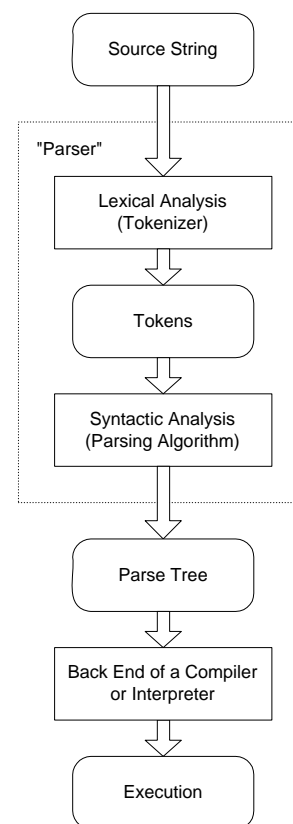


Figure 1-1.
"Parser" Components

1.3.4. "Parser"

The term "parser" is used in many contexts. In most cases, a parser – which performs the syntactic analysis of a token sequence – is used in tandem with the scanner – which creates that sequence. As a result, the term "parser" often is used to refer to both components.

To avoid confusion, the component which performs the syntactic analysis will be referred to as the "parsing algorithm" within the text. The term "parser" will be used abstractly to refer to both the scanner (tokenizer) and the actual parsing algorithm.

1.3.5. "Meta-grammar"

Each of these different parsing systems, such as YACC and ANTLR, use a different notation for describing a grammar. Even though Backus-Naur Form is a widely accepted notation, each parsing system uses a specialized notation to meet its particular needs.

In fact, a grammar written for different parsing systems will use different notations to describe terminals, rules and other information vital to system. Essentially, these different notations have their own syntax, and, in this sense, they also have grammars.

Needless to say, this text will refer often to grammars used to describe programming languages and the grammars used to describe these grammars. This can easily lead to confusion or misinterpretation. To avoid both, the notation that used to describe a grammar will be referred to as a "meta-grammar".

Since each parsing system uses a different format, the term will be further clarified by adding the name of the parsing system. For instance, the syntax used to describe a grammar using YACC will be referred to as the "YACC meta-grammar". For this system, GOLD, the phrase "GOLD meta-grammar" will be used.

1.3.6. "Meta-language"

Although the specifics of different meta-grammars are important, the actual semantics – how the meta-grammar is interpreted – is equally important. For instance, the %token" tag in YACC defines a very specific aspect on how the text will be interpreted. The content of this information will greatly change the meaning of the YACC grammar file.

When it is necessary to refer to both the syntax (meta-grammar) and the associated semantics, the term "meta-language" will be used. Basically, the syntax of a meta-language is specified using a meta-grammar.

1.3.7. "Source Language"

Often in the text, it will be necessary to distinguish between the language being created by the developer, the broad topic of "programming languages", and the programming language being used to develop the compiler and/or interpreter. To avoid confusion, the term "source language" will refer specifically to the language being designed.

1.3.8. "Implementation Language"

The term "implementation language" will refer specifically to the host programming language being used to develop the interpreter and/or compiler. In other words, if the developer is creating a programming language "X" using C++ to parse and compile "X", the implementation language is C++.

1.3.9. Engine "Implementation"

The Engine component of the GOLD parsing framework can be implemented in different programming languages and for different integrated design environments (IDEs). The exact nature of the Engine is described in Section 2.

To distinguish between, for instance, an Engine written to work with C++ and another designed to work with Visual Basic, the term "implementation" will be used. In other words, a C++ Implementation of the Engine can be created as well as a Visual Basic Implementation.

1.4. Theoretical Framework

1.4.1. Regular Expressions

A Regular Expression is a simple, yet powerful, notation that is used to represent simple patterns. They are used extensively in programming language theory. In particular, Regular Expressions are used to describe the "terminals" of a programming language. The term "terminal" refers to the reserved words, symbols, literals, identifiers, etc... which are the basic components of a programming language.

A set of identifiers such as "Student", "Test" and "Stress" are in all the same category of terminal – an identifier. Even though the individual meaning of each identifier varies, each represents the same type of data and, consequently, has an same effect on the syntax of a program.

When a program is analyzed, the text is chopped into different logical units by the scanner. The scanner produces a number of "tokens" which contain the same information as the original program. Of course, the scanner has the ability to ignore information such as comments. While terminals are used to represent the classification of information, tokens contain the actual information. Essentially, the category of token is its associated terminal.

For instance, the identifiers "Student", "Test" and "Stress" are different tokens since they contain actual information. On the other hand, each is the same type of token – an identifier terminal.

Terminals are typically recognized by using the pattern of the information. Traditionally, identifiers consist of a letter followed by a series of zero or more alphanumeric characters. Various programming languages use variations of this scheme, often allowing the use of underscores or dashes.

Regular expressions are used to describe these kind of patterns. The notation consists of expressions constructed from a series of characters. Sub-expressions are delimited by using parenthesis '(' and ')'. The vertical-bar character '|' is used to denote alternate expressions. Any of these items, can be followed by a special character that specifies the number that can appear in sequence.

*	Kleene Closure. This symbol denotes 0 or more of the specified characters or expressions
+	One or more. This symbol denotes 1 or more of the specified characters or expressions
?	Optional. This symbol denotes 0 or 1 of the specified characters or expressions

For example, the regular expression ab^* translates to "an a followed by zero or more b 's". Examples include: a , ab , abb , $abbb$, etc.... The regular expression $(a|b|c)^+$ translates to "a series of one or more expressions where each expression can be an a , b or c ". Examples include: abb , $bcaac$, $ccba$, etc...

Many scanner generators and parsing systems have expanded the notation to include set literals and sometimes named sets. In the case of Lex, literal sets of characters are delimited using the square brackets '[' and ']' and named sets are delimited by the braces '{' and '}'. For instance, the text "[abcde]" denotes a set of characters consisting of the first five letters of the alphabet while the text "{abc}" refers to a set named "abc". This type of notation permits a short-cut notation for regular expressions. The expression $(a|b|c)^+$ can be defined as $[abc]^+$.

1.4.2. Context Free Grammars

Grammars provide rules that specify the structure of languages, independently from the actual meaning of the content. Grammars are classified according to the complexity of the structure they describe. The class of context free grammars (CFG) is the most common one use to describe the syntax of programming languages. In this class, the category a token belongs to (e.g. reserved words, identifiers, etc.) is what matters rather than the specific token (e.g. the identifier xyz).

In addition, the formatting of the program (the content of whitespace) and the actual text of identifiers does not affect the syntax of the grammar. This is very important in parsing technology. Grammars that are not context free cannot be parsed by the LR, LALR or LL parsing algorithms.

1.4.2.1. Backus-Naur Form

Backus-Naur Form (Fischer 1988), or BNF for short, is a notation used to describe context free grammars. The notation breaks down the grammar into a series of rules - which are used to describe how the programming languages tokens form different logical units

The actual reserved words and recognized symbol categories in the grammar represent "terminals". Usually, terminals are left without special formatting or are delimited by single or double quotes. Examples include: if, while, '=' and identifier.

In Backus-Naur Form, rules are represented with a "nonterminal" - which are structure names. Typically, nonterminals are delimited by angle-brackets, but this is not always the case. Examples include <statement> and <exp>. Both terminals and nonterminals are referred to generically as "symbols". Each

nonterminal is defined using a series of one or more rules (also called productions). They have the following format:

$$\boxed{N ::= S}$$

where N is a nonterminal and s is a series of zero or more terminals and nonterminals. Different alternatives for rules can be specified in Backus-Naur Form. For readability, often productions are grouped together and separated by a vertical bar symbol which is read as the word “or”.

In summary, there are slight variations in use, but the notation has the following properties.

- A rule / production starts with a single nonterminal.
- This nonterminal is followed by the symbol $::=$ which means “is defined as”. The $::=$ symbol is often used interchangeably with the \rightarrow symbol. They both have the same meaning.
- The symbol is followed by a sequence of terminals and nonterminals.

The following chart identifies the various parts of a rule definition.

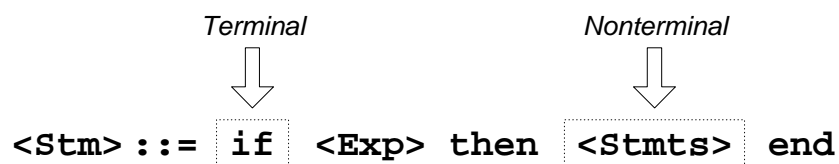


Figure 1-2. Example BNF Rule Definition

For example, the following defines a rule for **<Value>** that can contain either an Identifier terminal or a nonterminal **<Literal>**

```

<Value> ::= Identifier | <Literal>
<Literal> ::= Number | String

```

The <Literal> rule can contain either a `Number` or a `String` terminal. As a result of this definition, a <Value> can lead to an Identifier, a Number or a String.

Rules can also be recursively defined. The following rule defines a sequence of one or more Identifiers.

```

<Identifiers> ::= Identifier <Identifiers>
                | Identifier

```

1.4.2.2. Extended BNF

There is another version of BNF called Extended BNF, or EBNF (ISO/IEC 14977), for short. This variant was originally developed by Niklaus Wirth to define the syntax for the Pascal Programming Language. The notation was designed to simplify the notation of BNF by allowing the developer to use special notation for defining lists and optional sets of symbols.

Variations between different versions of EBNF exist, but most use similar notation. Square brackets "[...]" are used to denote optional elements of a rule. Elements of a rule can also be grouped together using braces "{ ... }" which denotes a repetition of zero to infinity. Symbols can also be grouped using parenthesis "(...)" and followed by a Kleene closure. In this case, the semantics are identical to those used in Regular Expressions.

This format, while powerful, creates a number of implied rules. For instance, if the programmer was to define a rule with an optional clause, the system would have two distinct forms of the rule - the one with the clause and one without. This is also the case with lists and other enhanced features. For instance, the If-then-else statement could be defined as:

```
<If Stm> ::= IF <Expression> THEN <Stms> [ ELSE <Stms> ]
```

This would create the following rules:

```
<If Stm> ::= IF <Expression> THEN <Stms>  
           | IF <Expression> THEN <Stms> ELSE <Stms>
```

1.4.3. Deterministic Finite Automata

Most parser engines implement the scanner as a Deterministic Finite Automaton (Louden 1997). The Lex scanner generator is one example. The scanner scans the source string and determines when and if a series of characters can be recognized as a token.

Essentially, regular expressions can be used to define a regular language. Regular languages, in turn, exhibit very simple patterns. A deterministic finite automaton, or DFA for short, is a method of recognizing this pattern algorithmically.

As the name implies, deterministic finite automata are deterministic. This means that from any given state there is only one path for any given input. In other words, there is no ambiguity in state transition. It is also complete which means there is one path from any given input. It is finite; meaning there is a fixed and known number of states and transitions between states. Finally, it is an automaton. The transition from state to state is completely determined by the input. The algorithm merely follows the correct branches based on the information provided (Cohen 1991).

A DFA is commonly represented with a graph. The term "graph" is used quite loosely by other scientific fields. Often, it refers to a plotted mathematical function or graphical representation of data. In computer science terms, however, a "graph" is simply a collection of nodes connected by vertices.

The figure below is a simple deterministic finite automaton that recognizes common identifiers and numbers. For instance, assume that the input contains the text "gunchy". From State 1 (the initial state),

the DFA moves to State 2 when the "g" is read. For the next five characters, "u", "n", "c", "h" and "y", the DFA continues to loop to State 2.

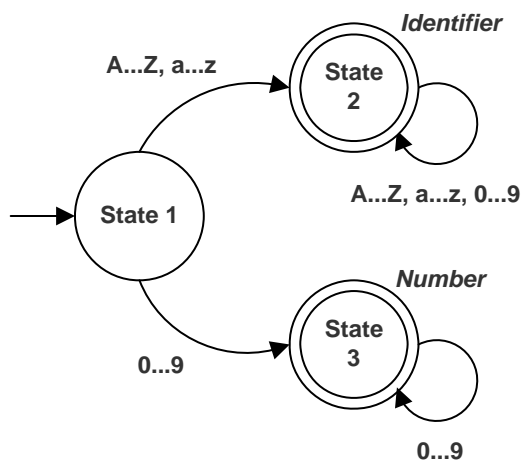


Figure 1-3. Example DFA Graph

By design, the scanner attempts to match the longest series of characters possible before accepting a token. For example: if the scanner is reading the characters "count" from the source, it can match the first character "c" as an identifier. It would not be prudent for the scanner to report five separate identifiers: "c", "o", "u", "n" and "t". Each time a token is identified, the scanner restarts at the initial state.

1.4.4. Parsing Algorithms

The primary goal a parser is to organize a sequence of tokens based on the rules of a formal language. As the parser accepts a sequence of tokens, it determines, based on this information, when the grammar's respective rules are complete and verifies the syntactic correctness of the token sequence. The end result of the process is a "derivation" which represents the token sequence organized following the rules of the grammar.

Typically, Backus-Naur Form is used to define the context free grammar used by the language. The entire language, as a whole, is represented through a single nonterminal called the "start symbol". Often the parse information is stored into a tree, called a derivation tree, where the start symbol is the root node.

There are two distinct approaches currently used to implement parsers. Recursive Descent Parsers and LL parsers are examples of top-down parsers and LR parsers (Appel 1998) are examples of bottom-up parsers. Most parser generators, such as YACC, use one of the LR algorithm variants.

1.4.4.1. LR / LALR Parsing

LR Parsing, or Left-to-right Right-derivation parsing, uses tables to determine when a rule is complete and when additional tokens must be read from the source string. LR parsers identify substrings which can be reduced to nonterminals. Unlike recursive descent parsers, LR parsers do very little "thinking" at runtime. All decisions are based on the content of the parse tables.

LR parser generators construct these tables by analyzing the grammar and determining all the possible "states" the system can have when parsing. Each state represents a point in the parse process where a number of tokens have been read from the source string and rules are in different states of completion. Each production in a state of completion is called a "configuration" and each state corresponds to a configuration set. Each configuration contains a "cursor" which represents the point where the production is complete.

LALR Parsing, or "Lookahead LR parsing", is a variant of LR Parsing which most parser generators, such as YACC, implement. LR Parsing combines related "configuration sets" thereby limiting

the size of the parse tables. As a result, the algorithm is slightly less powerful than LR Parsing but much more practical.

Grammars that can be parsed by the LR algorithm might not be able to be parsed by the LALR algorithm. However, this is very rarely the case and real-world examples are few. The number of states eliminated by choosing LALR over LR is sometimes huge. The C programming language, for instance, has over 10,000 LR states. LALR drops this number to around 350.

Typically, the LR / LALR parsing algorithms, like deterministic finite automata, are commonly represented by using a graph – albeit a more complex variant. For each token received from the scanner, the LR algorithm can take four different actions: Shift, Reduce, Accept and Goto.

For each state, the LR algorithm checks the next token on the input queue against all tokens that expected at that stage of the parse. If the token is expected, it is "shifted". This action represents moving the cursor past the current token. The token is removed from the input queue and pushed onto the parse stack.

A reduce is performed when a rule is complete and ready to be replaced by the single nonterminal it represents. Essentially, the tokens that are part of the rule's handle – the right-hand side of the definition – are popped from the parse stack and replaced by the rule's nonterminal plus additional information including the current state in the LR state machine.

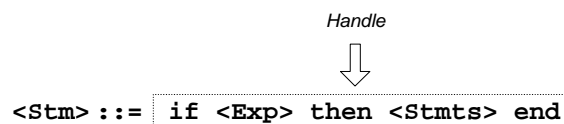


Figure 1-4. Rule Handle

When a rule is reduced, the algorithm jumps to (gotos) the appropriate state representing the reduced nonterminal. This simulates the shifting of a nonterminal in the LR state machine.

Finally, when the start symbol itself is reduced, the input is both complete and correct. At this point, parsing terminates.

State	Production/Action	Lookahead
0	<S'> ::= <S> {EOF}	
	<S> ::= '(' <L> ')'	EOF
	<S> ::= x	EOF
	<S> Goto 1	
	'(' Shift 2	
	x Shift 8	
1	<S'> ::= <S> {EOF}	
	{EOF} Accept	
2	<S> ::= '(' <L> ')'	EOF) ,
	<L> ::= <S>) ,
	<L> ::= <L> ',' <S>) ,
	<S> ::= '(' <L> ')') ,
	<S> ::= x) ,
	<S> Goto 3	
	'(' Shift 2	
	<L> Goto 4	
	x Shift 8	

Figure 1-5. LR / LALR Parse Table

1.4.4.2. Top-Down Parsing

Top-down parsers analyze the token sequence by tracing the left-most derivation of the grammar (Loudan 1997). Generally speaking, there are two major implementations of top-down parsers: recursive decent parsers and LL parsing.

Recursive decent parsers, as the name implies, use a recursive algorithm to trace different possible derivations based on the token sequence. Of all the different approaches used in parsing, this technique is by far one of the most straight forward and easy to implement manually (without the use of parser

generators or development suites). Essentially, the approach employs a procedure for each nonterminal in the grammar. Each procedure, in turn, is designed to read the contents of productions it represents. The procedure will either read a token from the input queue or recursively call another procedure. For instance, if a grammar contains the following production:

$$\langle \text{If-Stm} \rangle ::= \text{if } \langle \text{Exp} \rangle \text{ then } \langle \text{Stms} \rangle \text{ end}$$

the system will contain a procedure to handle the "if-statement". That procedure will read the "if" terminal, call the procedure that handles $\langle \text{Exp} \rangle$, read the "then" terminal, call the procedure that handles $\langle \text{Stms} \rangle$ and then, finally, read the "end".

The second major approach to top-down parsers is LL Parsing, or Left-to-right Left-derivative parsing. LL parsing, unlike recursive decent parsers, does not use recursion to trace different possible derivations for the token sequence. Instead, the algorithm uses a stack and pushes and pops the members depending on the next tokens in the input queue.

Essentially, the LL parsing algorithm analyzes all the productions with the same nonterminal that is on the top of the stack. The goal is to replace the nonterminal with the handle of production and continue the process until only terminals remain. Tables are constructed such that the algorithm knows when, depending on the next token on the input queue, which production to use.

Unfortunately, top-down parsers are not as powerful as bottom-up parsers. Productions cannot have left-recursion since it causes problems with the left-most derivation. In addition, given the "trace" nature of algorithm it is less efficient than bottom-up parsers. Optimization is possible, but the complexity of the top-down runtime engine increases dramatically.

1.4.5. Byte Ordering

Computers use data structures of all configurations. They range from the simple "byte" to 32-bit integers to those both complex and abstract. In most cases, the data structure consists of more than one byte. These bytes must be stored in memory and, sometimes, to files. However, the question remains: "When multiple bytes are used, what order are they stored in memory?" This is a fundamental issue that must be known with absolute certainty, or the reading and writing of binary data could be interpreted incorrectly.

Two different classes of byte ordering are used in computers: Big-Endian and Little-Endian (Ganssle 2003). When a data structure is stored in Little Endian format, the least significant byte is stored in the lower memory address. The opposite is true for Big Endian, the most significant byte is stored in the lower memory address.

For instance, assume you have a 32-bit integer containing the value $D7C4_{16}$. The least significant byte contains the value $C4_{16}$ and the most significant byte contains $D7_{16}$. If the number is stored using Little Endian byte ordering, the first byte in the file will contain the value $C4_{16}$; the second: $D7_{16}$. This is the format used on the Intel family of processors and is the standard used by most file formats.

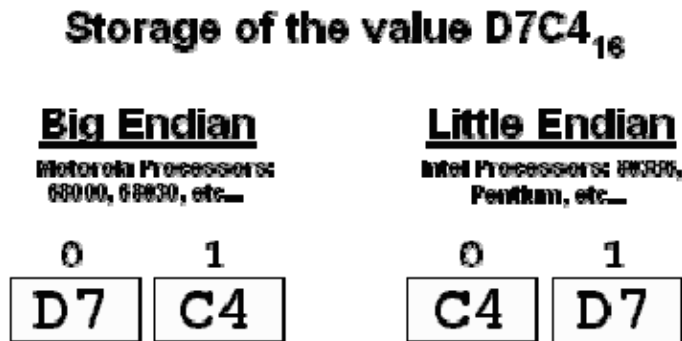


Figure 1-6. Big Endian vs. Little Endian

At first you might consider the Little Endian format "backwards", but this is not the case. In Little Endian, the low memory byte is stored in the lower memory location in the file - what we would logically assume would happen. The "backwards" appearance of Little Endian is not a result of the order of the bytes, but instead from differences in the notation commonly used to represent numbers and files. In fact, it is these representations that are reversed.

Typically, hexadecimal numbers are printed with the high-order bit on the leftmost position; the rightmost bit being the least significant. In storage terms, the higher byte is displayed first; this being the case with the value $D7C4_{16}$. In other words, hexadecimal and binary numbers are normally displayed in descending order. On the other hand, when displaying the binary information in files, the low-order byte of the file is displayed first. In this case, files are displayed in ascending order.

This reversal of notations is what gives Little Endian the appearance of being "backwards".

1.4.6. Character Encoding

1.4.6.1. ASCII

One of the most common storage formats is ASCII - American Standard Code for Information Interchange.

ASCII uses a total of 7 bits to store each character code - giving a total of 128 possible values. The first 32 characters of ASCII are reserved for control characters such as Line Feed, Carriage Return, and Form Feed (new page). Character #127 is interpreted as "delete" and was used in the past to delete information stored on tape and punch card media.

In many cases, different companies have expanded the set to the full 8 bits available in the byte - creating a total of 256 possible values. However, the values from 128 to 256 varied greatly from company to company. For instance, in the early 80's each computer platform used the 128-256 range for its own particular needs. The IBM-PC used the extra spaces for creating boxes and various mathematical symbols. The Commodore 64 used the extra space for displaying inverse versions of the standard characters.

Although ASCII only contains 128 characters (only 95 not counting the control characters), it is sufficient for the English language.

1.4.6.2. ISO 646

ASCII, although sufficiently expressive for English, lacks many of the characters found in other world languages such as German and Dutch. Many languages contain accented versions of normal Latin characters and additional characters not found in English. ASCII simply did not have the space to store these codes.

To help resolve this issue, the International Standards Organization (ISO), in 1972, created the 646 specification (ISO/IEC 646). This specification defined a number of variants of the ASCII character set to use in different world states. The number of bits in the ISO-646 character set is still 7-bit. To handle the additional characters, many of the symbols in ASCII were replaced. For instance, in the Dutch version, 646-DK, the backslash character was replaced with Ö. 646-US is identical to US-ASCII.

For the most part, each version is compatible with one another. However, problems can arise if information was passed between different sets. For instance, if a 646-US text file containing the "[" character is moved to a 646-DE (German) system, the character will change to "Ä".

1.4.6.3. ISO 8859

The first attempt to resolve this issue was in 1987 by International Standards Organization (ISO). The ISO 8859 specifications (ISO/IEC 8859) were not designed to create a single uniform character set, but to avoid the incompatibility of ISO-646. To accomplish this, rather than just use the first 7 bits of each code, the 8859 character set was expanded to 8 bits - giving a total of 256 total codes.

The codes between 0 and 127 were set to the same values in US-ASCII. This allowed easy portability of text - given that the lower 7-bits would be identical regardless of platform. The codes from 128 to 256, however, were specialized for different languages. While the first 128 codes would overlap between languages, the remaining 128 codes would not.

ISO created a total of 16 different sets between 1987 and present time. The chart on the right contains each of the ISO 8859 sets along with its primary and secondary names. The 8859-12 set was

rejected by the organization and numbering continued at 13. ISO 8859-16 was a revision of Latin-1 (a.k.a. "Western"). Various characters were replaced with those in higher demand such as the Euro.

1.4.6.4. Windows-1252

Microsoft modified the ISO 8859-1 character set to use in its Windows Operating System. The characters between 128 and 159, which beforehand contained control characters, were modified to contain commonly needed characters. These characters included: ÿ, the Euro symbol €, and the trademark symbol ™. This set is commonly, and vaguely, referred to as "ANSI".

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	€		,	f	„	...	†	‡	^	%	Š	<	Œ			
90		\	'	“	”	•	—	—	~	™	š	>	œ			ÿ

Figure 1-7. Windows-1252

Essentially, Windows-1252 is a superset of ISO 8859-1. Windows versions 3.1, 95, 98 and ME use this character set. The NT and XP series are strictly Unicode.

1.4.6.5. Unicode

1.4.6.5.1. Overview

It became apparent after the ISO 8859 standard was created, that it was ill-suited for transmitting information between different languages. To resolve this problem, as well as the problems with earlier encodings, work began on the "universal" coding system. The system is called Unicode.

The Unicode Consortium, which is based in Mountain View, California (near San Francisco), published "The Unicode Standard" in 1991 (Unicode Consortium 2003). The primary premise of the Unicode system is that each character should have a single and unique code. This value, called a "code point", would be used universally - regardless of where in the world the system is used.

The original Unicode standard set the coding system to 16-bits - giving a total of 65536 possible code points. This was more than sufficient space to include all the characters for every language on the planet - and leave plenty of room for future expansion. The problems that plagued ASCII, ISO 646 and ISO 8859 would not affect Unicode.

The characters that shared the 128-255 range in ISO 8859 were given unique code points by the Unicode system. The first 256 codes are identical to ISO 8859-1 and conversion between the two is simple. The characters themselves were organized into different ranges within the 65536 code range. For instance, Greek characters are stored between 880 and 1023 (0x370 and 0x3FF); Hebrew characters are stored between 1424 and 1535 (0x590 and 0x5FF).

Unicode was developed at the same time as many of the latter ISO 8859 standards. It has, subsequently, replaced it on most modern operating systems. The Unicode Consortium works with the International Standards Organization on the Unicode standard. However, the ISO standard (ISO/IEC 10646) is considered a subset of the Unicode. While it contains the same code points as Unicode, it does not contain additional information such as how the character is displayed and other metrics

1.4.6.5.2. Beyond 16-bit

In 2001, the Unicode Consortium released version 3.1 of the Unicode encoding specification. At this point, the 16-bit code range was expanded to 21-bits which made it possible to store over 1 million different code points.

The system was subdivided into different logical "planes" that contain different broad classes of characters. The initial 65536 characters of Unicode were organized into the Basic Multilingual Plane (BMP). This set includes all characters that are part of modern written languages and common symbols such as icons.

Plane 1, the Supplementary Multilingual Plane (SMP), is used to store characters that are part of historical languages such as Linear B. Musical and rare mathematical characters are also stored here. Plane 2, the Supplementary Ideographic Plane (SIP), is used to store over 40,000 rare historical Chinese characters. Plane 14 is used to store a number of nonrecommended and experimental tag symbols. The nature of this plane is nebulous and will, no doubt, change over time. Planes 15 and 16 are open for private use.

Since Unicode is a multiple-byte encoding standard, byte ordering is of vital importance. The Unicode Consortium defined a number of Unicode Transformation Formats (UTF) to encode characters. These include UTF-7, UTF-8, UTF-16 and UTF-32. The International Standards Organization (ISO), in the ISO/IEC 10646 specification, also defined two different Universal Character Sets (UCS) to store Unicode code points. Essentially, both UCS encodings are subsets of the UTF encoding.

1.4.6.5.3. UCS

UCS-2, like the original version of Unicode, is primarily 16-bit. As expected, UCS-2 is only able to store the Basic Multilingual Plane (the first 65536 Unicode Characters). UCS-4 encoding uses a total of 32-bits to store each character code. The full Unicode encoding can currently be represented with only 21 bits, which makes UCS-4 a particularly inefficient format. However, since computers generally store integer values in powers of 2, 32-bit integers are common on practically all platforms while 24-bit variants are exceedingly rare.

1.4.6.5.4. UTF-16

UTF-16 is almost identical to the UCS-2 format with some, very important, exceptions. This format usually stores each character code using 2-bytes like UCS-2, but also provides override sequences for encoding characters that are not part of the Basic Multilingual Plane. This allows the system to represent the normal Unicode characters using 16-bit, but also can provide the representation of Plane 1 and Plane 2 characters.

UTF-16 also supports different byte ordering sequences. To accomplish this, every transmitted UTF string is preceded by a Byte Order Mark (BOM) which tells the decoder the byte ordering of the following Unicode code points. The BOM is 2-bytes - with one byte containing FF and the other containing FE. 0xFFFE alerts the decoder that the information is stored in Little Endian; 0xFEFF is for Big Endian.

Since practically all real-world Unicode characters are part of the Basic Multilingual Plane, UCS-2 is usually sufficient. However, UTF-16 has the benefit of providing a method for supporting the full Unicode encoding. As a result, UTF-16 is predominately used in most systems that support Unicode. Both Windows NT / XP and Linux use UTF-16 internally.

1.4.6.5.5. UTF-8

The UTF-8 format supports a number of override sequences such that each code in the Unicode encoding can be represented. UTF-8 was designed specifically so that a string can be represented without any issues caused by byte ordering. The encoding also will not conflict with ASCII control characters - meaning that the string can be stored in legacy programs that are strictly based on ASCII and use the null-character to terminate strings.

UTF-8 is popular for transmitting Unicode information over the Internet and, more notably, e-mail. Unfortunately, the number of e-mail clients that support Unicode varies and most information sent via e-mail is done using ISO 8859 or Windows-1252.

1.4.6.5.6. UTF-7

UTF-7 is a 7-bit variant of UTF that uses a combination of Base64 (used in MIME) and override characters. However, since HTML-style encoding can also represent any Unicode code point, UTF-7 is rarely, if never, used.

1.5. Existing Approaches

1.5.1. Compiler-Compilers

The current approach used to develop parsers and compilers, is through a class of applications known as "compiler-compilers". In the Compiler-compiler paradigm, an application is used to analyze a hybrid program containing both actual source code and special markup directives. These directives are recognized by the application and used to generate a new program which can then be compiled and executed. In other words, the hybrid program (which cannot be compiled) is analyzed by the compiler-compiler and a new program is generated.

Overall, compiler-compilers share some of the same benefits and suffer from the same weaknesses regardless of the individual software suite. Compiler-compilers have the advantage of close integration between the source code and the special directives. It is fairly easy for the developer to identify which tokens should be stored in a data structure and which can be discarded. In many ways the hybrid language becomes a superset of the original language.

However, this attribute also limits compiler-compilers to the language for which they were developed. The notation that is used for the compiler-compiler directives are designed not to conflict with the host language. For each new programming language, a new system must be developed such that conflicts do not occur. As a result, grammars are not portable between systems.

The two major parsing systems that are currently available are YACC (Johnson 1979) and ANTLR (Parr 2000). Although differences in design and functionality exist between the two, both systems follow the compiler-compiler paradigm.

1.5.2. YACC

1.5.2.1. Overview

One of the oldest and most respected parsing engine generators available to developers is YACC. Like "vi" "grep" and "awk", this software is considered the de facto standard in the UNIX world. YACC, which is an acronym for Yet Another Compiler-Compiler, was developed by Stephen C. Johnson at AT&T (Johnson 1979). YACC can be used to create parsers using the C and C++ programming languages.

The term "YACC" actually refers to two separate compiler-compilers, "lex" and "yacc", which constitute the development platform. Each application is used to generate a different part of the parsing system and, in turn, use different input files with the same generic format, but varying semantics.

The "lex" application generates a new program called "lex.yy.c" which implements the tokenizer using a Deterministic Finite Automata. The other application, "yacc", generates a new program called "y.tab.c" which implements a LALR parsing algorithm. Once each application has created their respective program, both can be combined to create the complete system.

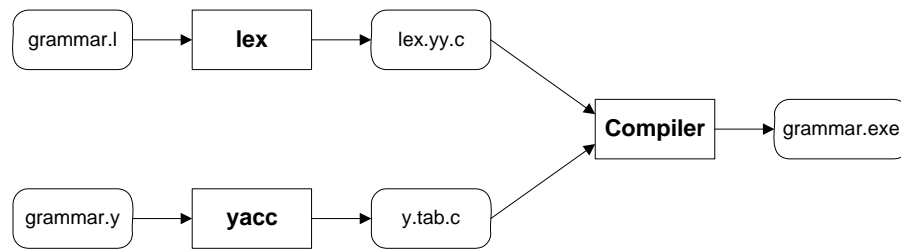


Figure 1-8. YACC/Lex Design Flow

When developing a parser using YACC / Lex, the grammar's rules and definitions are integrated directly into the C source code. The YACC and Lex applications use preprocessor-type statements to divide each program into three distinct parts. The first section allows the developer to define special datatypes, lexical rules and other information that will be used to generate source code. The second section contains rules, terminal definitions and their associated semantic actions. The last section contains the rest of the program that is not analyzed directly by the system.

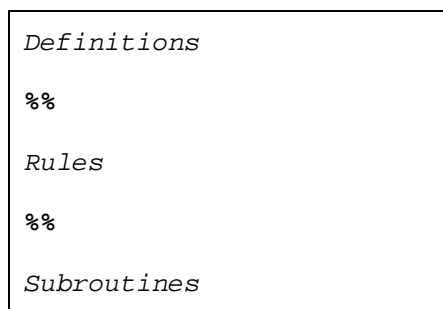


Figure 1-9. YACC / Lex Program Structure

1.5.2.2. Lex File Details

The first section, of Lex files, allows the developer to specify character sets that can be used to define each terminal. Like the C programming language, the Lex Meta-Language uses the backslash character as an override.

```
digit      [0-9]
letter     [A-Za-z]
id_tail    [0-9A-Za-z\_]
```

The second section of a Lex file contains the definitions for each terminal in the system. The first part of each definition contains a regular expression. Following each regular expression, the developer can include C source code which can analyze the text and pass an identifier to the calling procedure. These identifiers are declared by the "yacc" application, not by "lex" itself. If the C source does not return an identifier, the terminal is essentially ignored by the system.

```
{letter}{id_tail}*      { return (ID); }
{digit}+(\.{digit})*?   { return (NUMBER); }
\n                      { } /* Ignore newline */
```

1.5.2.3. YACC File Details

The first section in a YACC file allows the developer to specify the special datatypes used by the system, the start symbol of the grammar, and the identifiers which represent tokens. YACC files are denoted by a letter "y" extension.

The developer also has the ability to specify special constructs for defining mathematical expressions. Rather than defining each level of precedence, the %left and %right tags are used to designate left and right associative operators. Operators which are nonassociative, such as unary operators, use the %nonassoc tag. When the developer uses these special terminals in the grammar's rules, the system will automatically construct the hidden rules needed to handle precedence levels. Since the YACC system is a compiler-compiler, this additional logic can be hidden from the developer.

```
%token IF THEN ELSE END
%token WHILE LOOP

%left PLUS MINUS
%left DIVIDE ASTERISK

%start program
```

The second section allows the developer to specify each of the grammar's rules using a variation of Backus-Naur Form. The YACC format uses the same notation for both terminals and nonterminals. The developer specifies whether a symbol is terminal by defining it in the "Declarations" section.

Each rule contains an optional segment of C code that is used to handle the data produced when a rule is reduced. Special codes are used to define the data type used to store reductions as well as where YACC code is to link to the original C source. Each token in a reduced rule is assigned a numeric index starting at 1 and is referenced using a dollar character prefix. For instance, "\$1" will refer to the first token of the reduced rule. Reductions are assigned by the user using "\$\$". When YACC analyzes the source file, it replaces these special codes with actual C code.

By following each rule by the C code segment, it is fairly easy for the developer to identify which tokens should be stored in a data structure and which can be discarded. For instance, in the following YACC grammar:

```
stat : WHILE exp DO stats END { $$ = whileStmt($3, $5); }  
;
```

The C code calls a function called 'whileStmt' which creates a datatype used to store while statement and the associated values for the "exp" and "stats" tokens.

1.5.3. ANTLR

ANTLR, Another Tool for Language Recognition, is an object-oriented parser generator designed by Terrence Parr (who now teaches at the University of San Francisco). The system is able to generate parsers for several programming languages which have the same basic syntax as C++ (Parr 2000). These include Sun Java, Microsoft C#, and, naturally, C++ itself.

Unlike the approach used by YACC, ANTLR does not link different code segments together, but instead, creates a parser, lexer and tree storage class based on the ANTLR meta-language. This meta-language allows the developer to define the names and properties of different classes which will be generated by the ANTLR compiler-compiler. Developers define a grammar by integrating C++ style class headers into their code. Each class inherits one of the three built-in ANTLR classes: Parser, Lexer and TreeParser, and is subsequently followed by generation options, recognized tokens, and rules.

```
{ optional class preamble }  
class DeveloperClassName extends ANTLRObject ;  
options  
tokens...  
rules...
```

Figure 1-10. ANTLR Object Declaration

The "Lexer" class performs the duties of the tokenizer; in particular reading from the source text and producing tokens. The "Parser" and "TreeParser" classes contain the actual parsing algorithms. These two classes are virtually identical with the exception that the "Parser" accepts a series of tokens while the "TreeParser" uses an abstract datatype.

Each ANTLR class uses the LL(k) parsing algorithm – regardless of whether the system is identifying a token or completing part of a rule. As a result, the notation and functionality of each class is virtually identical. However, since the LL(k) is used, no rule can contain left-recursion.

Due to its close integration with the C++ family of programming languages, ANTLR is able to define an object hierarchy for storing lists and optional clauses. These include objects for lists, stacks, tokens, buffers, etc.... This allows the system to use Extended Backus-Naur Form (Fischer, 1988, Sec 2.3) for rules since the notation used for lists and clauses can be stored into the appropriate object.

The grammar format differentiates between terminals and nonterminals based on type case. If the first character of the identifier is capitalized, the system will interpret it as a terminal; otherwise, the identifier represents a nonterminal. For instance, the following defines a terminal called Identifier:

```
Identifier : ( 'a'..'z' )+ ;
```

and the following defines a series of Identifiers:

```
identifiers : ( Identifier )+ ;
```

2. Design

2.1. Primary Goal

The goal of this project is to design and implement a parsing system that can support multiple programming languages and, as a result, create a consistent development platform.

2.2. Programming Language Independence

2.2.1. Separate the Generator from the Actual Parser

If the parser is to be able to support multiple programming languages, the system must not produce any information that is limited to a single language. As result, the compiler-compiler concept cannot be used. Instead, the information created by the parser generator must be able to be saved to an independent file and then later used by the actual parsing engine.

Hence, the system will consist of two distinct components – the Builder and Engine. The Builder will be the main application and will be used to analyze a description of the grammar, create tables and

provide all other services that aid language development. The parse information created by the Builder will be subsequently saved to a file so that the actual parsing engine can load and use it.



Figure 2-1. Two Components & Shared File

The Engine, therefore, will read the parse tables and do the actual runtime work. Different versions of the Engine can be created for different programming languages and IDEs.

These two components of the development suite will also be referred to as the "GOLD Builder" and "GOLD Engine" if precision is required.

2.2.2. Use the LALR Algorithm

Many parsers use the LALR algorithm to analyze syntax and a Deterministic Finite Automata to identify different classes of tokens. Both of these algorithms are simple table-based automatas – requiring little logic at runtime to operate. It is, consequently, the computation of these tables where the complexity of the system resides. The parse tables essentially consist of a set of instructions for the automatas, and, as a result, the tables can be used between programming languages. Hence, the parsing engine can be implemented in different programming languages with a minimal of programming.

Since both LALR and DFA algorithms are simple automatas, minimal code will be necessary to create different implementations of the Engine in different programming languages.

2.3. Meta-Language

One of the most important aspects of a parser generator is the format and functionality of the meta-grammars. The notation used by each parsing system varies greatly, and grammars used by different generators are rarely compatible.

To allow the easy development of grammars, the following criteria were used in the design of the notation of the GOLD meta-grammar and its interaction with the Builder and related tools.

1. The GOLD meta-language must not contain any features which are programming language dependant. In compiler-compilers, the semantic actions are integrated directly into the meta-grammar. While this does aid the development of the application, it does not allow meta-grammars to be used for other programming languages, at least without major revisions.
2. The notation of the meta-language should be very close to the standards used in language theory. This will allow both students and professionals, familiar with language theory, to be able to write grammars without a large learning curve. As a result, the definitions for terminals will use regular expressions and definitions for rules will use Backus-Naur Form.
3. The meta-language should include all language attributes. There are many aspects of programming languages which cannot be specified using regular expressions or Backus-Naur Form. These include the actual name of the grammar, whether it is case sensitive or not, the author, etc.... This information

could be specified by the Builder application, but different versions of the Builder can be created over time. To create a consistent development process, the Builder application, regardless of how it is implemented, will follow the attributes set in the grammar rather than local settings.

2.4. Design Flow

Below is a brief synopsis of the development cycle:

1. The first step to design a compiler or interpreter is to write a grammar for the language being implemented. The description of the grammar is written using any text editor - such as Notepad or the editor that is built into the GOLD Builder. This does not require any coding.
2. Once the grammar is complete, it is analyzed by the GOLD Builder. During this process, LALR and DFA parse tables are constructed and any ambiguities or problems with the grammar are reported.
3. Once the grammar is analyzed, the tables are saved to a file to be used later by the actual parsing engine. At this point, the GOLD Builder is no longer needed - having performed its duties.
4. The parse tables are read by the parsing engine. This can be an ActiveX DLL, .NET Module, or another parsing engine implemented in another programming language.
5. The source text is analyzed by the parser engine and a parse tree is constructed.

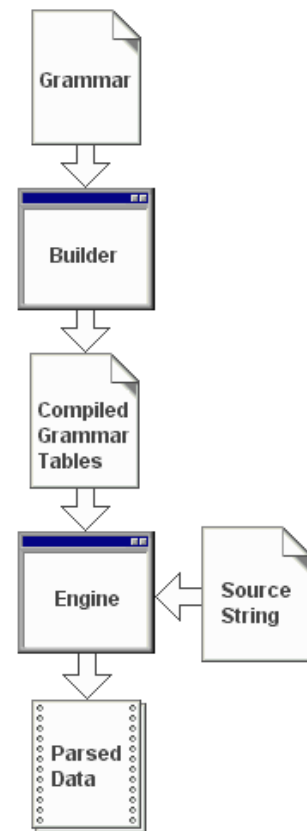


Figure 2-2. Data Flow

3. The Builder Module

3.1. Meta-Language

3.1.1. Defining Character Sets

3.1.1.1. Overview

Character sets are used to aid the construction of the regular expressions used to define terminals. Literal sets of characters are delimited using the square brackets '[' and ']'. The YACC parser generator, allows the developer to specify [a-z] to define a set including all lower case letters in the ANSI character set. However to maintain simplicity in set notation, GOLD will interpret the contents literally. The set [a-z] will, consequently, only include the characters "a", "-", and "z". The lost functionality that is caused by this design consideration will be easily recaptured through the use of pre-defined sets and user-defined sets.

User-defined sets are delimited by the braces '{' and '}'. For instance, the text "[abcde]" denotes a set of characters consisting of the first five letters of the alphabet; while the text "{abc}" refers to a set named "abc".

Sets can then be declared by adding and subtracting previously declared sets and literal sets. The GOLD Builder will provide a collection of pre-defined sets that contain characters often used to define terminals.

3.1.1.2. Syntax

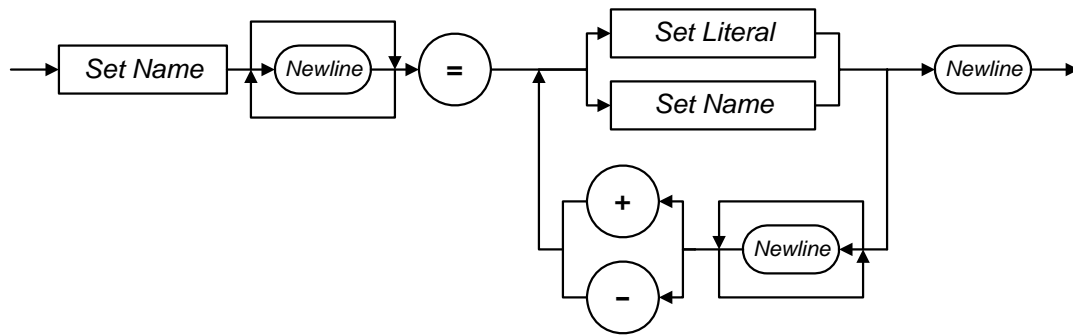


Figure 3-1. Set Syntax Diagram

3.1.1.3. Examples

Declaration	Resulting Set
{Bracket} = [' ']]
{Quote} = [' ']	'
{Vowels} = [aeiou]	aeiou
{Vowels 2} = {Vowels} + [y]	aeiouy
{Set 1} = [abc]	abc
{Set 2} = {Set 1} + [12] - [c]	ab12
{Set 3} = {Set 2} + [0123456789]	ab0123456789

The following declares a set named "Hex Char" containing the characters that are valid in a hexadecimal number.

```
{Hex Char} = {Digit} + [ABCDEF]
```

The following declares a set containing the characters that can be placed inside a normal "string". In this case, the double quote is the delimiting character.

```
{String Char} = {Printable} - ["]
```

3.1.2. Pre-Defined Character Sets

There are many sets of characters which are simply not accessible via the keyboard or so commonly used that it would be repetitive and time-consuming to redefine in each grammar. To resolve this issue, the GOLD Meta-Language contains a collection of useful pre-defined sets at the developer's disposal. These include the sets that are often used for defining terminals as well as characters not accessible via the keyboard.

3.1.2.1. Individual Characters

Some individual characters cannot be specified on a standard keyboard nor can be displayed on most systems. These include control characters such as space, carriage return and tab as well as Unicode characters which cannot be displayed on the current system.

Table 3-1. Individual Characters

Set Name	Characters
{HT}	Horizontal Tab character (#09).
{LF}	Line Feed character (#10).
{VT}	Vertical Tab character (#11). This character is rarely used.
{FF}	Form Feed character (#12). This character is also known as "New Page".
{CR}	Carriage Return character (#13).
{Space}	Space character (#32). Technically, this set is not needed since a "space" can be expressed by using single quotes: ' '. The set was added to allow the developer to more explicitly indicate the character and add readability.
{NBSP}	No-Break Space character (#160). The No-Break Space character is used to represent a space where a line break is not allowed. It is often used in source code for indentation.
{#n}	Using this notation, you can specify any character - in particular those not accessible via the keyboard. The value of, <i>n</i> can be any number between 1 and 65536. For instance, {#169} specifies the copyright character ©.
{&n}	This is the hexadecimal notation.

3.1.2.2. Commonly Used Character Sets

The GOLD Builder contains a series of pre-defined sets for characters commonly used in programming languages.

Table 3-2. Commonly Used Character Sets

Set Name	Characters
{Digit}	0123456789
{Letter}	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
{Alphanumeric}	This set includes all the characters in {Letter} and {Digit}
{Printable}	This set includes all standard characters that can be printed onscreen. This includes the characters from #32 to #127 and #160 (No-Break Space). The No-Break Space character was included since it is often used in source code.
{Whitespace}	This set includes all characters that are normally considered whitespace and ignored by the parser. The set consists of the Space, Horizontal Tab, Line Feed, Vertical Tab, Form Feed, Carriage Return and No-Break Space.
{Letter Extended}	This set includes all the letters which are part of the extended Unicode character set.
{Printable Extended}	This set includes all the printable characters above #127. Although rarely used in programming languages, they could be used, for instance, as valid characters in a string literal.
{ANSI Mapped}	This set contains the characters between 128 and 159 that have different values in Unicode.
{ANSI Printable}	This set contains all printable characters available in ANSI. Essentially, this is a union of {Printable}, {Printable Extended} and {ANSI Mapped}.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI	6
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	16
20	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	32
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	48
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	64
50	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	80
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	96
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL	112

80	XXX	XXX	BPH	NBH	XXX	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3	128
90	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	XXX	SCI	CSI	ST	OSC	PM	APC	144
A0	NB SP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯	160
B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	176
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	192
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	208
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	224
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	240

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
--	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

Control Characters

Digit

Letter

Whitespace

Printable

Letter Extended

Printable Extended

Figure 3-2. Commonly Used Character Set Map

3.1.2.3. Unicode Character Sets

Not all Unicode characters can be expressed on a standard keyboard. In fact, with characters numbering into the tens of thousands, no single keyboard can have easy access to all of them. Fortunately, the Unicode character set groups different classes of characters together.

To give the developer easy access to each group, the Builder contains pre-defined sets for each. The names of each of the following sets are based on the names given to them by the Unicode Consortium (Unicode Consortium 2003).

Table 3-3. Unicode Character Sets

Set Name	Character Range In Decimal		Character Range In Hexadecimal	
	From	To	From	To
{Printable Extended}	#161	#255	&A1	&FF
{Latin Extended}	#256	#687	&100	&2AF
{Greek}	#880	#1023	&370	&3FF
{Cyrillic}	#1024	#1279	&400	&4FF
{Cyrillic Supplementary}	#1280	#1327	&500	&52F
{Armenian}	#1328	#1423	&530	&58F
{Hebrew}	#1424	#1535	&590	&5FF
{Arabic}	#1536	#1791	&600	&6FF
{Syriac}	#1792	#1871	&700	&74F
{Thaana}	#1920	#1983	&780	&7BF
{Devanagari}	#2304	#2431	&900	&97F
{Bengali}	#2432	#2559	&980	&9FF
{Gurmukhi}	#2560	#2687	&A00	&A7F
{Gujarati}	#2688	#2815	&A80	&AFF
{Oriya}	#2816	#2943	&B00	&B7F
{Tamil}	#2944	#3071	&B80	&BFF
{Telugu}	#3072	#3199	&C00	&C7F
{Kannada}	#3200	#3327	&C80	&CFF
{Malayalam}	#3328	#3455	&D00	&D7F
{Sinhala}	#3456	#3583	&D80	&DFF
{Thai}	#3584	#3711	&E00	&E7F
{Lao}	#3712	#3839	&E80	&EFF
{Tibetan}	#3840	#4095	&F00	&FFF
{Myanmar}	#4096	#4255	&1000	&109F
{Georgian}	#4256	#4351	&10A0	&10FF
{Hangul Jamo}	#4352	#4607	&1100	&11FF
{Ethiopic}	#4608	#4991	&1200	&137F
{Cherokee}	#5024	#5119	&13A0	&13FF
{Ogham}	#5760	#5791	&1680	&169F
{Runic}	#5792	#5887	&16A0	&16FF
{Tagalog}	#5888	#5919	&1700	&171F
{Hanunoo}	#5920	#5951	&1720	&173F
{Buhid}	#5952	#5983	&1740	&175F
{Tagbanwa}	#5984	#6015	&1760	&177F
{Khmer}	#6016	#6143	&1780	&17FF
{Mongolian}	#6144	#6319	&1800	&18AF
{Latin Extended Additional}	#7680	#7935	&1E00	&1EFF
{Greek Extended}	#7936	#8191	&1F00	&1FFF
{Hiragana}	#12352	#12447	&3040	&309F

{Katakana}	#12448	#12543	&30A0	&30FF
{Bopomofo}	#12544	#12591	&3100	&312F
{Kanbun}	#12688	#12703	&3190	&319F
{Bopomofo Extended}	#12704	#12735	&31A0	&31BF

3.1.3. Comments

3.1.3.1. Overview

It is standard practice for developers to add comments to their code to add readability, aid in future maintenance of the grammar, and include other information that may be informative or useful. The GOLD Meta-Language will allow the developer to specify both line comments and block comments.

The "!" symbol can be used to denote that the rest of the current line is a comment. This symbol was selected because of its interpretation as "attention" or "alert" in the Latin languages. Essentially a comment serves this very purpose.

The symbols "!*" and "*!" were selected to specify the start and end of a block comment. The asterisk was added for its traditional use in computer science for representing zero or more characters in pattern matching strings. Such examples include the UNIX shell and MS-DOS.

3.1.3.2. Example

```
! This is a comment
! This is also a comment

!*
    Remember to always comment your code. This
    can add a great deal to readability.
*!
```

3.1.4. Defining Terminals

3.1.4.1. Overview

Terminals are used to define the reserved words, symbols and recognized patterns, such as identifiers, in a grammar. Each terminal is defined using a regular expression which is used to construct the Deterministic Finite Automata used by the tokenizer.

3.1.4.2. Syntax

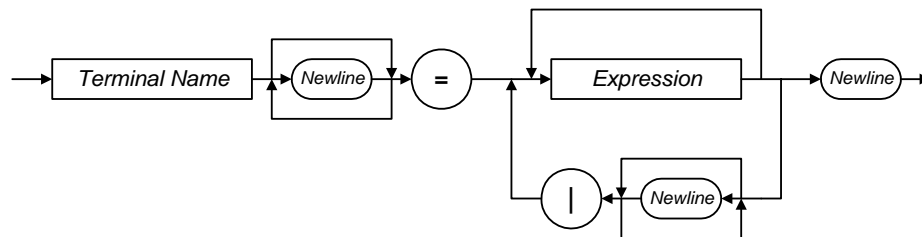


Figure 3-3. Terminal Syntax Diagram

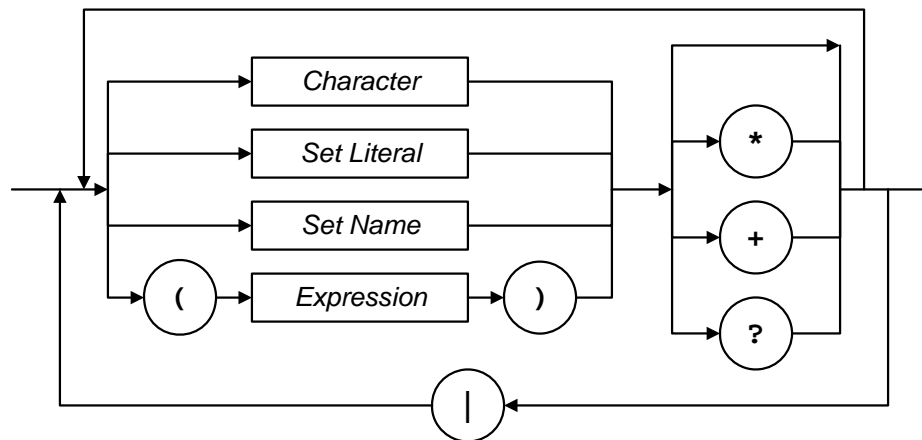


Figure 3-4. Regular Expression Syntax Diagram

3.1.4.3. Implicit Declaration

The content of some terminals, such as identifiers, is highly variable – following a pattern defined in its respective regular expression. However, there is a class of terminals whose format does not change. Instead, their purpose in the grammar is to add readability and often restrict the scope of different language constructs.

For instance, the vast majority of programming languages contain the reserved word 'if' which is typically used to define the start of a conditional statement or block. The format of the 'if' terminal is constant, and as a result, the programmer is able to identify the construct with ease.

When a developer defines a grammar for a programming language, these reserved words and symbols are used directly in the grammar description with the assumption that they are, in fact, constant. They expect that when the system looks at the 'if', it will know that tokenizer will only accept the text exactly as defined. For instance, in the following Backus-Naur Form statement, the terminals 'if', 'then', 'else' and 'end' are used.

```
if <exp> then <Stmts> else <Stmts> end
```

For languages containing a large number of reserved words and symbols, manually defining each would be tedious and time consuming. Rather than requiring the developer to manually define each terminal in the meta-language, the system will implicitly define these terminals. For the most part, the definition of a terminal will be exactly as it is typed in the meta-language. The 'if' terminal, for instance, is simply defined as follows:

```
if = 'if'
```

3.1.4.4. Examples

Declaration	Valid strings
Example1 = <code>abc*</code>	<code>ab, abc, abcc, abccc, abcccc, ...</code>
Example2 = <code>ab?c</code>	<code>abc, ac</code>
Example3 = <code>a b c</code>	<code>a, b, c</code>
Example4 = <code>a[12]*b</code>	<code>ab, alb, a2b, al2b, a21b, a22b, all1b, ...</code>
Example5 = <code>'*'+</code>	<code>*, **, ***, ****, ...</code>
Example6 = <code>{Letter}+</code>	<code>MrPoo, cat, perfect, ...</code>
Identifier = <code>{Letter}{AlphaNumeric}*</code>	<code>e4, Param4b, Color2, temp, ...</code>
ListFunction = <code>c[ad]+r</code>	<code>car, cdr, caar, cadr, cdar, cddr, caaar, ...</code>
ListFunction = <code>c(a d)+r</code>	The same as the above using a different, yet equivalent, regular expression.
NewLine = <code>{CR}{LF} {CR}</code>	Windows and DOS use <code>{CR}{LF}</code> for new lines, UNIX simply uses <code>{CR}</code> . This definition will detect both.

3.1.5. Whitespace Terminal

3.1.5.1. Introduction

In practically all programming languages, the parser recognizes (and usually ignores) the spaces, new lines, and other meaningless characters that exist between tokens. For instance, in the following code:

If	Done	Then
	Counter	= 1;
End	If	

the fact that there are three spaces between the 'If' and 'Done', a new line after 'Then', and multiple spaces before 'Counter' is irrelevant. From the parser's point of view (in particular the Deterministic Finite Automata that it uses) these whitespace characters are recognized as a special terminal which can be discarded. In GOLD, this terminal is simply called the "Whitespace terminal" and can be defined to whatever is needed.

If the Whitespace Terminal is not defined explicitly in the grammar, it will be implicitly declared as one or more of the characters in the pre-defined Whitespace set: {Whitespace}+.

Normally, you would not need to worry about the Whitespace terminal unless you are designing a language where the end of a line is significant. This is the case with Visual Basic, BASIC and many, many others.

3.1.5.2. Example

In programming languages such as Visual Basic, the grammar does not ignore the end of a line, but, instead, uses it as an essential part of the language.

To accomplish this, the grammar must be able to recognize the new line as a terminal rather than simply considering it whitespace. The characters used to represent a new line differ slightly between computer platforms. The Windows operating system uses the combination of a Carriage Return followed by a Line Feed; UNIX, on the other hand, merely uses the Carriage Return. The definition of a Newline terminal must take this into account.

In the declaration below, a NewLine terminal is declared with the two possible permutations of the Carriage Return and Line Feed. The Whitespace Terminal must also be declared such that it does not accept the NewLine as whitespace. Below, Whitespace is declared as a series of the normal whitespace characters without the Carriage Return and Line Feed.

$\{WS\} = \{Whitespace\} - \{CR\} - \{LF\}$ $Whitespace = \{WS\}^+$ $NewLine = \{CR\}\{LF\} \{CR\}$

3.1.6. Comment Terminals

3.1.6.1. Introduction

Essentially, there are two different types of comment terminals used in programming languages: those that tell the compiler to ignore the remaining text in the current line of code and those used to denote the start and end of a multi-line comment. To accommodate the intricacies of comments, the GOLD Builder provides for this special class of terminals.

3.1.6.2. Comment Start

The Comment Start terminal defines the symbol used to begin a block comment. When the tokenizer engine reads this symbol from the source text, it will increment an internal counter and ignore all other tokens until the Comment End token is encountered. Comments will be nested.

3.1.6.3. Comment End

The Comment End terminal defines the symbol that will denote the end of a block comment.

3.1.6.4. Comment Line

Unlike the Comment Start and Comment End terminals, the tokenizer will simply discard the rest of the line.

3.1.6.5. Examples of Comment Terminals

Below is a comparison of comment terminals in several common programming languages. Blanks fields denote the programming language lacks a terminal of that type. For instance, Visual Basic does not provide block comments.

Table 3-4. Examples of Comment Terminals

Programming Language	Line Comment	Block Comment Start	Block Comment End
C (Original)		<code>/*</code>	<code>*/</code>
C++	<code>//</code>	<code>/*</code>	<code>*/</code>
COBOL	<code>*</code>		
LISP (Modern)	<code>;</code>		
FORTRAN 90	<code>!</code>		
Java	<code>//</code>	<code>/*</code>	<code>*/</code>
Pascal		<code>{ or (*</code>	<code>} or *)</code>
Prolog	<code>%</code>	<code>/*</code>	<code>*/</code>
SQL	<code>--</code>	<code>/*</code>	<code>*/</code>
Visual Basic	<code>'</code>		
GOLD Meta-Language	<code>!</code>	<code>!*</code>	<code>*!</code>

3.1.7. Defining Rules

3.1.7.1. Overview

Typically, rules in a grammar are declared using Backus-Naur Form statements. This notation consists of a series of 0 or more symbols where nonterminals are delimited by the angle brackets '`<`' and '`>`' and terminals are delimited by single quotes or not delimited at all.

For instance, the following declares the common if-statement.

```
<Statement> ::= if <Expression> then <Statements> end if
```

The symbols 'if', 'then', 'end', and 'if' are terminals and `<Expression>` and `<Statements>` are nonterminals.

If you are declaring a series of rules that derive the same nonterminal (i.e. different versions of a rule), you can use a single pipe character '|' in the place of the rule's name and the "::<=" symbol.

3.1.7.2. Syntax

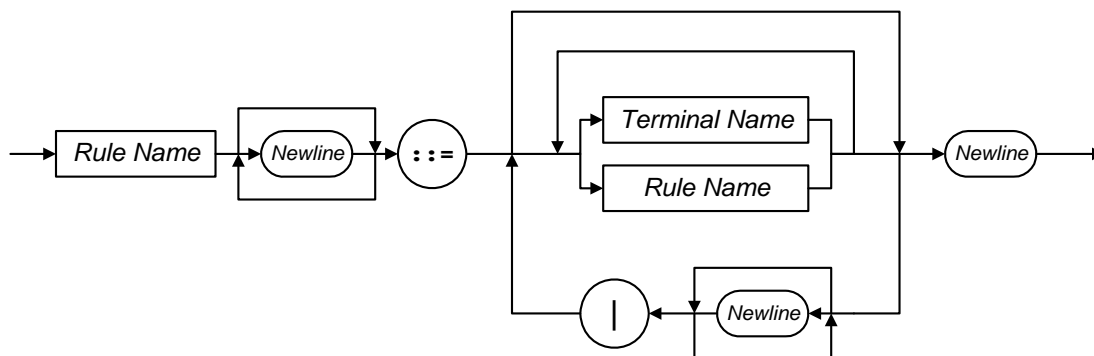


Figure 3-5. Rule Syntax Diagram

3.1.7.3. Examples

The following declares a series of 3 different rules that define a 'Statement'. In this example, the shortcut notation is used to simplify the declaration.

```
<Statement> ::= if <Expression> then <Statements> end if
               | while <Expression> do <Statements> end while
               | for Id '=' <Range> loop <Statements> end for
```

This is equivalent to:

```
<Statement> ::= if <Expression> then <Statements> end if
<Statement> ::= while <Expression> do <Statements> end while
<Statement> ::= for Id '=' <Range> loop <Statements> end for
```

Operator precedence is an important aspect of most programming languages. The following rules define the common arithmetic operators.

```
<Expression> ::= Identifier '+' <Expression>
                | Identifier '-' <Expression>
                | <Mult Exp>

<Mult Exp>    ::= Identifier '*' <Mult Exp>
                | Identifier '/' <Mult Exp>
                | Identifier
```

3.1.8. Defining Parameters

3.1.8.1. Overview

Many attributes of a grammar cannot be specified using Backus-Naur Form statements or regular expressions. These attributes can range from the formal name of the grammar to how parse tables will be constructed by the system. As a result, the Builder must allow the developer a means to declare this information within the meta-language.

The role of parameters is nebulous by design. Some may set the formal name of the grammar, for instance, and others can have significant impact on how the system constructs parse tables.

Parameter names are delimited by doubled-quotes and can be set to any of the symbols and literals recognized by the GOLD Meta-Language. In most cases, the value will be a string.

3.1.8.2. Syntax

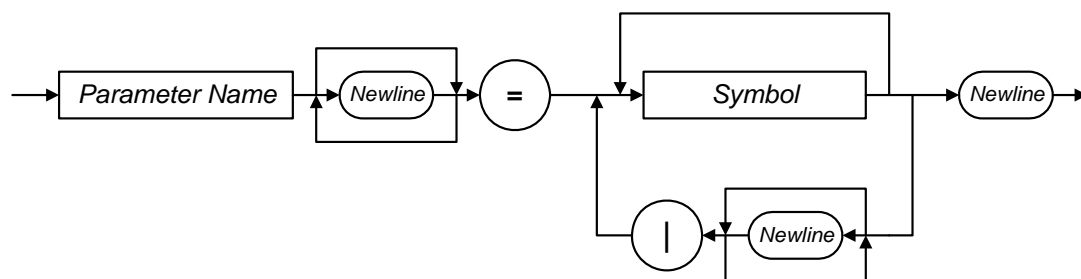


Figure 3-6. Parameter Syntax Diagram

3.1.8.3. "Name", "Version", "Author" and "About" Parameters

The purpose of the Name, Version, Author, and About parameters is to allow the developer to specify information about the grammar. The content of each of these parameters has no effect on how the system analyzes the grammar, but is important for information purposes. In all cases, the content of the parameters will be saved to the Compiled Grammar Table File, so that when it is subsequently loaded by the Engine, the grammar can be uniquely identified.

Each of these parameters is optional. If the developer does not specifically define these parameters in the grammar, they will default to predefined values. The Name parameter will default to '(Untitled)', Version will default to '(Not Specified)', Author will default to '(Unknown)', and About will default to a null string. The default values were selected arbitrarily.

For example, the following sets the parameters for the LISP programming language.

```
"Name"      = 'LISP'  
"Author"    = 'John McCarthy'  
"Version"   = 'Standard'  
"About"     = 'LISP is an abstract language that organizes ALL'  
            | 'data around "lists".'
```

3.1.8.4. "Case Sensitive" Parameter

One important aspect of a grammar, which cannot be specified using Backus-Naur Form, is whether it is case sensitive. In case sensitive grammars, such as C and Java, reserved words are uniquely identified by case as well as their respective alphabetic characters. If the grammar contains the terminal 'if', the text 'IF', 'If', and 'iF' will be invalid. Other programming languages, such as Visual Basic, are case insensitive. All the 'if' tokens would be equivalent.

The Engine, which uses a Deterministic Finite Automata for the tokenizer, must be able to handle both case sensitive and case insensitive grammars. The number of characters in Unicode is large, and, as a result, the Engine cannot be expected to contain the full mapping table. Hence, case mapping will be handled by the Builder.

If the Case Sensitive parameter is set to 'False', the Builder will populate each set in the Deterministic Finite Automata with upper and lower case versions of each character. For example, if a set contains the characters [Abc], the system will add [aBC] to create the set [AaBbCc]. The Deterministic Finite Automata only needs to perform a binary check for each character's ordinal value. The case mapping table is listed in Appendix G.

Essentially this parameter will affect the syntactic case sensitivity of the grammar rather than semantic case sensitivity. In the C programming language, identifiers are also case sensitive, but this attribute is defined by the language semantics rather than the syntax. In fact, it is possible to have a grammar where identifiers are case sensitive, but the reserved words are not. This parameter defaults to 'False'.

3.1.8.5. "Character Mapping" Parameter

For the most part, all the characters between 0 and 255 are identical in the Unicode and ANSI character sets. However, there are a number of characters which were needed for ANSI systems, but were not part of the first 256 Unicode characters. These include the Euro, the bullet (dot), trademark symbol, as well as a few other symbols.

The solution for ANSI systems was to find locations within the 256 characters which could be used to accommodate these symbols. The characters between 128 and 159 are defined as Unicode control characters, and it is here that the symbols were stored.

As a result, the parsing system must be able to handle "pure" Unicode systems as well as systems that use the hybrid format. The purpose of the Character Mapping parameter is to allow the developer to handle this issue. If the parameter is not directly specified or explicitly set to "ANSI", the system will create double entries for both the Unicode index and the ANSI index.

The following diagram enumerates each of the characters affected by this parameter.

Table 3-5. Character Mapping Table

Character	ANSI		Unicode	
	Decimal	Hexadecimal	Decimal	Hexadecimal
€	128	0x80	8364	0x20AC
,	130	0x82	8218	0x201A
<i>f</i>	131	0x83	402	0x0192
„	132	0x84	8222	0x201E
...	133	0x85	8230	0x2026
†	134	0x86	8224	0x2020
‡	135	0x87	8225	0x2021
^	136	0x88	710	0x02C6
‰	137	0x89	8240	0x2030
š	138	0x8A	352	0x0160
<	139	0x8B	8249	0x2039
œ	140	0x8C	338	0x0152
ž	142	0x8E	381	0x017D
`	145	0x91	8216	0x2018
'	146	0x92	8217	0x2019
“	147	0x93	8220	0x201C
”	148	0x94	8221	0x201D
•	149	0x95	8226	0x2022
–	150	0x96	8211	0x2013
—	151	0x97	8212	0x2014
~	152	0x98	732	0x02DC
™	153	0x99	8482	0x2122
š	154	0x9A	353	0x0161
>	155	0x9B	8250	0x203A
œ	156	0x9C	339	0x0153
ž	158	0x9E	382	0x017E
ÿ	159	0x9F	376	0x0178

3.1.8.6. "Auto Whitespace" Parameter

This parameter is set to 'True' by default, but can be changed to 'False'. When 'False', the system will not create a whitespace terminal unless it is manually defined.

3.1.8.7. "Start Symbol" Parameter

The starting symbol in the grammar. When LALR parse tables are constructed by the GOLD Builder, an "accepted" source string will reduce to this nonterminal.

3.1.8.8. Example

```
"Name"      = 'My Programming Language'
"Version"   = '1.0 beta'
"Author"    = 'John Q. Public'

"About"     = 'This is a test declaration. You can use'
              | 'multiple lines by using the "pipe" symbol'

"Case Sensitive" = False
"Start Symbol" = <Statement>
```


3.1.9. Examples

3.1.9.1. Identifiers

The format and function of identifiers vary from programming language to programming language. However, in most cases, the identifier follows the de-facto format used from BASIC to C. The following grammar defines a very simple format for identifiers using predefined character sets. Essentially, the Identifier terminal is defined as a letter followed by zero or more alphanumeric characters.

```
Identifier  = {Letter}{Alphanumeric}*  
<Value>    ::= Identifier
```

Of course, identifiers in most languages allow additional characters such as the hyphen in COBOL and the underscore in Visual Basic and the C family of languages. The grammar segment below defines the format used by C.

```
{ID Head} = {Letter} + [_]  
{ID Tail} = {AlphaNumeric} + [_]  
  
Identifier = {ID Head} {ID Tail}*  
<Value>    ::= Identifier
```

3.1.9.2. Numbers

Besides identifiers, programming languages often contain terminals for integer and floating point literals. The following grammar segment defines both.

```

Integer  = {Digit}+
Read     = {Digit}+ '.' {Digit}+

<Value> ::= Integer
          | Real

```

The following grammar demonstrates how to define Hex Literals using the formats for both C++ and Visual Basic. The {Hex Char} set explicitly contains both uppercase and lowercase letters; however, this is not required if the grammar is defined as case-insensitive.

```

{Hex Char}      = {Digit} + [abcdef] + [ABCDEF]

CHexLiteral     = '0x' {Hex Char}+
VBHexLiteral    = '&H' {Hex Char}+ [&]?

<Hex> ::= CHexLiteral
        | VBHexLiteral

```

3.1.9.3. String Terminal

The following example declares the string terminal commonly used in programming languages. It is declared as a series of zero or more printable characters (not including the double-quotes used for delimiters).

```

{String Char} = {Printable} - ["]
String = '"' {String Char}* '"'

```

However, this definition does not allow the programmer to specify the double-quote character. Two distinct approaches are used in modern programming languages. The first approach is to use another structure or constant to represent the double-quote. The second approach is the use of an "override"

character that allows the double-quote to be placed directly in the string. The latter approach is used in the C/C++ programming language family.

The following contains a much more complex definition for strings, but implements the second approach mentioned above. The backslash character '\' now acts as an override and can be used to represent the double-quote character. Essentially, the string terminal is now a series of any printable character (not including the double-quote and backslash) and any printable character preceded by the backslash.

```
{String Char} = {Printable} - ["\]
String = '"' ({String Char} | '\'{Printable})* '"'
```

3.1.9.4. Lists

3.1.9.4.1. Comma Delimited

The following declaration defines a list of identifiers where commas are used to separate list items. The last item in the list will not be followed by a comma. This definition does not allow the list to be completely empty.

```
Identifier = {Letter}{Alphanumeric}*
<List> ::= <List Item> ',' <List>
          | <List Item>
<List Item> ::= Identifier
```

3.1.9.4.2. Statement List

The following declaration defines a list of "Statements" where each <Statement> will be followed by a semicolon. Unlike the example above, the last item in the list will be followed by the symbol. However, since the <List> rule is nullable (the blank line), the list can be completely empty.

```
Identifier = {Letter}{Alphanumeric}*

<List> ::= <Statement> ';' <List>
        |
<Statement> ::= print '(' Identifier ')'
               | read '(' Identifier ')'
```

This is a declaration identical to the example above, except the list must contain at least one item.

```
Identifier = {Letter}{Alphanumeric}*

<List> ::= <Statement> ';' <List>
        | <Statement> ';'

<Statement> ::= print '(' Identifier ')'
               | read '(' Identifier ')'
```

3.1.9.5. Expressions

Practically all programming languages allow the developer to write expressions. This grammar contains the basic rules needed to define mathematical expressions

```

"Start Symbol"    = <Expression>

Identifier  = {Letter}{AlphaNumeric}*
Integer    = {Digit}+

<Expression>    ::= <Expression> '+' <Mult Exp>
                  | <Expression> '-' <Mult Exp>
                  | <Mult Exp>

<Mult Exp>      ::= <Mult Exp> '*' <Negate Exp>
                  | <Mult Exp> '/' <Negate Exp>
                  | <Negate Exp>

<Negate Exp>    ::= '-' <Value>
                  | <Value>

<Value>         ::= Identifier
                  | Integer
                  | '(' <Expression> ')'

```

3.1.9.6. Hanging-Else Problem

The following is a very simple grammar with a complex problem:

```

Id = {Letter}{AlphaNumeric}*

<Statement> ::= if Id then <Statement>
               | if Id then <Statement> else <Statement>
               | Id ':' Id

```

When this grammar is analyzed by the GOLD Builder (or any other LALR parser generator for that matter), a problem arises in the LALR (1) parse tables. Invariably, a shift-reduce error occurs when the parser reaches the "else" option on the If-Then statement.

This type of error is caused by ambiguity in the grammar itself; in the case of the above grammar, the parser does not know where it can reduce a rule or must push a token onto the parse stack. Technical issues aside, it is important to understand why this grammar is ambiguous.

The ambiguity of the grammar can be seen with a very simple piece of source code:

```
if Enrolled then if Studied then Grade:=A else Grade:=B
```

The sample source code could be interpreted two distinct ways by the grammar. The first interpretation would bind the "else" to the first "if".

```
if Enrolled then if Studied then Grade:=A else Grade:=B
```

The second interpretation would bind the "else" to the second "if" statement:

```
if Enrolled then if Studied then Grade:=A else Grade:=B
```

Fortunately, there are two approaches you can take to resolve the problem.

3.1.9.6.1. Solution #1: Modify the Grammar

This approach modifies the grammar such that the scope of the "if" statement is explicitly stated. Another terminal is added to the end of each "if" statement - in this case, an "end". A number of programming languages use this approach; the most notable are: Algol, Visual Basic and Ada.

```
Id = {Letter}{AlphaNumeric}*  
  
<Statement> ::= if Id then <Statement> end  
                | if Id then <Statement> else <Statement> end  
                | Id ':=' Id
```

As seen below, the ambiguity of the original grammar has been resolved.

```

if Enrolled then if Studied then Grade:=A end else Grade:=B end
if Enrolled then if Studied then Grade:=A else Grade:=B end end

```

3.1.9.6.2. Solution #2: Restrict the "Else"

This solution resolves the hanging-else problem by restricting the "if-then" statement to remove ambiguity. Two levels of statements are declared with the second, "restricted", group only used in the "then" clause of an "if-then-else" statement. The "restricted" group is completely identical to the first with one exception: only the "if-then-else" variant of the if-statement is allowed.

In other words, no "if" statements without "else" clauses can appear inside the "then" part of an "if-then-else" statement. Using this solution, the "else" will bind to the last "If" statement, and still allows chaining. This is the case with the C/C++ programming language family.

```

Id = {Letter}{AlphaNumeric}*

<Statement> ::= if Id then <Statement>
              | if Id then <Then Stm> else <Statement>
              | Id ':=' Id

<Then Stm>   ::= if Id then <Then Stm> else <Then Stm>
              | Id ':=' Id

```

3.1.9.7. Line-Based Grammar

The following grammars implement line-based programming languages. This type of grammar does not ignore the end of a line, but, instead, uses it as an essential part of the language. Real world examples include Visual Basic and many scripting languages.

To accomplish this, the grammar must be able to recognize the NewLine as a terminal rather than simply considering it whitespace. The characters used to represent a NewLine differ slightly between computer platforms. The Windows operating system uses the combination of a Carriage Return followed by a Line Feed; UNIX, on the other hand, merely uses the Carriage Return. The definition of a NewLine terminal must take this into account.

In the grammars below, a NewLine terminal is declared with the two possible permutations of the Carriage Return and Line Feed. It may also be advisable to make a solitary Line Feed recognized as a NewLine terminal (for fault tolerance).

The Whitespace Terminal must also be declared such that it does not accept the NewLine as whitespace. Below, Whitespace is declared as a series of the normal whitespace characters without the Carriage Return and Line Feed.

3.1.9.7.1. Solution #1 - NewLine Terminal

In this example, a NewLine is added to the end of each statement. So the grammar can allow blank lines, the statement rule also contains a single NewLine.


```

"Start Symbol" = <Program>

{WS} = {Whitespace} - {CR} - {LF}

Identifier = {Letter}{Alphanumeric}*

Whitespace = {WS}+
NewLine = {CR}{LF}|{CR}

<Program> ::= <Statements>

<Statement> ::= If <Exp> Then NewLine <Statements> End NewLine
                | Print '(' Identifier ')' NewLine
                | Read '(' Identifier ')' NewLine
                | NewLine                                !Allow blank lines

<Statements> ::= <Statement> <Statements>
                |

<Exp> ::= Identifier '=' Identifier
        | Identifier '<>' Identifier

```

3.1.9.7.2. Solution #2 - Using a NewLine rule and terminal

Although this solution above works for simple line-based grammars, it will not work well for more complex variants.

For grammars where the constructs can be quite complex, such as case-statements, this solution becomes difficult to write. For instance, assume you have the following Visual Basic Select-Case statement

1	Select Case Value
2	
3	Case 1, -1
4	Name = "True"
5	
6	Case 0
7	Name = "False"
8	Case Else
9	Name = "Error"
10	End Select

Figure 3-7. Line Based Grammar Example

Line #2 is a blank line and, as a result, must be specified in the grammar. The developer could manually declare each section where optional new lines are permitted, but this approach is very tedious and mistakes are easy to make. A better solution is to use a rule that accepts new lines rather than using the NewLine terminal at the end of each statement.

The following solution replaces each NewLine with a new rule called `<nl>` - for new lines. The `<nl>` rule is designed to accept one or more NewLine tokens. This solution makes it far easier to write complex line-based grammars. Each line is now logically followed by one or more new lines rather than just a one. The rule that accepted a blank line as a statement is no longer needed.

However, since NewLine tokens are only acceptable following a statement, any blank lines before the start of the program must be removed. In the grammar below, the `<nl opt>` rule removes any new lines before the start of the first actual line.

```

"Start Symbol" = <Program>

{WS} = {Whitespace} - {CR} - {LF}

Identifier = {Letter}{Alphanumeric}*

Whitespace = {WS}+
NewLine = {CR}{LF} | {CR}

<nl>      ::= NewLine <nl>          !One or more
           | NewLine

<nl Opt> ::= NewLine <nl Opt>      !Zero or more
           |

<Program> ::= <nl Opt> <Statements>

<Statement> ::= If Identifier Then <nl> <Statements> End <nl>
           | Print '(' Identifier ')' <nl>
           | Read '(' Identifier ')' <nl>

<Statements> ::= <Statement> <Statements>
           |

<Exp> ::= Identifier '=' Identifier
        | Identifier '<' Identifier

```

3.1.9.8. Comment Terminals

3.1.9.8.1. Comments in C++

The C++ programming language has had a strong influence on the format of comments used by different programming languages.

```

Comment Line   = '//'
Comment Start  = '/*'
Comment End    = '*/'

```

3.1.9.8.2. Comments in Pascal

In some variations of the Pascal programming language, the use of the { } and (* *) symbols must match. However this is not consistent in different implementations. The GOLD Builder will not require them to match in the declaration below.

```
Comment Start = '{' | '(*'
Comment End   = '}' | '*)'
```

3.1.9.8.3. Other Comments

The following example demonstrates the possibilities of using a complex regular expression to define comments. Though certainly not advised, it is possible to use the following declarations:

```
Comment Line = Comment
Comment Start = '{Alphanumeric}+'...'
Comment End   = '...' {Alphanumeric}+
```

By using this declaration, the line comment is the reserved word "Comment", an alphanumeric string followed by "..." starts a block comment and another alphanumeric string preceded by "..." ends it. As a result, the following would be valid tokens to start a block comment:

```
Reason...
Commented...
19...
```

And the following would be valid tokens to end a block comment:

```
...Analysis
...MyCat
...74
```

3.1.9.9. Generic Grammar Skeleton

The following contains a basic grammar that defines the terminals and character sets found in most programming languages. This declares the skeleton for a grammar that:

- Is not line based.
- Is not case sensitive.
- Has string literals consisting of all printable characters delimited by double-quotes. There are no override characters.
- Has identifiers that start with a letter and are followed by zero or more alphanumeric characters.
- Has floating point numbers which require, at least, one digit before the decimal point.

```
"Name"      = 'Add the name of the grammar'
"Version"   = 'Add a string describing the version'
"Author"    = 'Add your name'
"About"     = 'Add information about your grammar'

{String Char} = {Printable} - ["]

Identifier   = {Letter}{Alphanumeric}*
StringLiteral = '{String Char}*{'
IntegerLiteral = {Digit}+
FloatLiteral  = {Digit}+'.'{Digit}+

"Case Sensitive" = 'False'
"Start Symbol"   = <Program>

! Add the grammar's rules below:

<Program> ::= Add the rules here
```

3.2. Program Templates

3.2.1. Overview

One of the key obstacles for those using a specific implementation of the Engine is interacting with a table of rules and symbols. Each rule and symbol is uniquely identified by a table index. If a rule, for instance, has an index of 10 in the parse tables, the developer must use this value in their programs.

Manually typing each constant definition can be both tedious and problematic - given that a single incorrect constant could be difficult to debug. For most programming languages and scripting languages, the number of rules can easily exceed a hundred.

Program Templates are designed to resolve this issue. Essentially, program templates are a type of tool designed to help the programmer create a "skeleton program" which contains the source code that is necessary to use a particular implementation of the Engine. For instance, if an Engine is created for the Java Programming Language, a Program Template can be used to create a basic skeleton program to use it. This skeleton program would contain the necessary declarations and function calls to the Engine. In other words, Program Templates help a programmer use an Engine.

When a developer creates a new implementation of the Engine, they can create a template containing the bare-minimum code needed to interact with it. Of course, programmers who use a particular implementation of the Engine can create their own templates for whatever reason they need.

Some implementations of the Engine can work with multiple programming languages. These include, for instance, a version created and compiled to a Microsoft ActiveX (COM) object. This can be "plugged" into various development suites such as Microsoft Visual Basic 6, Microsoft C++ 6 and Borland Delphi 7.

A number of program templates can be created for this single Engine for different programming languages. The ActiveX implementation mentioned above, could have at least three: one for Visual Basic, one for Visual C++ and another for Delphi.

Program templates are implemented as simple text files that contain a number of preprocessor-type tags. These tags are used to designate template attributes and to mark where lists will be inserted containing the grammar's symbols and rules. The format of the tags designed to be versatile so that they can be used to create lists of constants, case statements, or whatever the developer needs.

This tool is integrated into the Builder Application, but, in reality, this tool could be implemented separately. In future implementations of the Builder – such as a command-line version for UNIX – the program template functional could be implemented as a separate application. In this case, the application would use a Compiled Grammar Table file as input.

3.2.2. Parameters

There are a number of preprocessor fields in each template which contain information about the implementation programming language, Engine name, author and instructions on how to create identifier names. The order of each field is unimportant, but all should be located at the start of the template file.

Most of the fields are informative and have no effect on how the skeleton program is created. However, the fields that are related to the format of the identifiers are important. The following diagram displays the syntax for the parameter fields.

[##TEMPLATE-NAME	<i>Template-Name</i>]
[##LANGUAGE	<i>ProgName</i>]
[##ENGINE-NAME	<i>Engine-Name</i>]
[##AUTHOR	<i>Author-Name</i>]
[##FILE-EXTENSION	<i>Extension</i>]
[##NOTES		
	...		
	##END-NOTES]
[##ID-CASE	(ProperCase Uppercase)]
[##ID-SEPARATOR	<i>Text</i>]
[##ID-SYMBOL-PREFIX	<i>Text</i>]
[##ID-RULE-PREFIX	<i>Text</i>]

Figure 3-8. Parameter Syntax

3.2.2.1. Template-Name Field

This field sets the formal name for the template. The name of the programming language and engine can be part of the template name as well as features of the template itself, but this is entirely up to the developer.

3.2.2.2. Language Field

This field contains the name of the template's programming language. The field is merely informative.

3.2.2.3. Engine-Name

The Engine-Name field designates the name of the specific implementation of the Engine that the template is designed for. Currently the field is merely informative, but in the future it may be used to categorize templates.

3.2.2.4. Author Field

This field designates the name of the template's author. It is merely informative.

3.2.2.5. File-Extension Field

When a skeleton program is created, the file will be saved with this extension.

3.2.2.6. Notes ... End-Notes Block

The Notes block allows the developer to describe the template in detail and add any information that can help the user. This can include, for instance, specific information about the template such as declared objects and dependencies.

3.2.2.7. ID-Case Field

When the GOLD Parser Builder creates identifiers for each constant (described below), the system can put each in either Proper Case or Uppercase. This value should be set to the standard conventions used in the programming language used to implement the Engine.

3.2.2.8. ID-Separator Field

For readability, many programming languages allow the use of characters, such as underscores and dashes, to be used in identifiers. The value of this field will be used in the constant names.

3.2.2.9. ID-Symbol-Prefix Field

The value of this field will be added to the front of each generated symbol constant.

3.2.2.10. ID-Rule-Prefix Field

The value of this field will be added to the front of each generated rule constant.

3.2.2.11. Example

```
##TEMPLATE-NAME 'Visual Basic - ActiveX DLL'  
##LANGUAGE 'Visual Basic'  
##ENGINE-NAME 'ActiveX DLL'  
##AUTHOR 'Devin Cook'  
##FILE-EXTENSION 'bas'  
##NOTES  
This template creates a Visual Basic skeleton program for  
use with the ActiveX DLL. The code will work with both  
Visual Basic 5 and 6.  
##END-NOTES  
##ID-CASE Propercase  
##ID-SEPARATOR '_'  
##ID-SYMBOL-PREFIX 'Symbol'  
##ID-RULE-PREFIX 'Rule'
```

3.2.3. Lists

3.2.3.1. Symbol Lists

Since the Compiled Grammar Table file contains the grammar's symbol table, it is often useful to list each symbol, their name and table index, within the skeleton program.. This "list" can be in the form of an enumerated constant definition, case statements, or anything else that the developer needs. The GOLD Parser Builder will scan a template file and insert a list of symbols when it locates a "symbols" block.

Essentially, a Symbol List contains a text block which will be used to create the elements of the list. When a "list" is created, this text block will be printed to the skeleton program for each symbol in the grammar's symbol table. The following diagram shows the format used to denote a Symbol Lists. The meaning of the `##Delimiter` field is described below.

```
##SYMBOLS  
[ ##DELIMITER Value ]  
...  
##END-SYMBOLS
```

Figure 3-9. Symbol List Syntax

Each time the text block is printed, a number of local "variables" will set to the corresponding value within the symbol table. The meaning of each variable is described in section 3.2.5.

3.2.3.2. Rule List

A rule list is used by the developer to specify where the Builder will insert a list of rules into the skeleton program. The notation allows the developer to specify anything from an enumerated constant list to a list of "case" statements. The following diagram shows the format used for to denote a Rule List. The notation is almost identical to that of symbol lists.

```
##RULES
[ ##DELIMITER Value ]
...
##END-RULES
```

Figure 3-10. Rule List Syntax

3.2.3.3. List Variables

When the Builder creates a symbol list or rule list the developer can specify a number of variables within the program template. The list field will create multiple rows for each item in the list. Each variable will, consequently, be set to the appropriate value for each item in the list. All variables are delimited by percent signs "%".

3.2.3.3.1. %ID% Variable

For each iteration in a Symbol List or Rule List, the value of this variable will contain a unique identifier generated by the Builder application. The content of the value will follow the de-facto standard used for identifiers in most programming languages – namely, an alphabet character followed by zero or more alphanumeric characters. The value of the ID variable is also dictated by the template parameters described above.

3.2.3.3.2. %Value% Variable

This variable simply contains the decimal-value index for the symbol or rule within their respective tables.

3.2.3.3.3. %Delimiter% Variable

This variable is used to create lists where a delimiter is used between each item. The value of this variable is set with the ##Delimiter option. For the last item in the list, the value of the Delimiter variable will be set to an empty string.

3.2.3.3.4. %Description% Variable

The identifier generated by the Builder may not be able to describe the symbol or rule in enough detail to aid the developer. As a result, the Description variable is used to insert comments into the generated program. Basically, the value of the variable will match the format used to define the symbol or rule in the original grammar. In the case of symbols, nonterminals will be delimited by angle-brackets and terminals will be delimited by single quotes if needed. For Rules, the text will contain the Backus-Naur Form representation of the rule.

3.2.3.3.5. %Description.XML% Variable

In the case with the .NET languages, such as Visual Basic .NET and C#, a hypertext version of the description can be used to create inline documentation. The content of this variable is essentially identical to that of the normal description variable. However, in this case, the formatting of the text follows the XML encoding. For instance, if a variable contains the value "<Statement>" in the description field, this variable will contain "<Statement>".

3.2.3.4. Examples

3.2.3.4.1. Overview

The following examples demonstrate the text created for Symbol Lists and Rule Lists. To give the examples meaningful results, each makes use of the grammar listed below.

```
<Stms> ::= <Stm> <Stms>
        | <Stm>

<Stm>  ::= if <Exp> then <Stms> end
        | Read Id
        | Write <Exp>

<Exp>  ::= Id '+' <Exp>
        | Id '-' <Exp>
        | Id
```

3.2.3.4.2. C++ Enumerated Constants

```
enum SymbolConstants
{
  ##SYMBOLS
  ##DELIMITER ' ',''
  %ID% = %Value%%Delimiter% // %Description%
  ##END-SYMBOLS
};

enum RuleConstants
{
  ##RULES
  ##DELIMITER ' ',''
  %ID% = %Value%%Delimiter% // %Description%
  ##END-RULES
};
```



```

enum SymbolConstants
{
    SYMBOL_EOF          = 0,  // (EOF)
    SYMBOL_ERROR        = 1,  // (Error)
    SYMBOL_WHITESPACE   = 2,  // (Whitespace)
    SYMBOL_MINUS        = 3,  // '-'
    SYMBOL_PLUS         = 4,  // '+'
    SYMBOL_END          = 5,  // end
    SYMBOL_ID           = 6,  // Id
    SYMBOL_IF           = 7,  // if
    SYMBOL_READ         = 8,  // Read
    SYMBOL_THEN         = 9,  // then
    SYMBOL_WRITE        = 10, // Write
    SYMBOL_EXP          = 11, // <Exp>
    SYMBOL_STM          = 12, // <Stm>
    SYMBOL_STMS         = 13, // <Stms>
};

enum RuleConstants
{
    RULE_STMS           = 0,  // <Stms> ::= <Stm> <Stms>
    RULE_STMS2          = 1,  // <Stms> ::= <Stm>
    RULE_STM_IF_THEN_END = 2,  // <Stm> ::= if <Exp> then <Stms> end
    RULE_STM_READ_ID    = 3,  // <Stm> ::= Read Id
    RULE_STM_WRITE      = 4,  // <Stm> ::= Write <Exp>
    RULE_EXP_ID_PLUS    = 5,  // <Exp> ::= Id '+' <Exp>
    RULE_EXP_ID_MINUS   = 6,  // <Exp> ::= Id '-' <Exp>
    RULE_EXP_ID         = 7,  // <Exp> ::= Id
};

```

3.2.3.4.3. C++ Switch Statement

This example assumes that the value of 'index' contains the index of the rule

```

switch (index)
{
    ##RULES
    case %ID% :
        // %Description%
        break;

    ##END-RULES
}

```




```
switch (index)
{
    case RULE_STMS
        // <Stms> ::= <Stm> <Stms>
        break;

    case RULE_STMS2
        // <Stms> ::= <Stm>
        break;

    case RULE_STM_IF_THEN_END
        // <Stm> ::= if <Exp> then <Stms> end
        break;

    case RULE_STM_READ_ID
        // <Stm> ::= Read Id
        break;

    case RULE_STM_WRITE
        // <Stm> ::= Write <Exp>
        break;

    case RULE_EXP_ID_PLUS
        // <Exp> ::= Id '+' <Exp>
        break;

    case RULE_EXP_ID_MINUS
        // <Exp> ::= Id '-' <Exp>
        break;

    case RULE_EXP_ID
        // <Exp> ::= Id
        break;
};
```

3.2.3.4.4. Visual Basic Enumerated Constants

```
Enum SymbolsConstants
##SYMBOLS
  %ID% = %Value% ' %Description%
##END-SYMBOLS
End Enum

Enum RuleConstants
##RULES
  %ID% = %Value% ' %Description%
##END-RULES
End Enum
```



```
Enum SymbolConstants
  Symbol_Eof      = 0 ' (EOF)
  Symbol_Error    = 1 ' (Error)
  Symbol_Whitespace = 2 ' (Whitespace)
  Symbol_Minus    = 3 ' '-'
  Symbol_Plus     = 4 ' '+'
  Symbol_End      = 5 ' end
  Symbol_Id       = 6 ' Id
  Symbol_If       = 7 ' if
  Symbol_Read     = 8 ' Read
  Symbol_Then     = 9 ' then
  Symbol_Write    = 10 ' Write
  Symbol_Exp      = 11 ' <Exp>
  Symbol_Stm      = 12 ' <Stm>
  Symbol_Stms     = 13 ' <Stms>
End Enum

Enum RuleConstants
  Rule_Stms      = 0 ' <Stms> ::= <Stm> <Stms>
  Rule_Stms2     = 1 ' <Stms> ::= <Stm>
  Rule_Stm_If_Then_End = 2 ' <Stm> ::= if <Exp> then <Stms> end
  Rule_Stm_Read_Id = 3 ' <Stm> ::= Read Id
  Rule_Stm_Write  = 4 ' <Stm> ::= Write <Exp>
  Rule_Exp_Id_Plus = 5 ' <Exp> ::= Id '+' <Exp>
  Rule_Exp_Id_Minus = 6 ' <Exp> ::= Id '-' <Exp>
  Rule_Exp_Id     = 7 ' <Exp> ::= Id
End Enum
```

3.2.3.4.5. Visual Basic Select Case Statement

This example assumes that the value of 'Index' contains the index of the rule.

```
Select Case Index
##RULES
  Case %ID%
    ' %Description%

##END-RULES
End Select
```



```
Select Case Index
  Case Rule_Stms
    ' <Stms> ::= <Stm> <Stms>

  Case Rule_Stms2
    ' <Stms> ::= <Stm>

  Case Rule_Stm_If_Then_End
    ' <Stm> ::= if <Exp> then <Stms> end

  Case Rule_Stm_Read_Id
    ' <Stm> ::= Read Id

  Case Rule_Stm_Write
    ' <Stm> ::= Write <Exp>

  Case Rule_Exp_Id_Plus
    ' <Exp> ::= Id '+' <Exp>

  Case Rule_Exp_Id_Minus
    ' <Exp> ::= Id '-' <Exp>

  Case Rule_Exp_Id
    ' <Exp> ::= Id

End Select
```

4. Compiled Grammar Table File

4.1. Introduction

The Compiled Grammar Table file is a file format that was designed to store the parse tables and other relevant information constructed by the Builder.

It is important to differentiate between the file's format and the file's content. When a document is saved to a file (for instance, a Microsoft Word file) it can be later retrieved and the work continued. This is possible since the application knows how information is stored and can, therefore, read its contents reliably. On the other hand, the actual content of the document can differ greatly from one file to the next (i.e. a report on Sacramento to an essay on parsing techniques).

4.1.1. Design Considerations

The file format used to save the Compiled Grammar Tables was based on following principles:

1. The file will be written to only once when it is created. Afterwards, information will only be read sequentially starting at the start of the file.
2. The format should be easy to implement on numerous platforms. In other words, to be very simple structurally.

3. The file structure should allow data structures to be added or expanded as needed in the future. The file will store structures such as integers, Unicode strings, bytes, but other types may be necessary in future versions.
4. The file structure should allow additional types of records to be added, if needed. In the future, the file format should be able to store different types of information besides the parse tables. These includes, for instance: sound files, pictures, source code, etc....

4.1.2. Why not XML?

Rather than designing a new file format, a number of existing formats were researched beforehand. The most notable of the formats is XML (W3C, 2003, Extensible Markup Language).

XML, like the commonly used HTML format, is a descendant of SGML (Standardized General Markup Language). However, XML is not a descendant of HTML and structural differences exist between the two formats. The XML syntax essentially allows a tree to be defined using normal ASCII characters. The format used to start and end logical objects is identical to the format used by HTML with three major exceptions: 1. All starting tags must have an ending tag, 2. Single tags (objects without sub-objects) end in `</>`, 3. All attributes must be delimited by double quotes

While the Builder and the system should be friendly to the XML format, given its popularity, XML has the following drawbacks which make it not the ideal format for this project.

1. The XML format is simple, but by its nature, not very compact. The tables created by the Builder can be extensive for complex grammars. The file can be compressed by any number of algorithms. However, this would place a high degree of burden on developers, and, therefore, not acceptable.

2. XML is a simple text file format, and can be easily edited by the developer. As a result, it would be possible to hack the tables to produce results not supported by the system. To insure backwards compatibility with future versions of GOLD, developers must work within the intended system structure.
3. XML does not allow different types of files to be embedded.

4.2. File Structure

4.2.1. Data Structures

The file format for the Compiled Grammar Table file is simple. The first data structure in the file is the File Header which contains a null-terminated Unicode string. This string contains the name and version of the type of information stored in the records. In the case of a Compiled Grammar Table file, the file will contain the text:

GOLD Builder Tables/v1.0

Since this string is stored in Unicode format, there are two bytes per each character give a total size of $(22+1)*2 = 46$ bytes. The header should be read as any normal Unicode string, since its size could change depending its contents. Following the Unicode string that contains the Header, the file will contain one or more records.



Figure 4-1. General File Structure

4.2.2. Records

Each logical record starts with a byte containing the value 77. This is the ASCII code for the letter "M", which, in turn, stands for multitype. So far, this is the only type of record stored in the file; however, it is possible, in the future, to add more types such as pictures, sounds, additional parse information, and other files.

Following the first byte, the row contains a 16-bit unsigned integer that contains the total number of entries in the record. The number is stored in Little Endian format, which means the least significant byte is stored first. This is the format used on the Intel family of processors and is the standard used by most file formats. Also, please note, this value is not the number of bytes to follow, but instead the number of different data types being stored. Developers should implement a simple "for-loop" to read the entries from the file.

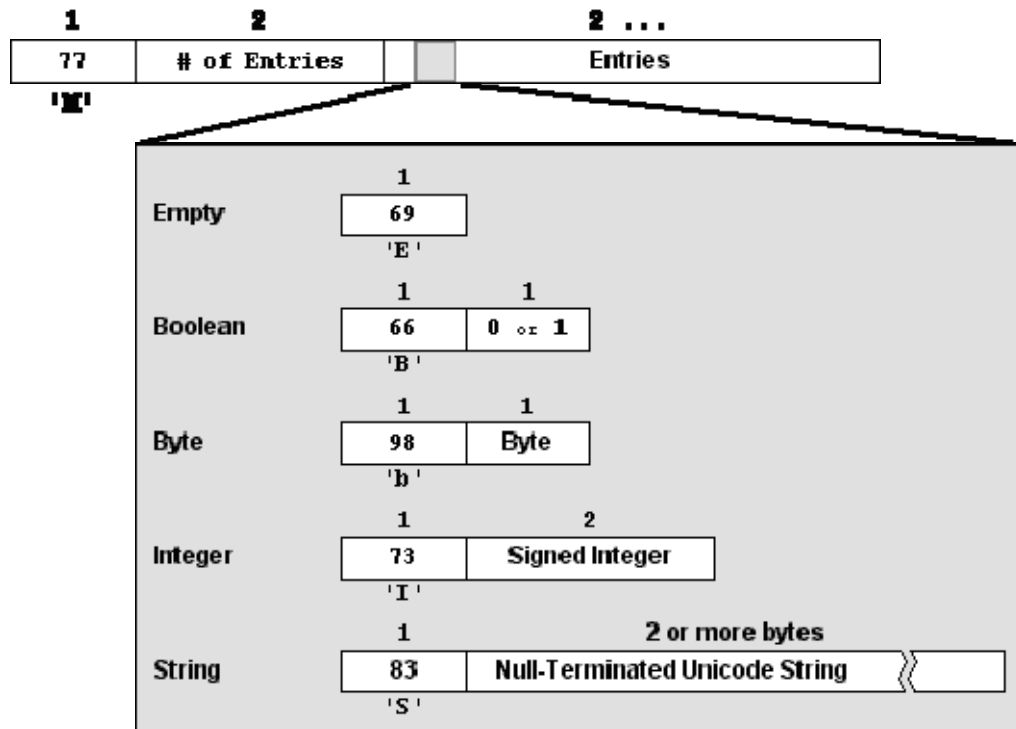


Figure 4-2. Record Contents

Each multitype row can contain any assortment of Boolean, Byte, Empty, Integer, and String entries. In future implementations of this file format, more entry types can be created.

4.2.3. Entries

Each multitype record consists of a series of entries which, in turn, can hold any number of data types. Preceding each entry is an identification byte which denotes the type of data which is stored. Based on this information, the appropriate number of bytes and the manner in which they are read can be deduced.

4.2.3.1. Empty

The entry only consists of an identification byte containing the value 69; the ASCII value of 'E'. This type of entry is used to represent a piece of information that has not been defined or is reserved for future use. It has no actual value and should be interpreted as a logical NULL.

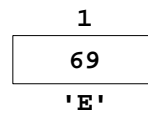


Figure 4-3. Empty Record Entry

4.2.3.2. Byte

A "byte" entry is preceded by a single byte containing the value 98; the ASCII value for 'b'. The next byte contains the actual information stored in the entry. This is a rather inefficient method for storing a mass number of bytes given that there is as much overhead as actual data. But, in the case of storing small numbers, it does save a byte over using an integer entry.

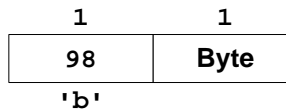


Figure 4-4. Byte Record Entry

4.2.3.3. Boolean

A Boolean entry is preceded by a byte containing the value 66; the ASCII value for 'B'. This entry is identical in structure to the Byte except the second byte will only contain a 1, for True, or a 0 for False.

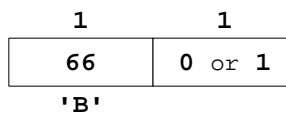


Figure 4-5. Boolean Record Entry

4.2.3.4. Integer

This is the most common entry used to store the Compiled Grammar Table information. Following the identification byte, the integer is stored using Little-Endian byte ordering. In other words, the least significant byte is stored first.



Figure 4-6. Integer Record Entry

4.2.3.5. String

A string entry starts with a byte containing the value 83, which is the ASCII value for "S". This is immediately followed by a sequence of 1 or more Unicode characters which are terminated by a null.

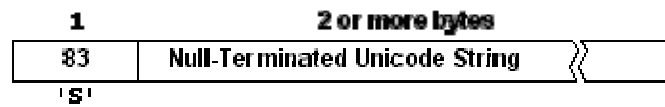


Figure 4-7. String Record Entry

4.3. File Content

4.3.1. Parameters

Byte	String	String	String	String	Boolean	Integer
80	Name	Version	Author	About	Case Sensitive?	Start Symbol
' P '						

Figure 4-8. Parameter Contents

A record containing parameter contents only occurs once in the Compiled Grammar Table file. It contains information about the grammar as well as attributes that affect how the grammar functions. The record is preceded by a byte field contains the value 80, the ASCII code for the letter 'P'.

Table 4-1. Parameter Content Details

Name	Type	Description
Name	String	Name of the grammar is important to know, but not vital to any of the parsing algorithms.
Version	String	This field contains the version of the grammar. Note that this field is a string, not an integer.
Author	String	Again, not a necessary field, but credit to the author should be included in any completed work.
About	String	This is a general-purpose field the designer can use to specify some notes about the grammar. Like, the name, version and author, this field does not affect the actual parsing algorithms.
Case Sensitive?	Boolean	This is an informative field that contains True if the grammar is case sensitive. Since the DFA tables contain upper and lowercase information, the value of this field has no effect on the system.
Start Symbol	Integer	This field contains the index of the grammar's start symbol in the Symbol Table. The start symbol is always a nonterminal which, when shifted by the LALR parser, indicates the source text is valid and complete.

4.3.2. Table Counts

Byte	Integer	Integer	Integer	Integer	Integer
84	Symbol Table	Character Set Table	Rule Table	DFA Table	LALR Table
'T'					

Figure 4-9. Table Count Contents

A single record will be stored that will contain entries for each of the table counts. These tables are the Symbol Table, Character Set Table, Rule Table, DFA Table and LALR table. The first field of the record contains a byte with the value 84 - the ASCII code for the letter 'T'

Each value contains the total number of objects for each of the listed tables. All indexing starts at 0 and ranges from 0 to the count - 1. After the record is read from the file, the developer can set the size of arrays and other data structures that will be used to store table information.

Table 4-2. Table Count Content Details

Item	Type	Description
Symbol Table	Integer	The number of symbols in the language.
Character Set Table	Integer	The number of character sets used by the DFA state table.
Rule Table	Integer	The number of rules in the language.
DFA Table	Integer	The number of Deterministic Finite Automata states.
LALR Table	Integer	The number of LALR States.

4.3.3. Character Set Table Member

Byte	Integer	String
67	Index	Characters

'C'

Figure 4-10. Character Set Table Member Contents

Each record describing a member in the Character Set Table is preceded by a byte field containing the value 67 - the ASCII value of "C". This table is used by the DFA State Table to store the valid characters for each edge in the DFA state machine. The file will contain one of these records for each character set used in the table. The Table Count record, which precedes any character set records, will contain the total number of entries.

Table 4-3. Character Set Table Entry Record Details

Name	Type	Description
Index	Integer	This field contains the index of the character set in the Character Set Table. The character set should be stored into the table at this value.
Characters	String	This field contains the actual character set. If the grammar is not case sensitive, the character set will not contain any uppercase letters, since they will not be required. Whether or not the grammar is case sensitive is indicated in the parameter record

4.3.4. Symbol Table Member

Byte	Integer	String	Integer
83	Index	Name	Kind

' S '

Figure 4-11. Symbol Table Member Contents

Each symbol in the Symbol Table will be stored to a record. The first entry will be a byte containing the value 83 - the ASCII value of "S". The file will contain one of these records for each symbol in the grammar. The Table Count record, which precedes any symbol records, will contain the total number of symbols.

Table 4-4. Symbol Table Member Content Details

Item	Type	Description
Index	Integer	This parameter holds the index of the symbol in the Symbol Table. The symbol should be stored directly at this Index.
Name	String	The name of the symbol is stored as a Unicode string.
Kind	Integer	This number denotes the kind (or type) of the symbol. The description of each constant is described below:

Table 4-5. Symbol 'Kind' Constants

Value	Description
0	Normal Nonterminal
1	Normal Terminal
2	Whitespace Terminal
3	End Character - End of File. This symbol is used to represent the end of the file or the end of the source input.
4	Start of a block quote
5	End of a block quote
6	Line Comment Terminal
7	Error Terminal. If the parser encounters an error reading a token, this kind of symbol can used to differentiate it from other terminal types.

4.3.5. Rule Table Member

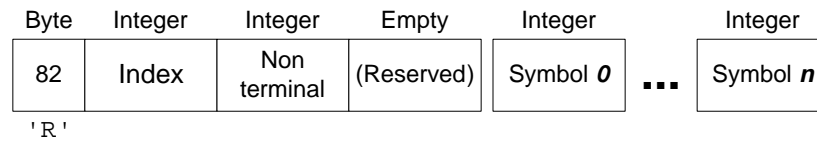


Figure 4-12. Rule Table Member Contents

Each rule in the Rule Table will be stored to a separate record. The first entry will be a byte containing the value 83 - the ASCII code for 'S'. The file will contain one of these records for each rule in the grammar. The Table Count record, which precedes any rule records, will contain the total number of rules.

Table 4-6. Rule Table Entry Record Details

Item	Type	Description
Index	Integer	This parameter holds the index of the rule in the Rule Table. The resulting rule should be stored at this Index.
Nonterminal	Integer	Each rule derives a single nonterminal symbol. This field contains the index of the symbol in the Symbol Table.
(Reserved)	Empty	This field is reserved for future use.
Symbol 0 ... n	Integer	The remaining entries in the record will contain a series of indexes to symbols in the Symbol Table. These constitute the symbols, both terminals and nonterminals, that define the rule. There can be 0 or more total symbols.

4.3.6. Initial States

Byte	Integer	Integer
73	DFA	LALR

' I '

Figure 4-13. Initial State Contents

The Initial State record only occurs once in the Compiled Grammar Table file. It will contain the initial states for both the DFA and LALR algorithms. The record is preceded by a byte entry containing the value 73, the ASCII code for the letter 'I'.

Table 4-7. Initial State Record Details

Item	Type	Description
DFA	Integer	The initial state in the Deterministic Finite Automata table. Normally, due to how the generation algorithm is implemented, this value should be 0
LALR	Integer	The initial state in the LALR state table. Like the DFA state table, this value should normally be 0.

4.3.7. DFA State Table Member

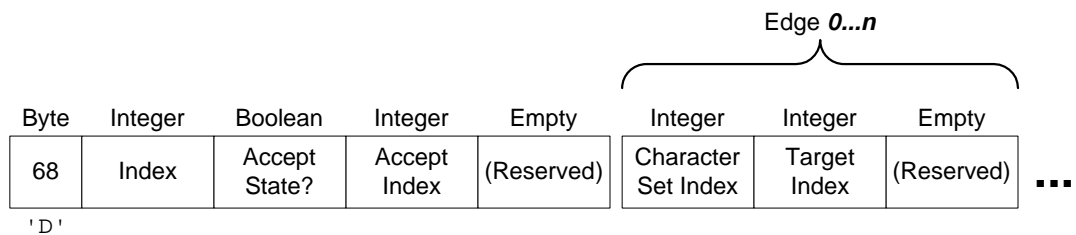


Figure 4-14. DFA State Table Member Contents

Each record that contains a DFA State Table Member is preceded by a byte field containing the value 68 - the ASCII code for "D". The file will contain one of these records for each state in the table. The Table Count record, which precedes any DFA records, will contain the total number of states.

The record contains information about the state itself: where it is located in the DFA State table and what symbols can be accepted (if any). Following this, there is a series of fields which describe each edge of the states. A DFA state can contain 0 or more edges, or links, to other states in the Table. These are organized in groups of 3 and constitute the rest of the record.

Table 4-8. DFA State Table Member Content Details

Item	Type	Description
Index	Integer	This parameter holds the index of the DFA state in the DFA State Table.
Accept State?	Boolean	Each DFA state can accept one of the grammar's terminal symbols. If the state accepts a terminal symbol, the value will be set to True

		and the Accept Index parameter will contain the symbol's index.
Accept Index	Integer	If the state accepts a terminal symbol, this field will contain the symbol's index in the Symbol Table. Otherwise, the value in this field should be ignored.
(Reserved)	Empty	This field is reserved for future use.
Edge 0...n		Described below.

Table 4-9. Edge 0...n

Value	Type	Description
Character Set Index	Index	Each edge contains a series of characters that are used to determine whether the Deterministic Finite Automata will follow it. The actual set of valid characters is not stored in this field, but, rather, an index in the Character Set Table.
Target Index	Index	Each edge is linked to state in the DFA Table. This field contains the index of that state.
(Reserved)	Empty	This field is reserved for future use.

4.3.8. LALR State Table Member

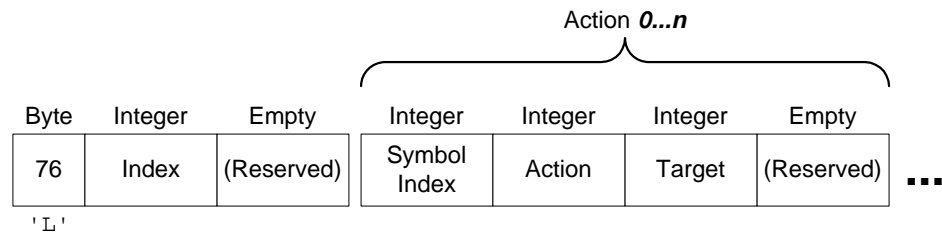


Figure 4-15. LALR State Table Member Contents

Each record that contains a state in the LALR State Table is preceded by a byte field containing the value 76 - the ASCII code for "L". The file will contain one of these records for each state in the table. The Table Count record, which precedes any LALR records, will contain the total number of states.

A LALR State contains a series of actions that are performed based on the next token. The record mostly consists of a series of fields (in groups of 4) which describe each of these actions.

Table 4-10. LALR State Table Member Content Details

Item	Type	Description
Index	Integer	This parameter holds the index of the state in the LALR State Table.
(Reserved)	Empty	This field is reserved for future use.
Action 0...n		Described below.

Table 4-11. Action 0...n

Value	Type	Description
Symbol Index	Index	This field contains the index in the Symbol Table.
Action	Index	This field contains a value that represents the action that LALR parsing engine is to take based on the symbol. These values are enumerated below.
Target	Integer	Depending on the value of the Action field, the target will hold different types of information.
(Reserved)	Empty	This field is reserved for future use.

Table 4-12. 'Action' Constants

Value	Name	Description
1	Shift	This action indicates the symbol is to be shifted. The Target field will contain the index of the state in the LALR State table that the parsing engine will advance to.
2	Reduce	This action denotes that the parser can reduce a rule. The Target field will contain the index of the rule in the Rule Table.
3	Goto	This action is used when a rule is reduced and the parser jumps to the state that represents the shifted nonterminal. The Target field will contain the index of the state in the LALR State table that the parsing engine will jump to after a reduction if completed.
4	Accept	When the parser encounters the Accept action for a given symbol, the source text is accepted as correct and complete. In this case, the Target field is not needed and should be ignored.

4.4. Example Compiled Grammar Table File

4.4.1. Example Grammar

To demonstrate how information is stored into a Compiled Grammar Table file, the following example grammar is used. Sections 4.4.2 and 4.4.3 show the content of the computed tables and how this information is stored in the file.

```
"Name"      = 'Example'
"Version"    = '1.0'
"Author"     = 'Devin Cook'
"About"      = 'N/A'

"Start Symbol" = <Stms>

<Stms> ::= <Stm> <Stms>
        | <Stm>

<Stm>  ::= if <Exp> then <Stms> end
        | Read Id
        | Write <Exp>

<Exp>  ::= Id '+' <Exp>
        | Id '-' <Exp>
        | Id
```

4.4.2. Table Content

The following text contains the content of the tables created for the example grammar. This information was "dumped" from the Builder Application. Please use this information in conjunction with the diagram displayed in the next section.

It should also be noted that the configurations displayed in the LALR State Table are not saved to the Compiled Grammar Table file. This information is created during the construction of the LALR State Table, but has no bearing on the behavior of the actual LALR parsing algorithm. Only the states' actions, reduce, shift, accept, and goto, are important.

```

=====
Symbol Table
=====

Index   Name
-----  -
0       (EOF)
1       (Error)
2       (Whitespace)
3       '-'
4       '+'
5       end
6       Id
7       if
8       Read
9       then
10      Write
11      <Exp>
12      <Stm>
13      <Stms>

=====
Rules
=====

```


Index	Name	::=	Definition
0	<Stms>	::=	<Stm> <Stms>
1	<Stms>	::=	<Stm>
2	<Stm>	::=	if <Exp> then <Stms> end
3	<Stm>	::=	Read Id
4	<Stm>	::=	Write <Exp>
5	<Exp>	::=	Id '+' <Exp>
6	<Exp>	::=	Id '-' <Exp>
7	<Exp>	::=	Id

=====
Character Set Table
=====

Index	Characters
0	{HT}{LF}{VT}{FF}{CR}{Space}{NBSP}
1	+
2	-
3	Ee
4	Ii
5	Rr
6	Tt
7	Ww
8	Nn
9	Dd
10	Ff
11	Aa
12	Hh

=====
DFA States
=====

Index	Description	Character Set
0	Goto 1	0
	Goto 2	1
	Goto 3	2
	Goto 4	3
	Goto 7	4
	Goto 10	5
	Goto 14	6
	Goto 18	7

1	Goto 1 Accept (Whitespace)	0
2	Accept '+'	
3	Accept '-'	
4	Goto 5	8
5	Goto 6	9
6	Accept end	
7	Goto 8 Goto 9	9 10
8	Accept Id	
9	Accept if	
10	Goto 11	3
11	Goto 12	11
12	Goto 13	9
13	Accept Read	
14	Goto 15	12
15	Goto 16	3
16	Goto 17	8
17	Accept then	
18	Goto 19	5
19	Goto 20	4
20	Goto 21	6
21	Goto 22	3
22	Accept Write	

```

=====
LALR States
=====

```

Index	Configuration/Action
0	if Shift 1 Read Shift 9 Write Shift 11 <Stm> Goto 13 <Stms> Goto 17
1	<Stm> ::= if · <Exp> then <Stms> end Id Shift 2 <Exp> Goto 7
2	<Exp> ::= Id · '+' <Exp> <Exp> ::= Id · '-' <Exp> <Exp> ::= Id · '-' Shift 3 '+' Shift 5 (EOF) Reduce 7 end Reduce 7 if Reduce 7 Read Reduce 7 then Reduce 7 Write Reduce 7
3	<Exp> ::= Id '-' · <Exp> Id Shift 2 <Exp> Goto 4
4	<Exp> ::= Id '-' <Exp> · (EOF) Reduce 6 end Reduce 6 if Reduce 6 Read Reduce 6 then Reduce 6 Write Reduce 6
5	<Exp> ::= Id '+' · <Exp> Id Shift 2 <Exp> Goto 6
6	<Exp> ::= Id '+' <Exp> · (EOF) Reduce 5 end Reduce 5 if Reduce 5 Read Reduce 5 then Reduce 5 Write Reduce 5
7	<Stm> ::= if <Exp> · then <Stms> end

```

      then Shift 8

8      <Stm> ::= if <Exp> then . <Stms> end
      if Shift 1
      Read Shift 9
      Write Shift 11
      <Stm> Goto 13
      <Stms> Goto 15

9      <Stm> ::= Read . Id
      Id Shift 10

10     <Stm> ::= Read Id .
      (EOF) Reduce 3
      end Reduce 3
      if Reduce 3
      Read Reduce 3
      Write Reduce 3

11     <Stm> ::= Write . <Exp>
      Id Shift 2
      <Exp> Goto 12

12     <Stm> ::= Write <Exp> .
      (EOF) Reduce 4
      end Reduce 4
      if Reduce 4
      Read Reduce 4
      Write Reduce 4

13     <Stms> ::= <Stm> . <Stms>
      <Stms> ::= <Stm> .
      if Shift 1
      Read Shift 9
      Write Shift 11
      <Stm> Goto 13
      <Stms> Goto 14
      (EOF) Reduce 1
      end Reduce 1

14     <Stms> ::= <Stm> <Stms> .
      (EOF) Reduce 0
      end Reduce 0

15     <Stm> ::= if <Exp> then <Stms> . end
      end Shift 16

16     <Stm> ::= if <Exp> then <Stms> end .
      (EOF) Reduce 2
      end Reduce 2

```

```
if Reduce 2
Read Reduce 2
Write Reduce 2

17      <S'> ::= <Stms> · (EOF)
        (EOF) Accept
```

4.4.3. File Representation

The following diagram displays how information will be stored in the Compiled Grammar Table File. To conserve space, only one of each type of record was selected. For readability, the ASCII values used to represent different types of records and entries are denoted by the character delimited by single quotes. For instance, 'P' represents the ASCII value for the character P, 80.

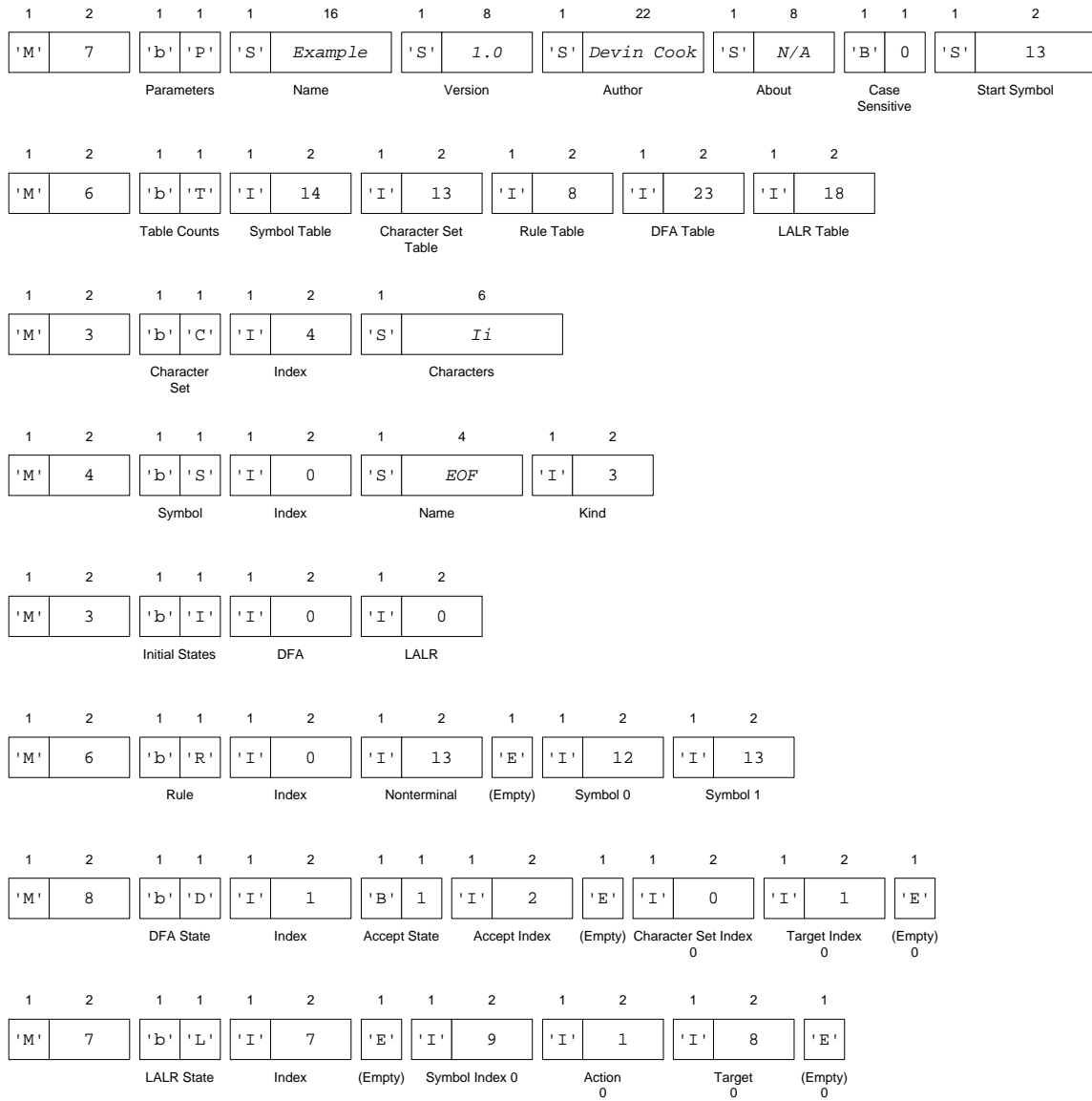


Figure 4-16. File Representation of Parse Information

5. The Engine Module

5.1. Overview

As different implementations of the Engine are created for different programming languages and integrated development environments, the approach and design will vary. Engine developers will use the programming language constructs and design principles which are best for the language which they are using. Each programming language has advantages and disadvantages and developers are encouraged to use the interface and format that is most appropriate for the language.

As a result, an implementation of the Engine written for Visual Basic 6 will differ greatly from one written for ANSI C.

An implementation of the GOLD Parser Engine was developed in conjunction with the Builder for the Visual Basic 6 programming language. The code was subsequently compiled into a Microsoft ActiveX DLL and made available with the Builder. Although Visual Basic 6 has well-known limitations

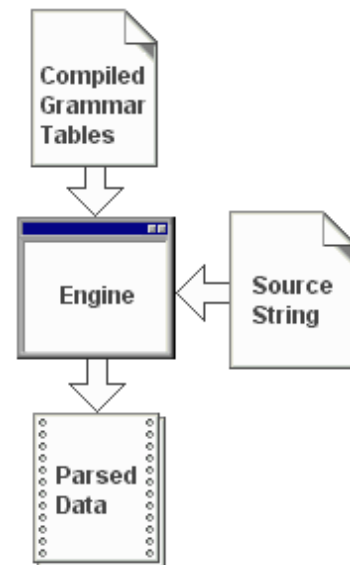


Figure 5-1. Parsing Engine Data Flow

affecting file access and object inheritance, the language is fairly easy to read by programmers of other languages.

The object hierarchy and interface of the Visual Basic Engine was designed to set a simple standard that could be used as a guide for future implementations. Essentially, it is designed around a central object aptly named "GOLDParser". This object performs all the parsing logic in the system including the LALR and DFA algorithms. The remaining objects are used for storage or to support the GOLDParser Object itself.

5.2. Objects

5.2.1. Object Hierarchy

The Visual Basic 6 implementation of the Engine contains five different objects which are used to represent information stored in the Compiled Grammar Table file and information created at parse time. The main functionality of the parser is performed by the GOLDParse Object. This object contains both the Rule Table and Symbol Table. To represent this information, the Symbol and Rule Objects were created.

During parse time, the GOLDParse Object will create instances of Reduction and Token Objects. The Reduction Object contains a series of tokens and is associated with the reduced rule. Each Token is associated with the corresponding symbol in the reduced rule.

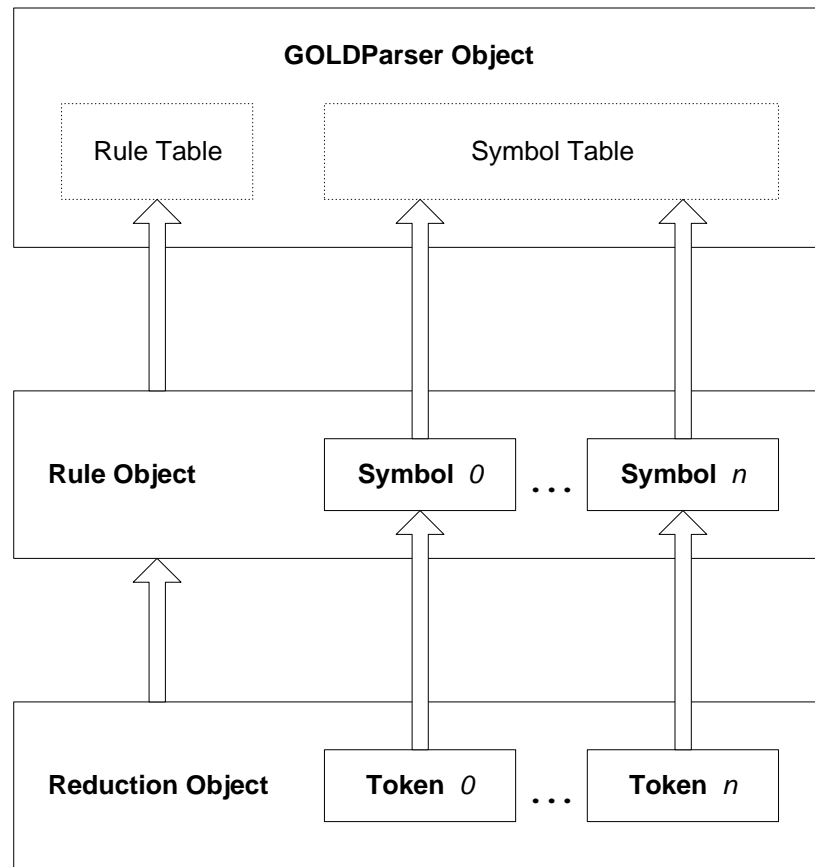


Figure 5-2. Object Hierarchy

5.2.2. Symbol Object

5.2.2.1. Overview

The Symbol Object is used to represent each terminal, special terminal and nonterminal in the GOLDParseObject's symbol table.

5.2.2.2. Properties

Property	Data Type	Access	Description
Kind	SymbolTypeConstants	Read only	Returns an enumerated data type that denotes the class of symbols that the object belongs to.
Name	String	Read only	Returns the name of the symbol.
TableIndex	Integer	Read only	Returns the index of the symbol in the GOLDParseObject's Symbol Table.
Text	String	Read only	Returns the text representation of the symbol. In the case of nonterminals, the name is delimited by angle brackets, special terminals are delimited by parenthesis and terminals are delimited by single quotes (if special characters are present).

5.2.2.3. Methods

This object does not contain any methods.

5.2.2.4. Symbol Type Constants

Message	Numeric Value	Details
SymbolTypeNonterminal	0	A normal nonterminal.
SymbolTypeTerminal	1	A normal terminal.
SymbolTypeWhitespace	2	This Whitespace symbols is a special terminal that is automatically ignored by the parsing engine. Any text accepted as whitespace is considered to be inconsequential and "meaningless".
SymbolTypeEnd	3	The End symbol is generated when the tokenizer reaches the end of the source text.
SymbolTypeCommentStart	4	This type of symbol designates the start of a block quote.
SymbolTypeCommentEnd	5	This type of symbol designates the end of a block quote.
SymbolTypeCommentLine	6	When the engine reads a token that is recognized as a line comment, the remaining characters on the line are automatically ignored by the parser.
SymbolTypeError	7	The Error symbol is a general-purpose means of representing characters that were not recognized by the tokenizer. In other words, when the tokenizer reads a series of characters that is not accepted by the DFA engine, a token of this type is created.

5.2.3. Rule Object

5.2.3.1. Overview

Each rule consists of a series of symbols, both terminals and nonterminals, and the single nonterminal that the rule defines. Rules are not creatable during runtime but are instead accessed through the GOLDParse object's RuleTable property.

5.2.3.2. Properties

Property	Data Type	Access	Description
RuleNonterminal	Symbol	Read only	Returns the head symbol of the rule.
SymbolCount	Integer	Read only	Returns the number of symbols that consist the body (right-hand-side) of the rule.
Symbols (<i>Index</i>)	Symbol	Read only	Returns one of the symbols that consist the body of the rule. The index of the symbol ranges from 0 to SymbolCount -1
TableIndex	Integer	Read only	Returns the index of the rule in the GOLDParse object's Rule Table.
Text	String	Read only	Returns the Backus-Naur representation of the rule.

5.2.3.3. Methods

This object does not contain any methods.

5.2.4. Token Object

5.2.4.1. Overview

The Token Object inherits from the Symbol Object and used to represent and organize parsed data. Unlike symbols, which are used to represent a category of terminals and nonterminals, tokens represent different instances of these symbols. For instance, the common "identifier" is a specific type of symbol, but can exist in various forms such as "Value1", "cat", "Sacramento", etc...

The information that is read from the source text is stored into the 'data' property which can be modified at the developer's will. These objects can be created at runtime.

5.2.4.2. Properties

Property	Data Type	Access	Description
Data	Variant	Read/Write	Returns/sets the information stored in the token. This can be either an standard data type or an object reference.
Kind	SymbolTypeConstants	Read only	Returns an enumerated data type that denotes the symbol class of the token.
Name	String	Read only	Returns the name of the token. This is equivalent to the parent symbol's name.
Parent Symbol	Symbol	Read/Write	Returns a reference the token's parent symbol.
TableIndex	Integer	Read only	Returns the index of the token's parent symbol in the GOLDParse object's symbol table.
Text	String	Read only	Returns the text representation of the token's parent symbol. In the case of nonterminals, the name is delimited by angle brackets, special terminals are delimited by parenthesis and terminals are delimited by single quotes (if special characters are present).

5.2.4.3. Methods

This object does not contain any methods.

5.2.5. Reduction Object

5.2.5.1. Overview

When the parsing engine has read enough tokens to conclude that a rule in the grammar is complete, it is 'reduced' and passed to the developer. Basically a 'reduction' will contain the tokens which correspond to the symbols of the rule. Tokens can represent actual data read from the file (a.k.a. terminals), but also may contain objects as well. Since a reduction contains the terminals of a rule as well as the nonterminals (reductions made earlier), the parser engine creates a 'parse tree' which contains a breakdown of the source text along the grammar's rules.

Essentially, when a reduction takes place, the `GOLDParser` object will respond with the `gpMsgReduce` message, create a new `Reduction` object, and then store it in the `CurrentReduction` property. This property can be read and then reassigned to another object if you decide to create your own objects. However, this is not required since the parse tree will be created using the produced `Reduction`.

The `Reduction` object is designed to be general purpose - able to store any rule, but at a cost. Since it is general purpose, there is additional overhead required that would not be needed in a specialized object. Invariably there will be numerous instances of the `Reduction` object, so it was designed to use as little memory as possible while maintaining basic functionality.

5.2.5.2. Properties

Property	Data Type	Access	Description
ParentRule	Rule	Read only	Returns the reference of the rule in the GOLDParse object's Rule Table that the Reduction represents.
TokenCount	Integer	Read only	Returns the number of tokens that consist the body of the Reduction. This will always be equal to the number of symbols in the ParentRule.
Tokens (<i>Index</i>)	Token	Read only	Returns one of the tokens that consist the body of the reduction. The index of the token ranges from 0 to TokenCount -1
Tag	Integer	Read/Write	This is a general purpose field that can be used at the developer's leisure.

5.2.5.3. Methods

This object does not contain any methods.

5.2.6. GOLDParse Object

5.2.6.1. Overview

The GOLDParse object is the primary class in the parser engine. Unlike the Rule, Reduction, Symbol and Token objects, the GOLDParse object performs all the required logic and operations needed to analyze the source text. Like all DFA and LALR engines, the GOLDParse relies on tables to drive the tokenizer and parser.

The developer interacts with the GOLDParse object by first loading the Compiled Grammar Table file and then subsequently calling the "Parse" method within a control loop. The "Parse" method returns codes that inform developer of the action(s) performed by the system and if any lexical or syntax errors were found.

Each time a rule is reduced, the system will create a Reduction Object. This object, which is described in the following sections, contains a series of tokens and is directly associated with the reduced rule. The tokens can contain actual text for the source string as well as other Reduction Objects (or even customized ones). This mechanism allows the construction of a parse tree.

Essentially the developer performs the following actions to initiate the GOLDParse object.

1. Call LoadCompiledGrammar() method to load a previously developed Compiled Grammar Table file
2. Call the appropriate method to open the source string to be parsed
3. Continue to call the Parse() method until the string is either accepted or an error occurs.

5.2.6.2. Properties

Property	Data Type	Access	Description
CurrentLineNumber	Integer	Read only	Returns the current line in the source text.
CurrentReduction	Object	Read/Write	Returns/sets the reduction made by the parsing engine. When a reduction takes place, this property will be set to a Reduction object which will store the reduced rule and its related tokens. This property may be reassigned a customized object if the developer so desires. The value of this property is only valid when the Parse() method returns the gpMsgReduction message.
CurrentToken	Token	Read only	Returns the token that is ready to be parsed by the engine. This property is only valid when the gpMsgTokenRead message is returned from the Parse method.
RuleTableCount	Integer	Read only	Returns the number of rules in the parser's internal Rule Table.
RuleTableEntry (Index)	Rule	Read only	Returns rule in the parser's Rule Table at the specified Index. The Index will range from 0 to RuleTableCount - 1.
SymbolTableCount	Integer	Read only	Returns the number of symbols in the parser's internal Symbol Table.
SymbolTableEntry (Index)	Symbol	Read only	Returns the symbol specified by Index. The Index will range from 0 to SymbolTableCount - 1.
TrimReductions	Boolean	Read/Write	Returns/sets the TrimReductions flag. When this property is set to True, the parser engine will automatically trim (i.e. remove) unneeded reductions from the parse tree.
TokenCount	Integer	Read only	Returns the number of tokens expected when a syntax error occurs.
Tokens (Index)	Token	Read only	Returns the token specified by Index. The Index will range from 0 to TokenCount - 1.

5.2.6.3. TrimReductions Property

The TrimReductions property is used to indicate whether the GOLD Parse Engine will automatically remove unneeded reductions from the parse tree. Although each rule is necessary to define the grammar, not all rules contain terminal symbols. As a result, when the system creates a reduction for

that rule, it does not contain any of the actual data being parsed. In some cases, this reduction is not needed in the parse tree since it only represents a single point on a branch. In particular, reductions which contain a single nonterminal can be eliminated from the tree without changing its meaning.

When the TrimReductions property is set to True, those rules with the following format will be automatically trimmed from the parse tree.

<Rule Name> ::= <Single Nonterminal>

The behavior of the TrimReductions property can best be demonstrated with a simple example. For this example, the following grammar will be used:

```
"Start Symbol" = <Expression>

ID = {Letter}{AlphaNumeric}*

<Expression> ::= <Mult Exp> '+' <Expression>
                | <Mult Exp> '-' <Expression>
                | <Mult Exp>

<Mult Exp>    ::= <Negate Exp> '*' <Mult Exp>
                | <Negate Exp> '/' <Mult Exp>
                | <Negate Exp>

<Negate Exp> ::= '-' <Value>
                | <Value>

<Value>      ::= ID
                | '(' <Expression> ')'
```

This grammar defines the operator precedence used in most programming languages for arithmetic expressions. If the source text

a + b * c

is parsed by the system, the parser will produce the following parse tree. The tree represents the source text broken down precisely using the grammar's rules.

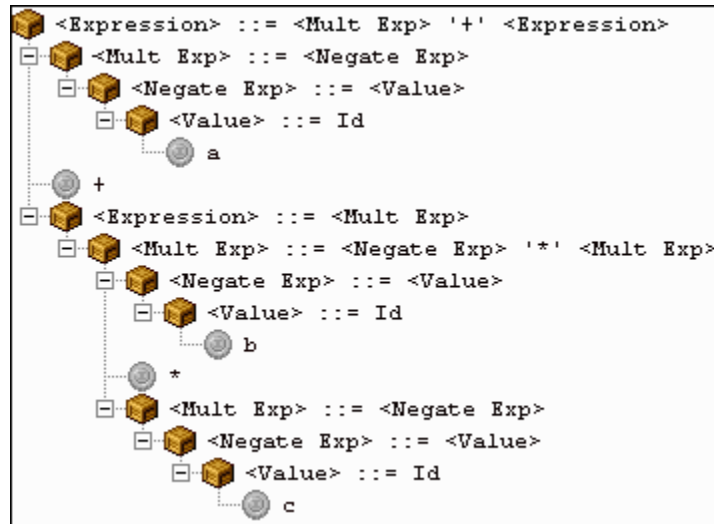


Figure 5-3. Parse Tree for "a+b*c"

For many grammars, such as those with a large number of operator precedence levels, the parse tree can quickly become complex. With the TrimReductions property set to True, the system will eliminate reductions where the rule contains a single nonterminal.

In the chart below, the highlighted reductions will be trimmed from the parse tree.

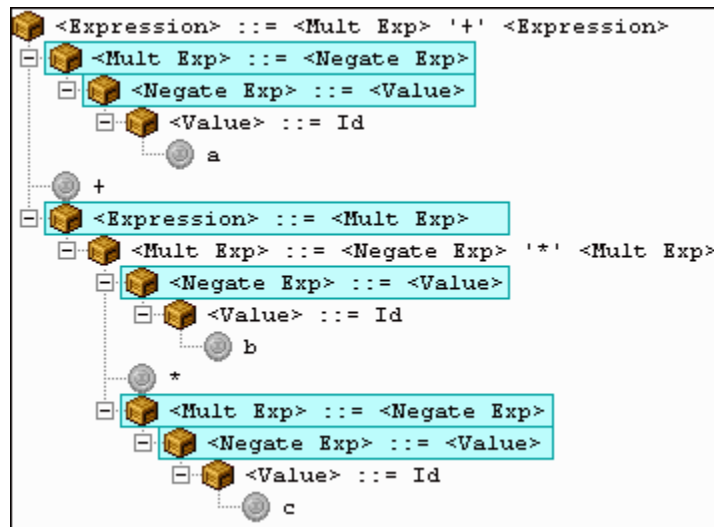


Figure 5-4. Reductions That Can Be Trimmed

This is performed 'behind-the-scenes' and is invisible to the developer. The `gpMsgReduction` message will not be generated for these trimmed reductions.

The resulting parse tree will contain far fewer reductions, but will not match the grammar verbatim. For instance, the `<Mult Exp>` rule contains the nonterminals `<Negate Exp>` and `<Mult Exp>`, but the reduction itself contains two `<Value>` rules instead.

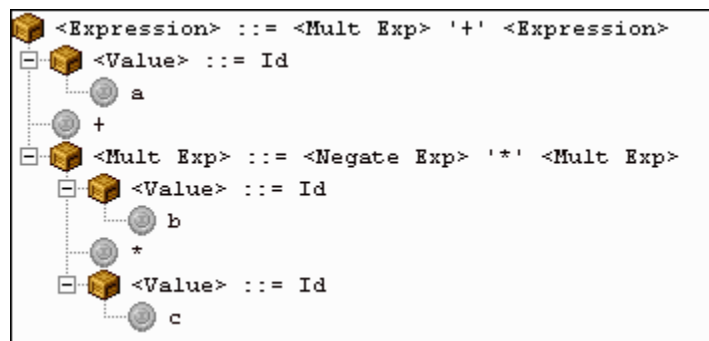


Figure 5-5. Parse Tree For "a+b*c" with TrimReductions

5.2.6.4. Methods

Method	Returns	Details
Clear	(None)	The GOLDParse is reset and the internal tables are cleared.
CloseFile	(None)	The file opened with the OpenFile method is closed.
LoadCompiledGrammar (FileName)	Boolean	If the Compiled Grammar Table file is successfully loaded the method returns True; otherwise False. This method must be called before any Parse calls are made.
OpenFile (FileName)	Boolean	Opens the FileName for parsing. If the file is successfully is open, the method returns True; otherwise False.
OpenTextString (SourceString)	Boolean	Opens the SourceString for parsing. If successful the method returns True; otherwise False.
Parameter (ParameterName)	String	Returns a string containing the value of the specified parameter. The ParameterName is the same as the parameters entered in the grammar's description. These include: Name, Version, Author, About, Case Sensitive and Start Symbol. If the name specified is invalid, this method will return an empty string.
Parse	GPMMessageConstants	Executes a parse. When this method is called, the parsing engine reads information from the source text (either a string or a file) and then reports what action was taken. This ranges from a token being read and recognized from the source, a parse reduction, or a type of error.
PopInputToken	Token	Removes the next token from the front of the parser's internal input queue.
PushInputToken (Token)	(None)	Pushes the token onto the front of the GOLDParse's internal input queue. It will be the next token analyzed by the parsing engine.
Reset	(None)	Resets the GOLDParse. The parser's internal tables are not affected.
ShowAboutWindow	(None)	Displays a simple window about the GOLDParse engine.

5.2.6.5. GOLD Parser Message Constants

Message	Numeric Value	Details
gpMsgTokenRead	1	Each time a token is read, this message is generated.
gpMsgReduction	2	When the engine is able to reduce a rule, this message is returned. The rule that was reduced is set in the GOLDParser's ReduceRule property. The tokens, that are reduced and correspond the rule's definition, are stored in the Tokens() property.
gpMsgAccept	3	The engine will returns this message when the source text has been accepted as both complete and correct. In other words, the source text was successfully analyzed.
gpMsgNotLoadedError	4	Before any parsing can take place, a Compiled Grammar Table file must be loaded.
gpMsgLexicalError	5	The tokenizer will generate this message when it is unable to recognize a series of characters as a valid token. To recover, pop the invalid token from the input queue.
gpMsgSyntaxError	6	Often the parser will read a token that is not expected in the grammar. When this happens, the Tokens() property is filled with tokens the parsing engine expected to read. To recover: push one of the expected tokens on the input queue.
gpMsgCommentError	7	The parser reached the end of the file while reading a comment. This is caused when the source text contains a "run-away" comment, or in other words, a block comment that lacks the delimiter.
gpMsgInternalError	8	Something is wrong, very wrong.

6. Testing and Development

6.1. LALR and DFA table generation

6.1.1. Overview

In the primary stage, the algorithms used to generate the LALR and DFA tables had to be tested and resulting tables verified as correct. To accomplish this, the tables displayed in books such as *Crafting a Compiler* by Charles N. Fischer and Richard J. LeBlanc Jr., and *Modern Compiler Implementation* by Andrew W. Appel. were tested directly against the generated tables. Both of these sources manually computed the tables and displayed the information in chart form. As a result, the tables created by this system could be directly compared to determine if the two tables were equivalent.

To parse test grammars, a simple ad-hoc scanner was developed to read Backus-Naur Form rules and regular expressions. For testing, the scanner only had to support a limited character set and a very rigid format for rules and regular expressions. The formatting and functionality of the final meta-language was not present.

In addition to the examples displayed in the two aforementioned books, a number of ad-hoc regular expressions and grammars were created. In each case, the tables were created manually and checked against the results generated by the LALR and DFA algorithms.

6.1.2. Book: Crafting a Compiler

In Crafting a Compiler by Charles N. Fischer and Richard J. LeBlanc Jr., a number of simple grammars are used to demonstrate different parsing algorithms. In particular, the authors defined a grammar called "G3" which defines very simple arithmetic expressions. This grammar was written using the GOLD Builder meta-language and analyzed. The grammar is as follows:

```
! G3
! Charles N. Fischer and Richard J. LeBlanc Jr
! Crafting a Compiler

Id = {Letter}+

"Start Symbol" = <E>

<E> ::= <E> '+' <T> | <T>

<T> ::= <T> '*' <P> | <P>

<P> ::= Id | '(' <E> ')'
```

The resulting LALR table was directly compared with the diagram that was manually computed and displayed in the book. The following diagram was scanned from Crafting a Compiler.

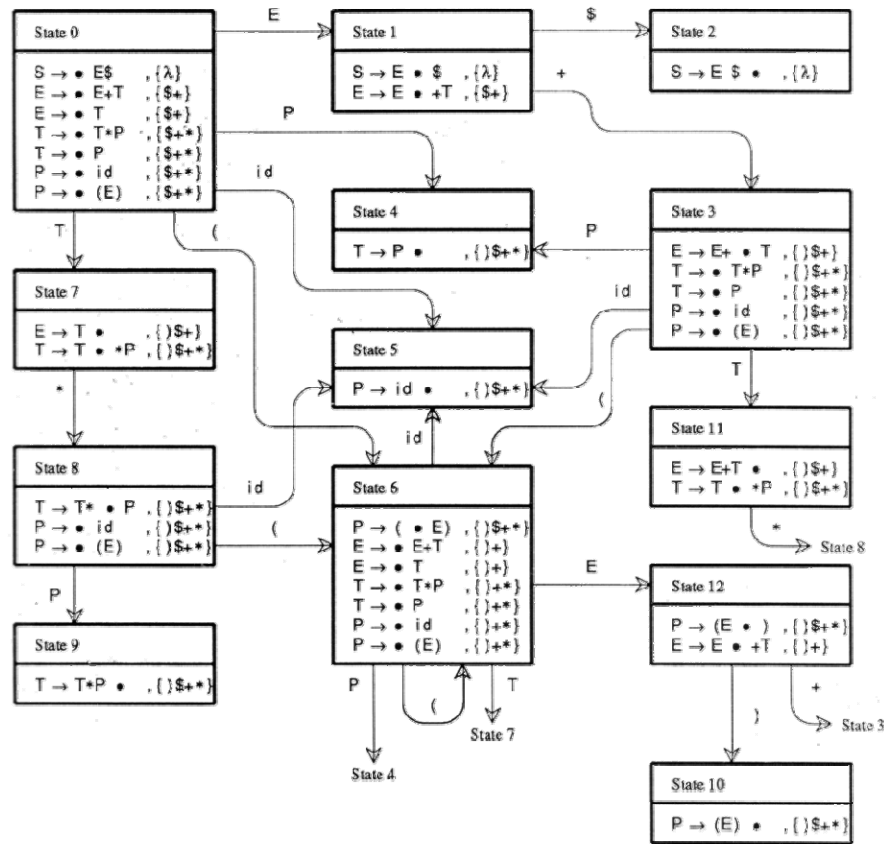


Figure 6-1. Diagram from Crafting a Compiler, Figure 6.22, pg 167

The following is the LALR state listing created by the GOLD Builder. The indexes between the states in the book and those created by the Builder do not match. However, this is the result of different orders in which each state was created, and, as a result, has no bearing on behavior of the resulting tables. Also, the state diagram from Crafting a Compiler contains an additional state. The authors accepted the virtual end-of-file token and advanced to a new state (State #2). The GOLD table, however, does not advance to a new state and, rather, accepts the grammar once the end-of-file token is read. Hence, the two tables are equivalent.

LALR States		
Index	Configuration/Action	Lookahead
0	<S'> ::= . <E> (EOF) <E> ::= . <E> '+' <T> <E> ::= . <T> <T> ::= . <T> '*' <P> <T> ::= . <P> <P> ::= . Id <P> ::= . '(' <E> ')' '(' Shift 1 Id Shift 2 <E> Goto 11 <P> Goto 6 <T> Goto 10	EOF + EOF + EOF * + EOF * + EOF * + EOF * +
1	<P> ::= '(' . <E> ')' <E> ::= . <E> '+' <T> <E> ::= . <T> <T> ::= . <T> '*' <P> <T> ::= . <P> <P> ::= . Id <P> ::= . '(' <E> ')' '(' Shift 1 Id Shift 2 <E> Goto 3 <P> Goto 6 <T> Goto 10	EOF) * +) +) +) * +) * +) * +) * +
2	<P> ::= Id . (EOF) Reduce 4 ')' Reduce 4 '*' Reduce 4 '+' Reduce 4	EOF) * +
3	<P> ::= '(' <E> . ')' <E> ::= <E> . '+' <T> ')' Shift 4 '+' Shift 5	EOF) * +) +
4	<P> ::= '(' <E> ')' . (EOF) Reduce 5 ')' Reduce 5 '*' Reduce 5 '+' Reduce 5	EOF) * +
5	<E> ::= <E> '+' . <T>	EOF) +

	<T> ::= . <T> '*' <P>	EOF)	*	+
	<T> ::= . <P>	EOF)	*	+
	<P> ::= . Id	EOF)	*	+
	<P> ::= . '(' <E> ')'	EOF)	*	+
	'(' Shift 1				
	Id Shift 2				
	<P> Goto 6				
	<T> Goto 7				
6	<T> ::= <P> .	EOF)	*	+
	(EOF) Reduce 3				
	')' Reduce 3				
	'*' Reduce 3				
	'+' Reduce 3				
7	<E> ::= <E> '+' <T> .	EOF)	+	
	<T> ::= <T> . '*' <P>	EOF)	*	+
	'*' Shift 8				
	(EOF) Reduce 0				
	')' Reduce 0				
	'+' Reduce 0				
8	<T> ::= <T> '*' . <P>	EOF)	*	+
	<P> ::= . Id	EOF)	*	+
	<P> ::= . '(' <E> ')'	EOF)	*	+
	'(' Shift 1				
	Id Shift 2				
	<P> Goto 9				
9	<T> ::= <T> '*' <P> .	EOF)	*	+
	(EOF) Reduce 2				
	')' Reduce 2				
	'*' Reduce 2				
	'+' Reduce 2				
10	<E> ::= <T> .	EOF)	+	
	<T> ::= <T> . '*' <P>	EOF)	*	+
	'*' Shift 8				
	(EOF) Reduce 1				
	')' Reduce 1				
	'+' Reduce 1				
11	<S'> ::= <E> . (EOF)				
	<E> ::= <E> . '+' <T>	EOF	+		
	(EOF) Accept				
	'+' Shift 5				

6.1.3. Book: Modern Compiler Implementation

A second book that used to test and design the LALR table generation algorithm was Modern Compiler Implementation by Andrew W. Appel. The grammar used in this book was named "Grammar 3.26".

```
! Grammar 3.26
! Andrew W. Appel
! Modern Compiler Implementation

"Start Symbol" = <S>

<S> ::= <V> '=' <E> | <E>

<E> ::= <V>

<V> ::= x | '*' <E>
```

The book contains a state chart for "Grammar 3.26". This chart uses the shortcut notation for LR-family parsers. In particular, shifts are represented by a letter "s" followed by the target state, reductions are represented with the letter "r" followed by the rule number, gotos are represented by the letter "g" followed by the target state, and an accept is the letter "a". The symbol which is used to denote the end-of-file marker is dollar sign '\$'.

	x	*	=	\$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

Figure 6-2. Diagram from Modern Compiler Implementation, Table 3.28 (revised), pg 66

The following is the output created by the GOLD Builder for "Grammar 3.26"

```

=====
LALR States
=====

Index      Configuration/Action      Lookahead
-----
0
  <S'> ::= . <S> (EOF)
  <S> ::= . <V> '=' <E>      EOF
  <S> ::= . <E>              EOF
  <E> ::= . <V>              EOF
  <V> ::= . x                EOF =
  <V> ::= . '*' <E>          EOF =
  '*' Shift 1
  x Shift 2
  <E> Goto 5
  <S> Goto 6
  <V> Goto 7

1
  <V> ::= '*' . <E>          EOF =
  <E> ::= . <V>              EOF =
  <V> ::= . x                EOF =
  <V> ::= . '*' <E>          EOF =
  '*' Shift 1
  x Shift 2
  <E> Goto 3
  <V> Goto 4

2
  <V> ::= x .                EOF =
  (EOF) Reduce 3
  '=' Reduce 3

3
  <V> ::= '*' <E> .          EOF =

```


	(EOF) Reduce 4	
	'=' Reduce 4	
4	<E> ::= <V> . (EOF) Reduce 2 '=' Reduce 2	EOF =
5	<S> ::= <E> . (EOF) Reduce 1	EOF
6	<S'> ::= <S> . (EOF) (EOF) Accept	
7	<S> ::= <V> . '=' <E> <E> ::= <V> . '=' Shift 8 (EOF) Reduce 2	EOF EOF
8	<S> ::= <V> '=' . <E> <E> ::= . <V> <V> ::= . x <V> ::= . '*' <E> '*' Shift 1 x Shift 2 <E> Goto 9 <V> Goto 4	EOF EOF EOF EOF
9	<S> ::= <V> '=' <E> . (EOF) Reduce 0	EOF

For easy comparison, the information created by the GOLD Builder was placed into a table following the same format in book. Like the results in the Crafting a Compiler book, the state numbers did not match up exactly. In addition, since GOLD indexes rules and states starting at zero, the reduction numbers do not match.

	x	*	=	\$	S	E	V
0	s2	s1			g6	g5	g7
1	s2	s1				g3	g4
2			r3	r3			
3			r4	r4			
4			r2	r2			
5				r1			
6				a			
7			s8	r2			
8	s2	s1				g9	g4
9				r0			

Figure 6-3. GOLD Builder LALR State Table

The following figure displays both the table from the Appel book and the table created by the GOLD Builder. The states in the GOLD Builder Table were rearranged to match the same order as the book. They match.

Appel LALR Table								GOLD Builder LALR Table							
	x	*	=	\$	S	E	V		x	*	=	\$	S	E	V
1	s8	s6			g2	g5	g3	0	s2	s1			g6	g5	g7
2				a				6				a			
3			s4	r3				3			s8	r2			
4	s8	s6				g9	g7	8	s2	s1				g9	g4
5				r2				5				r1			
6	s8	s6				g10	g7	1	s2	s1				g3	g4
7			r3	r3				4			r2	r2			
8			r4	r4				2			r3	r3			
9				r1				9				r0			
10			r5	r5				3			r4	r4			

Figure 6-4. Side by Side Comparison

6.1.4. Testing Correctness

After it was determined that table generation functioned correctly for small grammars, the algorithms needed to be tested with for more complex examples to confirm correctness and determine efficiency. A series of "real world" programming languages were written and tested. Due to the complexity of "real" programming languages, any flaws in either the system design or table generation would quickly become evident.

If the LALR table construction algorithm is operating correctly, the tables created by GOLD should be equivalent to those created by YACC. To test the correctness of GOLD, a very complex grammar was selected to be tested. A natural choice for complexity – and one that is widely known – is the ANSI C (Kernighan 1988) Programming Language. An ANSI C grammar was written in both the GOLD and YACC meta-languages. Both versions were designed to be identical. The rules defined in both versions are the same – both in structure and in the order in which they are defined.

The ANSI C tables for YACC were created on the Athena Mainframe at California State University, Sacramento. The tables for GOLD were created and exported on a local microcomputer. The resulting tables from both GOLD and YACC are identical. A side-by-side comparison of both tables is listed in Appendix D. This section also contains the ANSI C grammar written using the YACC meta-language.

In addition, the grammars for BASIC, COBOL (ANSI 1974), HTML (W3C, 1995, HTML), LISP (Wilensky 1984), Smalltalk, SQL, Visual Basic .NET (Microsoft 2003), XML (W3C, 2003, XML) were subsequently written and tested on actual source code. These grammars are listed in Appendix E.

6.2. Engine Development

6.2.1. Overview

After it was determined that the LALR and DFA table generation algorithms were functioning correctly, an implementation of the Engine was developed using the Visual Basic 6 Programming Language. Each of the grammars that were written to test the performance and robustness of the table generation algorithms, were subsequently used to test the Engine. After the tables were successfully created for each grammar listed above, a local copy of the Engine was used to parse actual source strings.

This approach allows the correctness of the grammar itself to be confirmed as well as the performance of the Engine. For instance, if an actual correct ANSI C program (which compiles) failed to be accepted by the parser, a flaw could exist in either the grammar declaration or the algorithms used to implement the parser.

In each case, the parse tree was checked to confirm that the source string was correctly analyzed by the Engine algorithms. Appendix F contains an actual ANSI C program that was tested and its resulting parse tree. The following sections contain source strings that were used to test the grammars used in Crafting and Compiler and Modern Compiler Implementation.

6.2.2. Book: Crafting a Compiler

This section contains two example test source strings that were used on the "G3" grammar that specified in Crafting a Compiler. This grammar is located in Section 6.1.2.

6.2.2.1. Example 1

```
a + b * c
```



```

+-- <E> ::= <E> '+' <T>
|
| +-- <E> ::= <T>
| |
| | +-- <T> ::= <P>
| | |
| | | +-- <P> ::= Id
| | | |
| | | | +-- a
| |
| | +-- +
| |
| | +-- <T> ::= <T> '*' <P>
| | |
| | | +-- <T> ::= <P>
| | | |
| | | | +-- <P> ::= Id
| | | | |
| | | | | +-- b
| | |
| | | +-- *
| |
| | +-- <P> ::= Id
| | |
| | | +-- c

```

6.2.2.2. Example 2

```
(a + b) * c + d
```



```

+-- <E> ::= <E> '+' <T>
|
+-- <E> ::= <T>
|
|   +-- <T> ::= <T> '*' <P>
|   |
|   |   +-- <T> ::= <P>
|   |   |
|   |   |   +-- <P> ::= '(' <E> ')'
|   |   |   |
|   |   |   |   +-- (
|   |   |   |   +-- <E> ::= <E> '+' <T>
|   |   |   |   |
|   |   |   |   |   +-- <E> ::= <T>
|   |   |   |   |   |
|   |   |   |   |   |   +-- <T> ::= <P>
|   |   |   |   |   |   |
|   |   |   |   |   |   |   +-- <P> ::= Id
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   +-- a
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   +-- +
|   |   |   |   |   |   |   |   +-- <T> ::= <P>
|   |   |   |   |   |   |   |   +-- <P> ::= Id
|   |   |   |   |   |   |   |   +-- b
|   |   |   |   |   |   |   |   +-- )
|   |   |   |   |   |   |   |   +-- *
|   |   |   |   |   |   |   |   +-- <P> ::= Id
|   |   |   |   |   |   |   |   +-- c
|   |   |   |   |   |   |   |   +-- +
|   |   |   |   |   |   |   |   +-- <T> ::= <P>
|   |   |   |   |   |   |   |   +-- <P> ::= Id
|   |   |   |   |   |   |   |   +-- d

```


6.2.3. Book: Modern Compiler Implementation

This section contains two example source strings that were tested on "Grammar 3.26". This grammar is specified in the book Modern Compiler Implementation and is listed in Section 6.1.2.

6.2.3.1. Example 1

```
x = x
```



```
+-- <S> ::= <V> '=' <E>
|
| +-- <V> ::= x
| | +-- x
| +-- =
| +-- <E> ::= <V>
| | +-- <V> ::= x
| | | +-- x
```

6.2.3.2. Example 2

```
* x = *** x
```



```

+-- <S> ::= <V> '=' <E>
|
+-- <V> ::= '*' <E>
|   +-- *
|   +-- <E> ::= <V>
|       |   +-- <V> ::= x
|       |   +-- x
+-- =
+-- <E> ::= <V>
|   +-- <V> ::= '*' <E>
|       +-- *
|       +-- <E> ::= <V>
|           +-- <V> ::= '*' <E>
|               +-- *
|               +-- <E> ::= <V>
|                   +-- <V> ::= '*' <E>
|                       +-- *
|                       +-- <E> ::= <V>
|                           +-- <V> ::= x
|                               +-- x

```


6.3. Meta-Language Development

Once the table generation algorithms and Engine functioned correctly, the temporary scanner that was used to read meta-grammars was replaced with a more advanced version using an internal copy of the Visual Basic Engine. In fact, the parser used to analyze a GOLD meta-grammar would be the GOLD Parser Engine itself. The meta-grammar for the GOLD meta-language can be found in Appendix C.

By replacing the temporary scanner, the meta-language was able to be expanded. The editions to the meta-language included block comments and the ability to add new lines to aid readability. These changes made it easier for more complex grammars to be written and subsequently tested.

7. Implementation

7.1. Builder Application

7.1.1. Overview

The Builder is the main component of the GOLD Parser. The main duty of the Builder is to read a source grammar written in the GOLD Meta-Language, produce the LALR and DFA parse tables, and, finally, save this information to Compiled Grammar Table file.

The Builder Application runs on the Windows 32-bit operating systems which include, but are not limited to, Windows 9x, Windows NT and Windows XP. As it is implemented, the application also contains a number of features which could have been implemented in different applications. Each feature was included to create an easy-to-use integrated development environment.

In addition to table construction, the Builder Application can create skeleton programs using the Program Templates - which are discussed in Chapter 3.2. Each template is stored in a subfolder of the Builder Application. This, however, is specific to this particular version of the Builder. In future versions, templates may be stored in the same folder as the main executable. For instance, if a UNIX version of the Builder is developed, templates may be available in a myriad of folders – depending on the user's settings.

The Builder Application can also interactively test a grammar using an integrated instance of the Visual Basic 6 Engine. Since both the Builder and the Visual Basic 6 Engine were constructed concurrently, they were integrated to benefit the developer. Different implementations of the Engine will not be allowed to be "plugged" into the Builder Application.

To implement a simple tool for testing grammars, a local copy of the Visual Basic Engine was compiled with the Builder source.

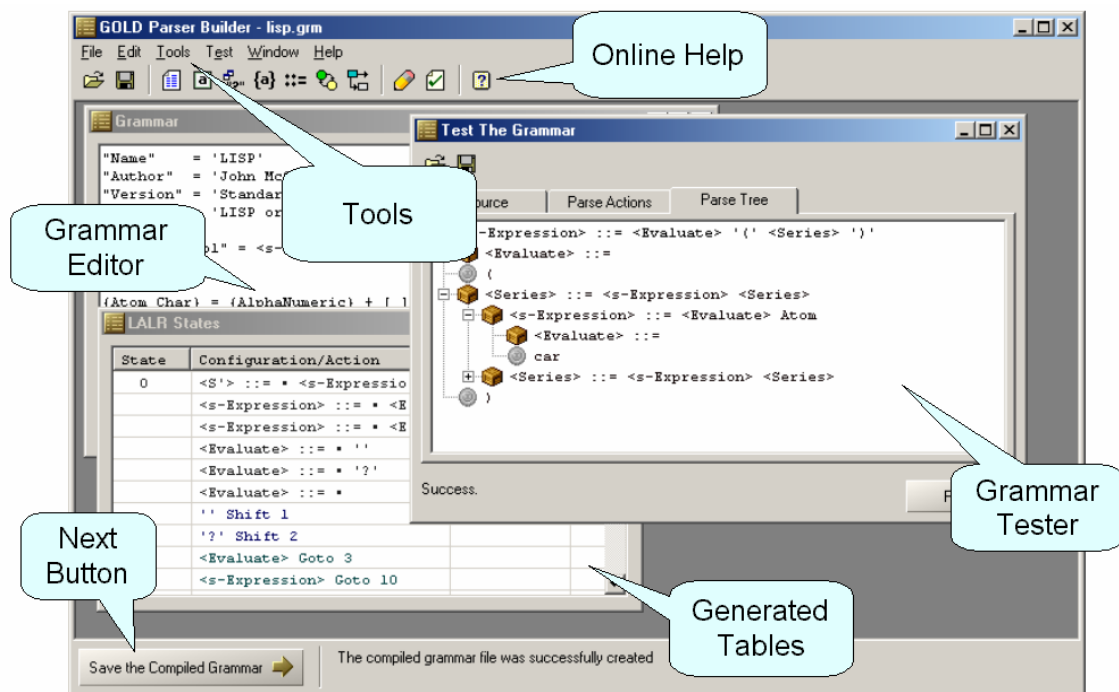


Figure 7-1. Application Layout

The flow of the application is broken down into three distinct steps that progress from a grammar to the completed parse tables. The developer advances by clicking the "Next" button which is located on

the bottom left side of the main window. This is similar to the interface commonly used by installation software and application "wizards".

Step 1 Enter the Grammar

During this step, the Builder checks the syntax of the grammar itself and prepares the system to compute the parse tables. If the grammar contains an error, it is reported to the user and the system resets.

Step 2 Compute The Tables

During this step, the Builder attempts to construct the LALR and DFA parse tables. Most conflicts occur during the construction of the LALR parse tables, and, as a result, these are constructed first.

Step 3 Save the Compiled Grammar

If no problems are found, the developer now has the ability to save the parse information to a Compiled Grammar Table file. If the developer clicks on the button at this point, the system will automatically display the "Save File" dialog window.

Since the parse tables are complete and ready to use, the developer has the ability to interactively test the grammar using the embedded parser engine. Also, the developer can export the grammar to different file formats (besides Compiled Grammar Table files), and create skeleton programs.

7.1.2. Grammar Edit Window

The GOLD Parser Builder contains a simple text editor. Currently, the editor does not contain any more functionality than Notepad. However, in future versions of the Builder, other features can be added to aid the developer. These can include color highlighting and line numbers

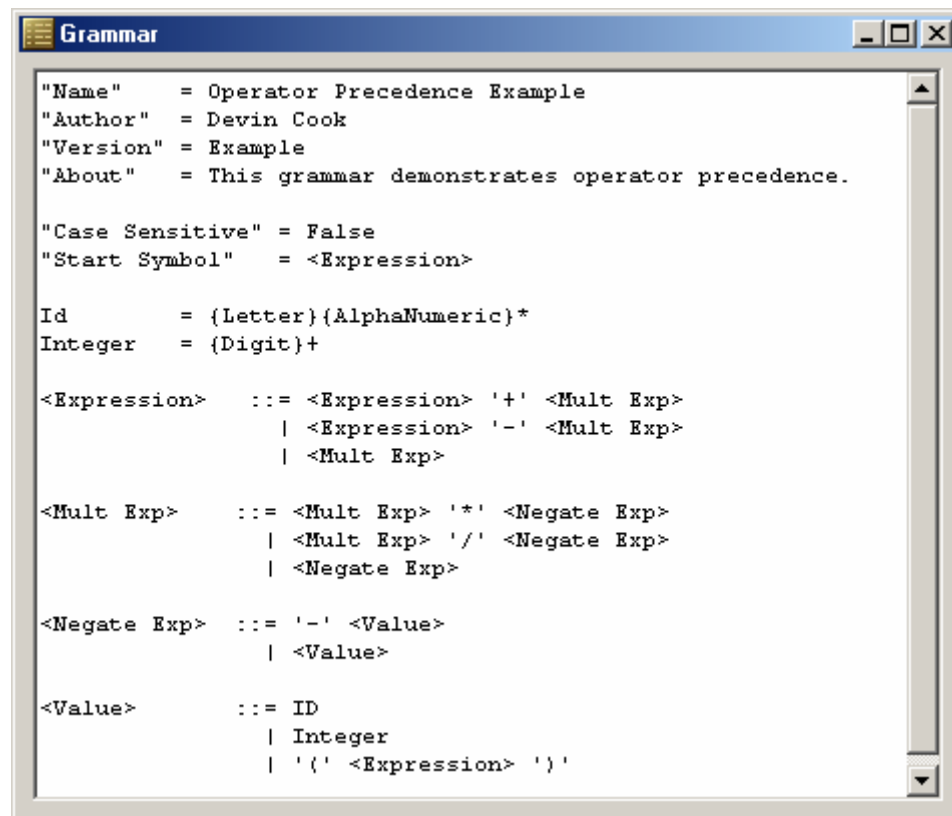
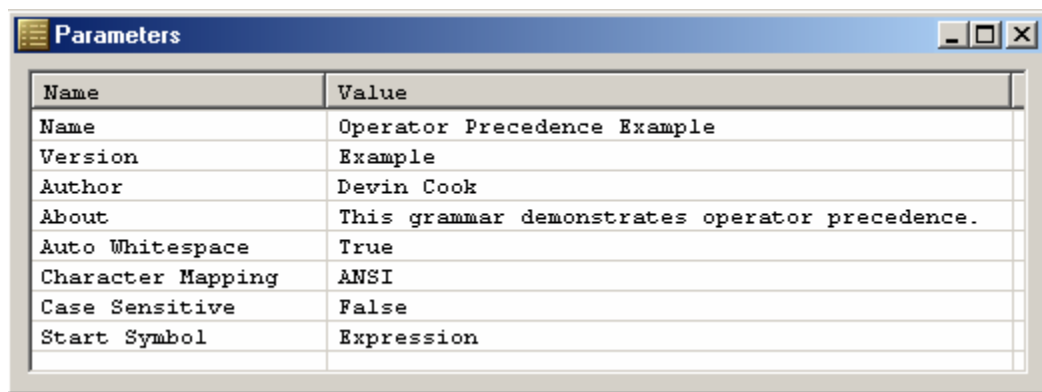


Figure 7-2. Grammar Edit Window

7.1.3. Parameter Window

The Parameter Window displays all attributes that were set by the grammar. These include, but are not limited to, the grammar's name, version, author and whether the language is case sensitive.



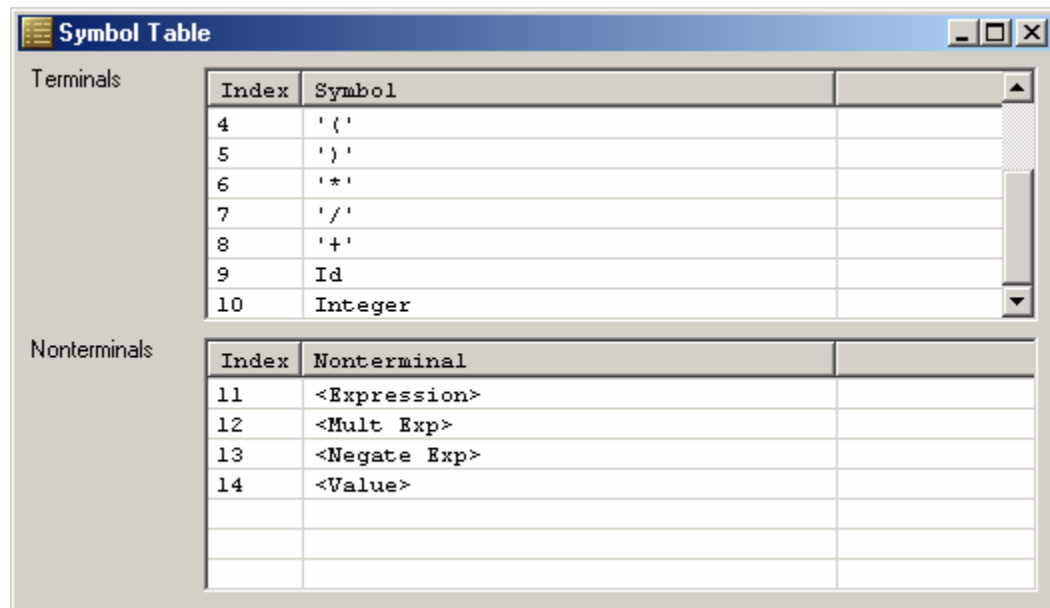
The screenshot shows a window titled "Parameters" with a table containing the following data:

Name	Value
Name	Operator Precedence Example
Version	Example
Author	Devin Cook
About	This grammar demonstrates operator precedence.
Auto Whitespace	True
Character Mapping	ANSI
Case Sensitive	False
Start Symbol	Expression

Figure 7-3. Parameter Window

7.1.4. Symbol Table

The Symbol Table Window allows the developer to review the grammar's symbols and reserved words. The system automatically sorts all the symbols by type, either terminal or nonterminal, and then by their respective names.



The image shows a window titled "Symbol Table" with two sections: "Terminals" and "Nonterminals". Each section contains a table with "Index" and "Symbol" or "Nonterminal" columns. The "Terminals" table lists symbols from index 4 to 10, including parentheses, asterisk, slash, plus, and identifiers. The "Nonterminals" table lists nonterminals from index 11 to 14, including expression, multiplication, negation, and value.

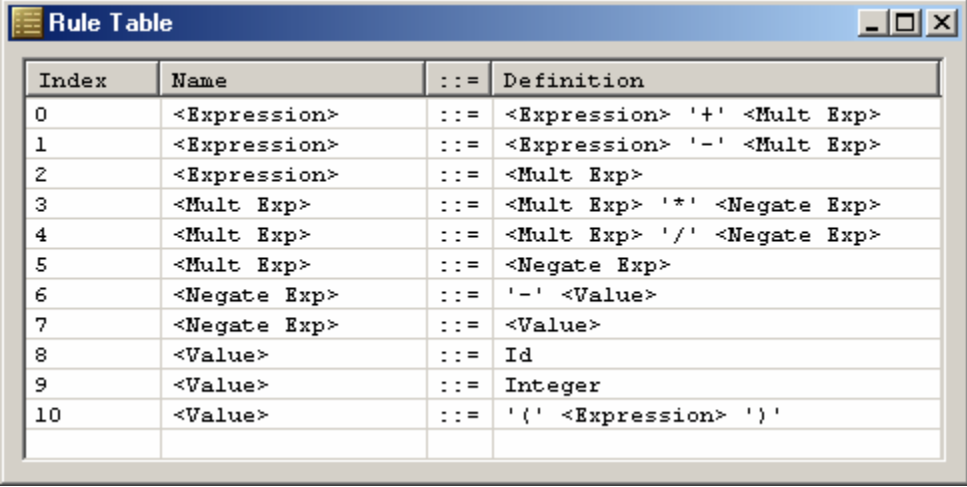
Terminals	
Index	Symbol
4	' ('
5	') '
6	'*'
7	'/'
8	'+'
9	Id
10	Integer

Nonterminals	
Index	Nonterminal
11	<Expression>
12	<Mult Exp>
13	<Negate Exp>
14	<Value>

Figure 7-4. Symbol Table Window

7.1.5. Rule Table Window

The Rule Table Window displays all the rules used in the grammar. The developer can review each rule to determine if they made any logistical errors in the grammar that may not have been apparent. This window also allows the developer to make sure that the grammar scanner worked correctly.



The screenshot shows a window titled "Rule Table" with a table containing 11 rows of grammar rules. The table has four columns: Index, Name, ::=, and Definition. The rules define the structure of expressions, including addition, subtraction, multiplication, division, negation, and values (Id, Integer, and parenthesized expressions).

Index	Name	::=	Definition
0	<Expression>	::=	<Expression> '+' <Mult Exp>
1	<Expression>	::=	<Expression> '-' <Mult Exp>
2	<Expression>	::=	<Mult Exp>
3	<Mult Exp>	::=	<Mult Exp> '*' <Negate Exp>
4	<Mult Exp>	::=	<Mult Exp> '/' <Negate Exp>
5	<Mult Exp>	::=	<Negate Exp>
6	<Negate Exp>	::=	'-' <Value>
7	<Negate Exp>	::=	<Value>
8	<Value>	::=	Id
9	<Value>	::=	Integer
10	<Value>	::=	'(' <Expression> ')'

Figure 7-5. Rule Table Window

7.1.6. Log Window

The Log Window displays a myriad of information. This includes general information about the number of symbols, which ones were defined implicitly, table counts, and any errors that occur.

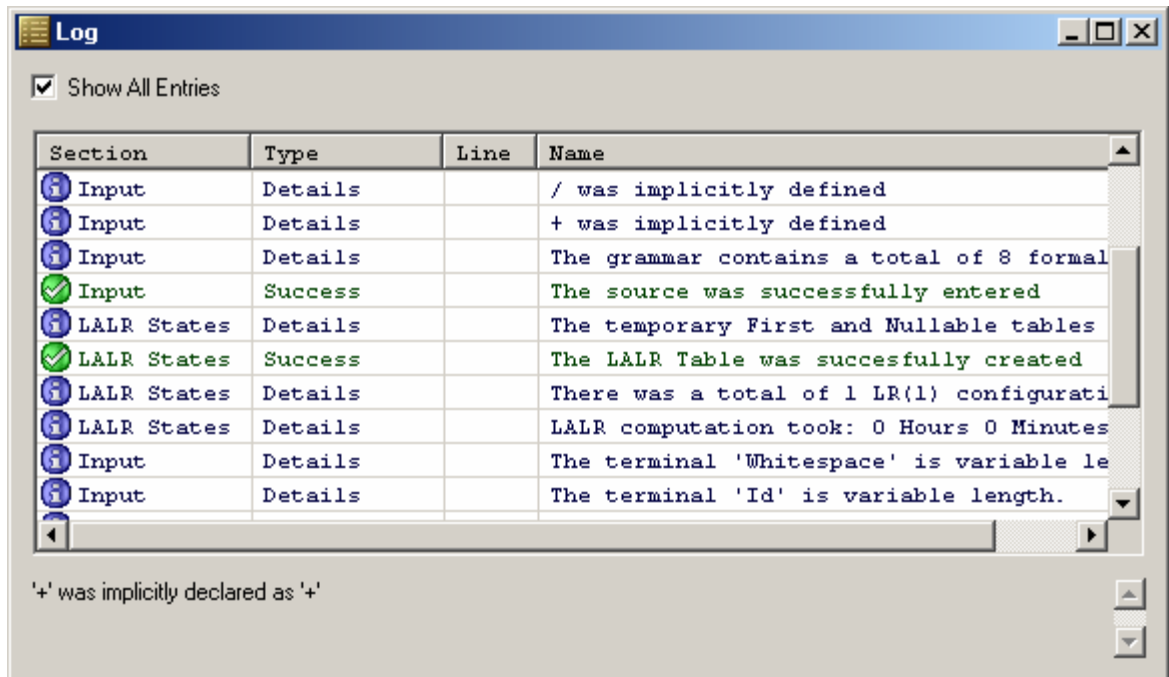
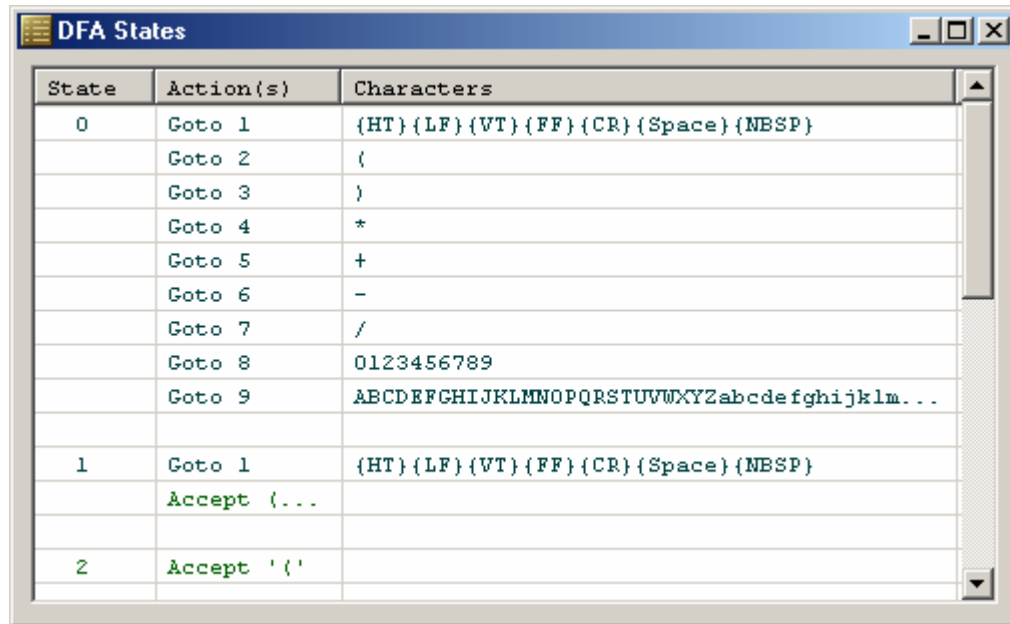


Figure 7-6. Log Window

7.1.7. DFA State Table Window

If a terminal is not functioning as expected, the developer can review the actual Deterministic Finite Automata. This window also allows students to view the actual information produced by the GOLD Parser Builder.



The image shows a window titled "DFA States" with a table containing DFA state information. The table has three columns: State, Action(s), and Characters. State 0 is the start state with transitions to states 1-9 for various characters. State 1 is an accepting state for the empty string. State 2 is an accepting state for the opening parenthesis '('.

State	Action(s)	Characters
0	Goto 1	{HT} {LF} {VT} {FF} {CR} {Space} {NBSP}
	Goto 2	{
	Goto 3	}
	Goto 4	*
	Goto 5	+
	Goto 6	-
	Goto 7	/
	Goto 8	0123456789
	Goto 9	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz...
1	Goto 1	{HT} {LF} {VT} {FF} {CR} {Space} {NBSP}
	Accept (...)	
2	Accept '('	

Figure 7-7. DFA State Table Window

7.1.8. LALR State Table Window

Most errors that occur in grammars are found in the LALR State Table. When the system analyzes a grammar and computes the parser tables, often shift-reduce and reduce-reduce conflicts are found. The LALR State Table Window allows the developer to review the produced states – in particular the state that contains the error.

Unfortunately, tracing LALR states can be tedious and challenging. In the future, the GOLD Parser Builder will contain tools for analyzing LALR conflicts.

LALR States		
State	Configuration/Action	Lookahead
	Integer Shift 4	
	<Expression> Goto 19	
	<Mult Exp> Goto 17	
	<Negate Exp> Goto 13	
	<Value> Goto 10	
1	<Negate Exp> ::= '-' ■ ...	EOF -) * / +
	<Value> ::= ■ Id	EOF -) * / +
	<Value> ::= ■ Integer	EOF -) * / +
	<Value> ::= ■ '(' <Expr...	EOF -) * / +
	'(' Shift 2	
	Id Shift 3	
	Integer Shift 4	
	<Value> Goto 18	
2	<Value> ::= '(' ■ <Expr...	EOF -) * / +
	<Expression> ::= ■ <Exp...	-) +

Figure 7-8. LALR State Table Window

7.1.9. Grammar Test Window

One of key features of the Builder Application is the Grammar Test Window. After the grammar has been successfully compiled by the Builder, the developer can check how Deterministic Finite Automata and LALR algorithms will analyze any number of test cases.

To test the behavior of the algorithms, a local embedded implementation of the Visual Basic 6 Engine is used. Although different implementations of the Engine exist, the behavior of the LALR and DFA algorithms is constant. Regardless of whether an Engine is written for Visual Basic, C++ or Java, the parse tables will create the same number of reductions and will accept or reject the same information. As a result, the specific details on how the Visual Basic 6 Engine was implemented, will have no effect on testing.

The Grammar Test Window contains three separate "tabs". The first tab allows the developer to enter or load a test string. In the screenshot below, the text string "a + b * c - d" was tested.

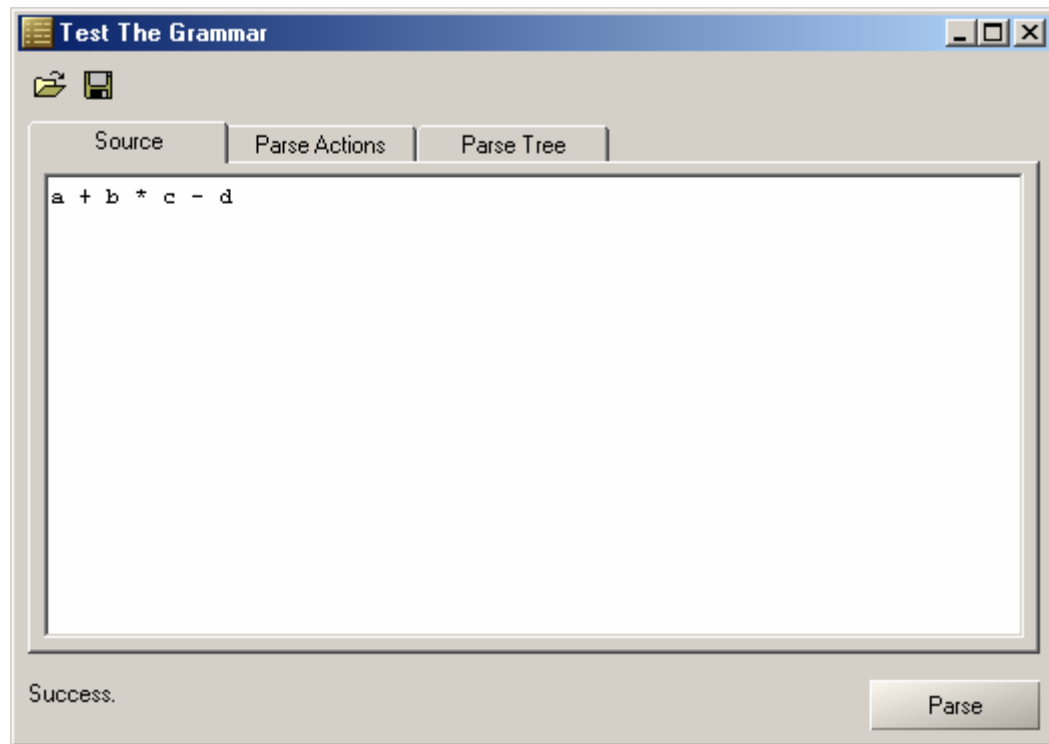


Figure 7-9. Grammar Test Window

After the string is ready, the developer can click on the "Parse" button to view the parse actions performed by the LALR Parsing Engine.

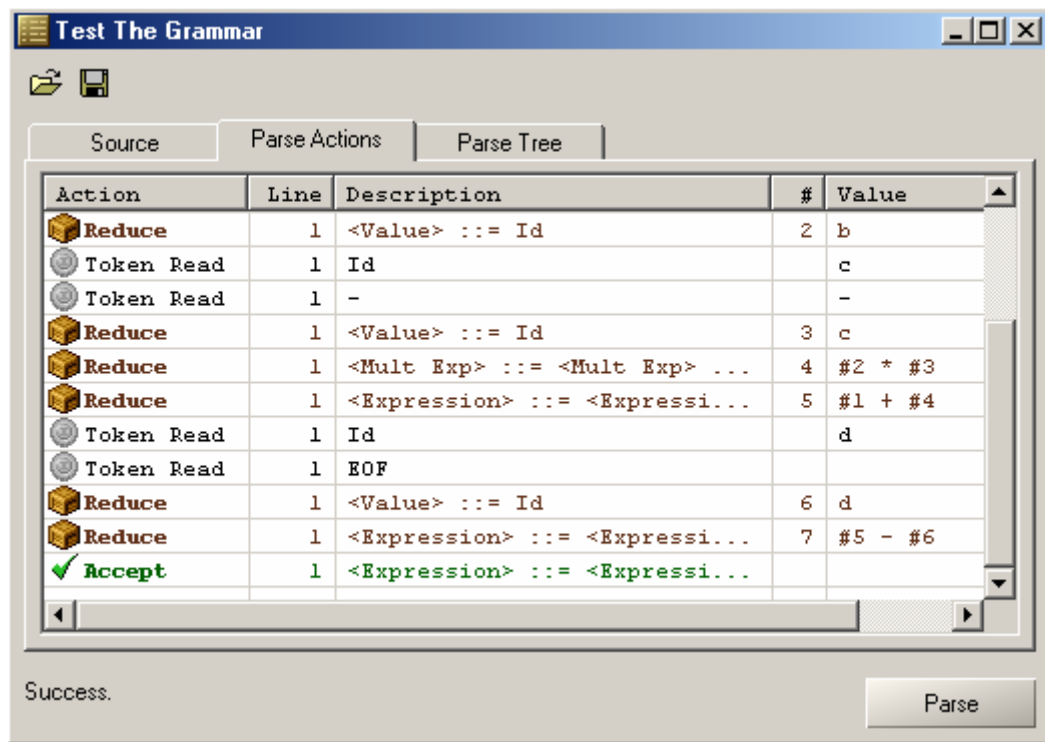


Figure 7-10. Test Window Parse Action List

If the test string is successfully parsed, the GOLD Parser Builder also draws a parse tree for the user to review. This tree can be saved to formatted text.

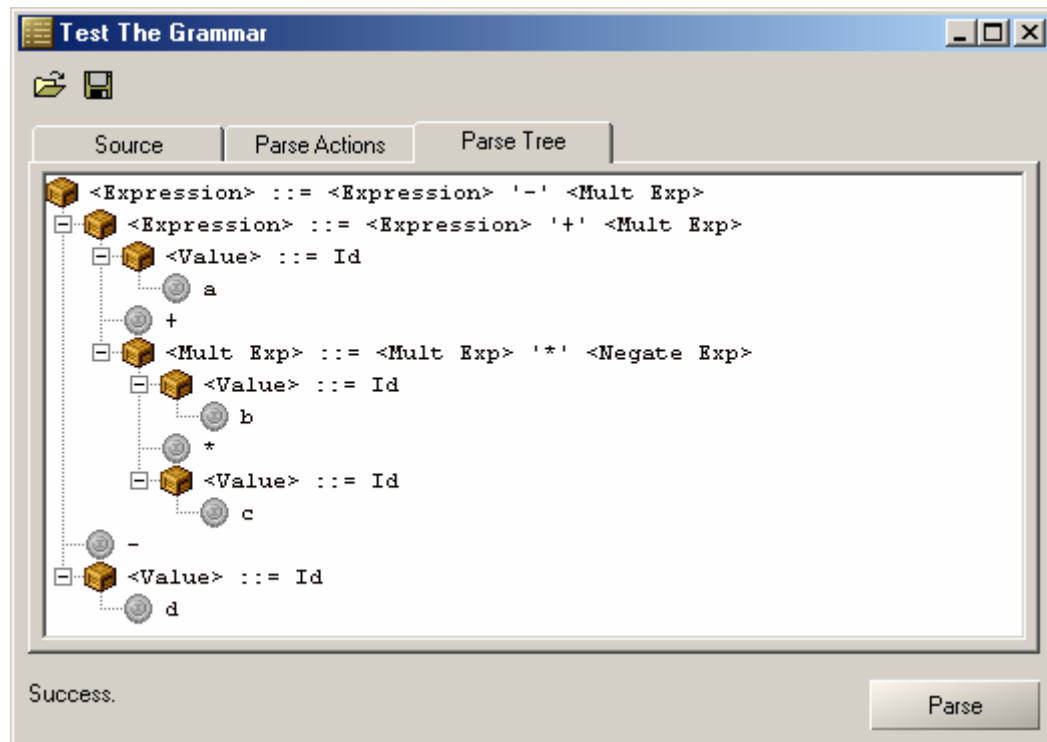
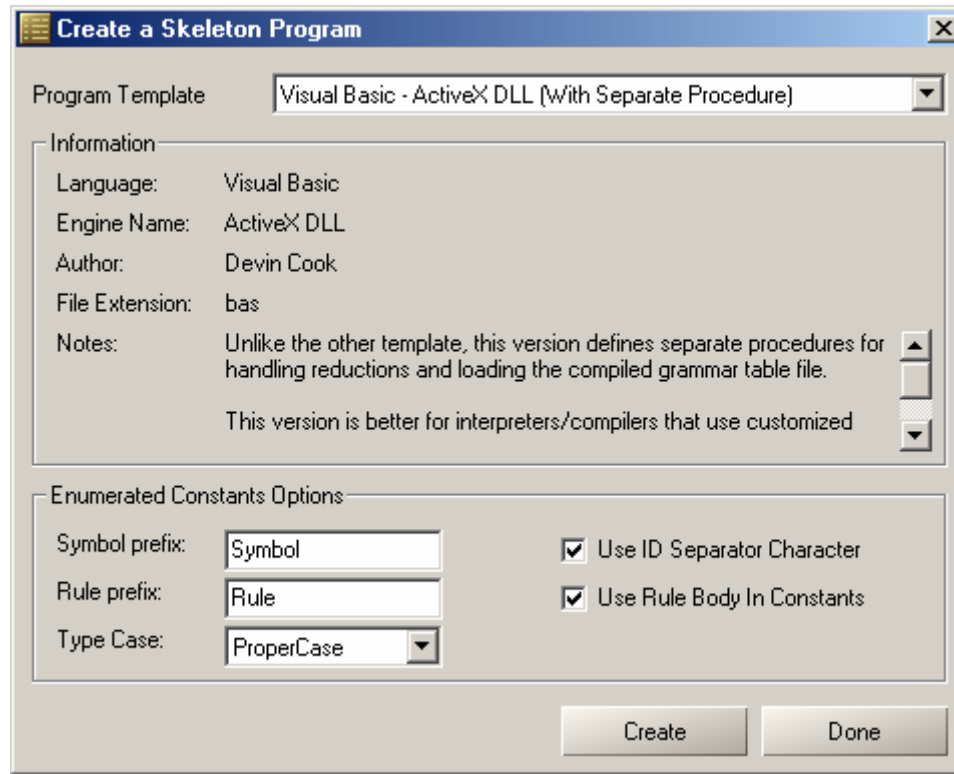


Figure 7-11. Test Window Parse Tree



The image shows a Windows-style dialog box titled "Create a Skeleton Program". It contains a "Program Template" dropdown menu set to "Visual Basic - ActiveX DLL (With Separate Procedure)". Below this is an "Information" section with fields for Language (Visual Basic), Engine Name (ActiveX DLL), Author (Devin Cook), and File Extension (bas). A "Notes" section contains two paragraphs of text. At the bottom is an "Enumerated Constants Options" section with fields for Symbol prefix (Symbol), Rule prefix (Rule), and Type Case (ProperCase), along with two checked checkboxes: "Use ID Separator Character" and "Use Rule Body In Constants". "Create" and "Done" buttons are at the bottom right.

Create a Skeleton Program

Program Template: Visual Basic - ActiveX DLL (With Separate Procedure)

Information

Language: Visual Basic
Engine Name: ActiveX DLL
Author: Devin Cook
File Extension: bas
Notes: Unlike the other template, this version defines separate procedures for handling reductions and loading the compiled grammar table file.
This version is better for interpreters/compilers that use customized

Enumerated Constants Options

Symbol prefix: Symbol ☒ Use ID Separator Character
Rule prefix: Rule ☒ Use Rule Body In Constants
Type Case: ProperCase

Create Done

7.1.10. Exporting the Parse Tables

7.1.10.1. Web Page

The GOLD Parser Builder has the ability to export a grammar's information and computed tables to a web page. This tool was created so that students, professors and developers can display parse information on the Internet.

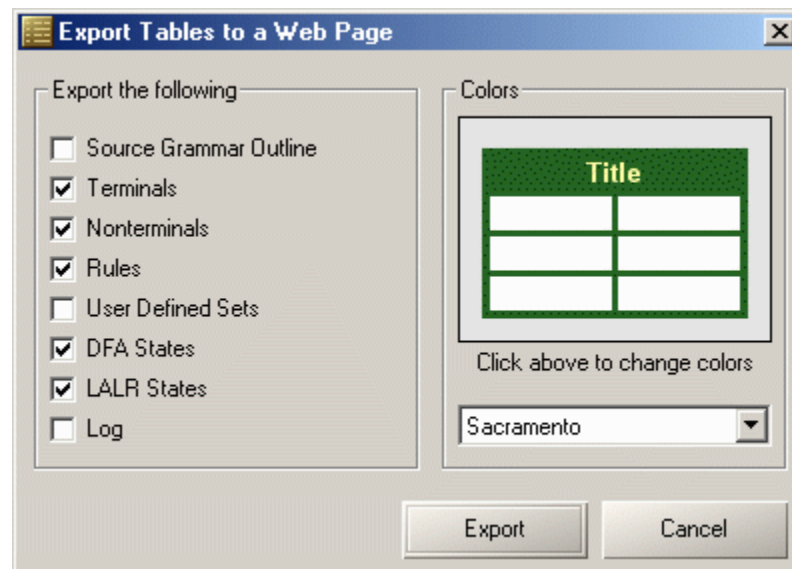
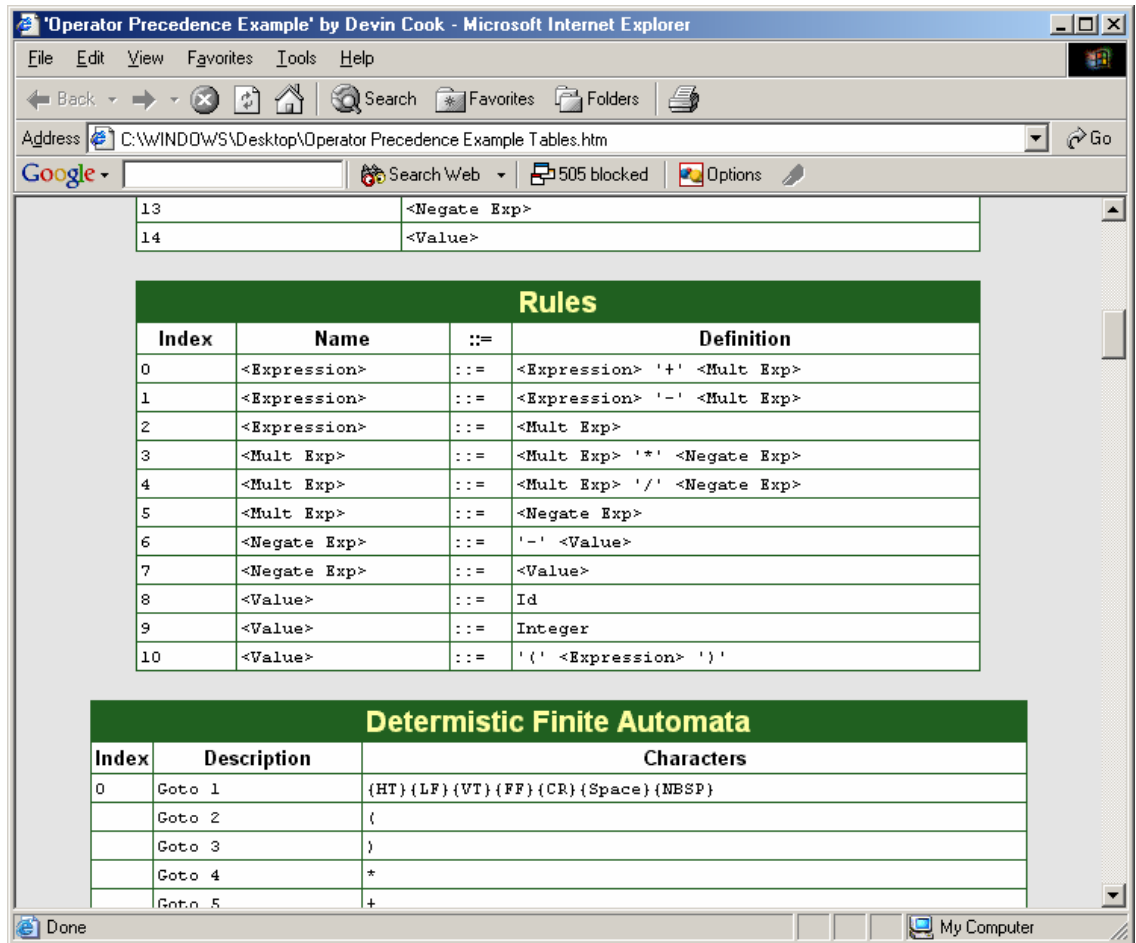


Figure 7-12. Export Tables to a Web page

The web page makes full use of cascading style sheets (W3C, 2003, Cascading Style Sheets), which, in turn, makes changing the design easy. Below is a sample web page created by this tool:



7.1.10.2. Formatted Text

The size of a web page can become quite large for complex grammars. To resolve this issue, the GOLD Parser Builder can also export the parse tables to formatted text.

```

=====
Terminals
=====

Index    Name
-----
0        (EOF)
1        (Error)
2        (Whitespace)

```

```

3      '-'
4      '('
5      ')'
6      '*'
7      '/'
8      '+'
9      Id
10     Integer

```

```

=====
Nonterminals
=====

```

Index	Name
11	<Expression>
12	<Mult Exp>
13	<Negate Exp>
14	<Value>

```

=====
Rules
=====

```

Index	Name	::=	Definition
0	<Expression>	::=	<Expression> '+' <Mult Exp>
1	<Expression>	::=	<Expression> '-' <Mult Exp>
2	<Expression>	::=	<Mult Exp>
3	<Mult Exp>	::=	<Mult Exp> '*' <Negate Exp>
4	<Mult Exp>	::=	<Mult Exp> '/' <Negate Exp>
5	<Mult Exp>	::=	<Negate Exp>
6	<Negate Exp>	::=	'-' <Value>
7	<Negate Exp>	::=	<Value>
8	<Value>	::=	Id
9	<Value>	::=	Integer
10	<Value>	::=	'(' <Expression> ')'

```

=====
DFA States
=====

```

Index	Description	Characters
0	Goto 1	{HT}{LF}{VT}{FF}{CR}{Space}{NBSP}
	Goto 2	(
	Goto 3)
	Goto 4	*

	Goto 5	+
	Goto 6	-
	Goto 7	/
	Goto 8	0123456789
	Goto 9	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
1	Goto 1 Accept (Whitespace)	{HT}{LF}{VT}{FF}{CR}{Space}{NBSP}
2	Accept '('	
3	Accept ')'	
4	Accept '*'	
5	Accept '+'	
6	Accept '-'	
7	Accept '/'	
8	Goto 8 Accept Integer	0123456789
9	Goto 10 Accept Id	0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
10	Goto 10 Accept Id	0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
=====		
LALR States		
=====		
Index	Configuration/Action	Lookahead
-----	-----	-----
0	'-' Shift 1 '(' Shift 2 Id Shift 3 Integer Shift 4 <Expression> Goto 19 <Mult Exp> Goto 17 <Negate Exp> Goto 13 <Value> Goto 10	
1	'(' Shift 2 Id Shift 3	

	Integer Shift 4 <Value> Goto 18
2	'-' Shift 1 '(' Shift 2 Id Shift 3 Integer Shift 4 <Expression> Goto 5 <Mult Exp> Goto 17 <Negate Exp> Goto 13 <Value> Goto 10
3	(EOF) Reduce 8 '-' Reduce 8)' Reduce 8 '*' Reduce 8 '/' Reduce 8 '+' Reduce 8
4	(EOF) Reduce 9 '-' Reduce 9)' Reduce 9 '*' Reduce 9 '/' Reduce 9 '+' Reduce 9
5	'-' Shift 6)' Shift 14 '+' Shift 15
6	'-' Shift 1 '(' Shift 2 Id Shift 3 Integer Shift 4 <Mult Exp> Goto 7 <Negate Exp> Goto 13 <Value> Goto 10
7	'*' Shift 8 '/' Shift 11 (EOF) Reduce 1 '-' Reduce 1)' Reduce 1 '+' Reduce 1
8	'-' Shift 1 '(' Shift 2 Id Shift 3 Integer Shift 4 <Negate Exp> Goto 9 <Value> Goto 10

```
9      (EOF) Reduce 3
      '-' Reduce 3
      ')' Reduce 3
      '*' Reduce 3
      '/' Reduce 3
      '+' Reduce 3

10     (EOF) Reduce 7
      '-' Reduce 7
      ')' Reduce 7
      '*' Reduce 7
      '/' Reduce 7
      '+' Reduce 7

11     '-' Shift 1
      '(' Shift 2
      Id Shift 3
      Integer Shift 4
      <Negate Exp> Goto 12
      <Value> Goto 10

12     (EOF) Reduce 4
      '-' Reduce 4
      ')' Reduce 4
      '*' Reduce 4
      '/' Reduce 4
      '+' Reduce 4

13     (EOF) Reduce 5
      '-' Reduce 5
      ')' Reduce 5
      '*' Reduce 5
      '/' Reduce 5
      '+' Reduce 5

14     (EOF) Reduce 10
      '-' Reduce 10
      ')' Reduce 10
      '*' Reduce 10
      '/' Reduce 10
      '+' Reduce 10

15     '-' Shift 1
      '(' Shift 2
      Id Shift 3
      Integer Shift 4
      <Mult Exp> Goto 16
      <Negate Exp> Goto 13
      <Value> Goto 10
```

```

16      '*' Shift 8
        '/' Shift 11
        (EOF) Reduce 0
        '-' Reduce 0
        ')' Reduce 0
        '+' Reduce 0

17      '*' Shift 8
        '/' Shift 11
        (EOF) Reduce 2
        '-' Reduce 2
        ')' Reduce 2
        '+' Reduce 2

18      (EOF) Reduce 6
        '-' Reduce 6
        ')' Reduce 6
        '*' Reduce 6
        '/' Reduce 6
        '+' Reduce 6

19      (EOF) Accept
        '-' Shift 6
        '+' Shift 15

```

7.1.10.3. XML

The two formats above are quite easy for humans to understand, but are not easy to import into different applications. As a result, the GOLD Parser Builder can also save the content of the tables to XML.

```

<?GOLDParserTables version="1.0"?>
<Tables>
  <Parameters>
    <Parameter Name="Name" Value="Operator Precedence Example"/>
    <Parameter Name="Author" Value="Devin Cook"/>
    <Parameter Name="Version" Value="Example"/>
    <Parameter Name="About" Value="This grammar demonstrates operator
precedence."/>
    <Parameter Name="Case Sensitive" Value="False"/>
    <Parameter Name="Start Symbol" Value="11"/>
  </Parameters>
  <SymbolTable Count="15">
    <Symbol Index="0" Name="EOF" kind="3"/>
    <Symbol Index="1" Name="Error" kind="7"/>
    <Symbol Index="2" Name="Whitespace" kind="2"/>
    <Symbol Index="3" Name="-" kind="1"/>

```

```

<Symbol Index="4" Name="(" kind="1"/>
<Symbol Index="5" Name=")" kind="1"/>
<Symbol Index="6" Name="*" kind="1"/>
<Symbol Index="7" Name="/" kind="1"/>
<Symbol Index="8" Name="+" kind="1"/>
<Symbol Index="9" Name="Id" kind="1"/>
<Symbol Index="10" Name="Integer" kind="1"/>
<Symbol Index="11" Name="Expression" kind="0"/>
<Symbol Index="12" Name="Mult Exp" kind="0"/>
<Symbol Index="13" Name="Negate Exp" kind="0"/>
<Symbol Index="14" Name="Value" kind="0"/>
</SymbolTable>
<RuleTable Count="11">
  <Rule Index="0" NonTerminalIndex="11" SymbolCount="3">
    <RuleSymbol Index="0" SymbolIndex="11"/>
    <RuleSymbol Index="1" SymbolIndex="8"/>
    <RuleSymbol Index="2" SymbolIndex="12"/>
  </Rule>
  <Rule Index="1" NonTerminalIndex="11" SymbolCount="3">
    <RuleSymbol Index="0" SymbolIndex="11"/>
    <RuleSymbol Index="1" SymbolIndex="3"/>
    <RuleSymbol Index="2" SymbolIndex="12"/>
  </Rule>
  <Rule Index="2" NonTerminalIndex="11" SymbolCount="1">
    <RuleSymbol Index="0" SymbolIndex="12"/>
  </Rule>
  <Rule Index="3" NonTerminalIndex="12" SymbolCount="3">
    <RuleSymbol Index="0" SymbolIndex="12"/>
    <RuleSymbol Index="1" SymbolIndex="6"/>
    <RuleSymbol Index="2" SymbolIndex="13"/>
  </Rule>
  <Rule Index="4" NonTerminalIndex="12" SymbolCount="3">
    <RuleSymbol Index="0" SymbolIndex="12"/>
    <RuleSymbol Index="1" SymbolIndex="7"/>
    <RuleSymbol Index="2" SymbolIndex="13"/>
  </Rule>
  <Rule Index="5" NonTerminalIndex="12" SymbolCount="1">
    <RuleSymbol Index="0" SymbolIndex="13"/>
  </Rule>
  <Rule Index="6" NonTerminalIndex="13" SymbolCount="2">
    <RuleSymbol Index="0" SymbolIndex="3"/>
    <RuleSymbol Index="1" SymbolIndex="14"/>
  </Rule>
  <Rule Index="7" NonTerminalIndex="13" SymbolCount="1">
    <RuleSymbol Index="0" SymbolIndex="14"/>
  </Rule>
  <Rule Index="8" NonTerminalIndex="14" SymbolCount="1">
    <RuleSymbol Index="0" SymbolIndex="9"/>
  </Rule>
  <Rule Index="9" NonTerminalIndex="14" SymbolCount="1">
    <RuleSymbol Index="0" SymbolIndex="10"/>
  </Rule>
  <Rule Index="10" NonTerminalIndex="14" SymbolCount="3">
    <RuleSymbol Index="0" SymbolIndex="4"/>
    <RuleSymbol Index="1" SymbolIndex="11"/>
    <RuleSymbol Index="2" SymbolIndex="5"/>
  </Rule>

```

```

</RuleTable>
<CharSetTable Count="10">
  <CharSet Index="0" Count="7">
    <Char UnicodeIndex="9"/>
    <Char UnicodeIndex="10"/>
    <Char UnicodeIndex="11"/>
    <Char UnicodeIndex="12"/>
    <Char UnicodeIndex="13"/>
    <Char UnicodeIndex="32"/>
    <Char UnicodeIndex="160"/>
  </CharSet>
  <CharSet Index="1" Count="1">
    <Char UnicodeIndex="40"/>
  </CharSet>
  <CharSet Index="2" Count="1">
    <Char UnicodeIndex="41"/>
  </CharSet>
  <CharSet Index="3" Count="1">
    <Char UnicodeIndex="42"/>
  </CharSet>
  <CharSet Index="4" Count="1">
    <Char UnicodeIndex="43"/>
  </CharSet>
  <CharSet Index="5" Count="1">
    <Char UnicodeIndex="45"/>
  </CharSet>
  <CharSet Index="6" Count="1">
    <Char UnicodeIndex="47"/>
  </CharSet>
  <CharSet Index="7" Count="10">
    <Char UnicodeIndex="48"/>
    <Char UnicodeIndex="49"/>
    <Char UnicodeIndex="50"/>
    <Char UnicodeIndex="51"/>
    <Char UnicodeIndex="52"/>
    <Char UnicodeIndex="53"/>
    <Char UnicodeIndex="54"/>
    <Char UnicodeIndex="55"/>
    <Char UnicodeIndex="56"/>
    <Char UnicodeIndex="57"/>
  </CharSet>
  <CharSet Index="8" Count="52">
    <Char UnicodeIndex="65"/>
    <Char UnicodeIndex="66"/>
    <Char UnicodeIndex="67"/>
    <Char UnicodeIndex="68"/>
    <Char UnicodeIndex="69"/>
    <Char UnicodeIndex="70"/>
    <Char UnicodeIndex="71"/>
    <Char UnicodeIndex="72"/>
    <Char UnicodeIndex="73"/>
    <Char UnicodeIndex="74"/>
    <Char UnicodeIndex="75"/>
    <Char UnicodeIndex="76"/>
    <Char UnicodeIndex="77"/>
    <Char UnicodeIndex="78"/>
    <Char UnicodeIndex="79"/>
  </CharSet>

```

```

<Char UnicodeIndex="80"/>
<Char UnicodeIndex="81"/>
<Char UnicodeIndex="82"/>
<Char UnicodeIndex="83"/>
<Char UnicodeIndex="84"/>
<Char UnicodeIndex="85"/>
<Char UnicodeIndex="86"/>
<Char UnicodeIndex="87"/>
<Char UnicodeIndex="88"/>
<Char UnicodeIndex="89"/>
<Char UnicodeIndex="90"/>
<Char UnicodeIndex="97"/>
<Char UnicodeIndex="98"/>
<Char UnicodeIndex="99"/>
<Char UnicodeIndex="100"/>
<Char UnicodeIndex="101"/>
<Char UnicodeIndex="102"/>
<Char UnicodeIndex="103"/>
<Char UnicodeIndex="104"/>
<Char UnicodeIndex="105"/>
<Char UnicodeIndex="106"/>
<Char UnicodeIndex="107"/>
<Char UnicodeIndex="108"/>
<Char UnicodeIndex="109"/>
<Char UnicodeIndex="110"/>
<Char UnicodeIndex="111"/>
<Char UnicodeIndex="112"/>
<Char UnicodeIndex="113"/>
<Char UnicodeIndex="114"/>
<Char UnicodeIndex="115"/>
<Char UnicodeIndex="116"/>
<Char UnicodeIndex="117"/>
<Char UnicodeIndex="118"/>
<Char UnicodeIndex="119"/>
<Char UnicodeIndex="120"/>
<Char UnicodeIndex="121"/>
<Char UnicodeIndex="122"/>
</CharSet>
<CharSet Index="9" Count="62">
  <Char UnicodeIndex="48"/>
  <Char UnicodeIndex="49"/>
  <Char UnicodeIndex="50"/>
  <Char UnicodeIndex="51"/>
  <Char UnicodeIndex="52"/>
  <Char UnicodeIndex="53"/>
  <Char UnicodeIndex="54"/>
  <Char UnicodeIndex="55"/>
  <Char UnicodeIndex="56"/>
  <Char UnicodeIndex="57"/>
  <Char UnicodeIndex="65"/>
  <Char UnicodeIndex="66"/>
  <Char UnicodeIndex="67"/>
  <Char UnicodeIndex="68"/>
  <Char UnicodeIndex="69"/>
  <Char UnicodeIndex="70"/>
  <Char UnicodeIndex="71"/>
  <Char UnicodeIndex="72"/>

```

```

    <Char UnicodeIndex="73"/>
    <Char UnicodeIndex="74"/>
    <Char UnicodeIndex="75"/>
    <Char UnicodeIndex="76"/>
    <Char UnicodeIndex="77"/>
    <Char UnicodeIndex="78"/>
    <Char UnicodeIndex="79"/>
    <Char UnicodeIndex="80"/>
    <Char UnicodeIndex="81"/>
    <Char UnicodeIndex="82"/>
    <Char UnicodeIndex="83"/>
    <Char UnicodeIndex="84"/>
    <Char UnicodeIndex="85"/>
    <Char UnicodeIndex="86"/>
    <Char UnicodeIndex="87"/>
    <Char UnicodeIndex="88"/>
    <Char UnicodeIndex="89"/>
    <Char UnicodeIndex="90"/>
    <Char UnicodeIndex="97"/>
    <Char UnicodeIndex="98"/>
    <Char UnicodeIndex="99"/>
    <Char UnicodeIndex="100"/>
    <Char UnicodeIndex="101"/>
    <Char UnicodeIndex="102"/>
    <Char UnicodeIndex="103"/>
    <Char UnicodeIndex="104"/>
    <Char UnicodeIndex="105"/>
    <Char UnicodeIndex="106"/>
    <Char UnicodeIndex="107"/>
    <Char UnicodeIndex="108"/>
    <Char UnicodeIndex="109"/>
    <Char UnicodeIndex="110"/>
    <Char UnicodeIndex="111"/>
    <Char UnicodeIndex="112"/>
    <Char UnicodeIndex="113"/>
    <Char UnicodeIndex="114"/>
    <Char UnicodeIndex="115"/>
    <Char UnicodeIndex="116"/>
    <Char UnicodeIndex="117"/>
    <Char UnicodeIndex="118"/>
    <Char UnicodeIndex="119"/>
    <Char UnicodeIndex="120"/>
    <Char UnicodeIndex="121"/>
    <Char UnicodeIndex="122"/>
  </CharSet>
</CharSetTable>
<DFA Count="11" InitialState="0">
  <DFAState Index="0" EdgeCount="9" AcceptSymbol="-1">
    <DFAEdge CharSetIndex="0" Target="1"/>
    <DFAEdge CharSetIndex="1" Target="2"/>
    <DFAEdge CharSetIndex="2" Target="3"/>
    <DFAEdge CharSetIndex="3" Target="4"/>
    <DFAEdge CharSetIndex="4" Target="5"/>
    <DFAEdge CharSetIndex="5" Target="6"/>
    <DFAEdge CharSetIndex="6" Target="7"/>
    <DFAEdge CharSetIndex="7" Target="8"/>
    <DFAEdge CharSetIndex="8" Target="9"/>
  </DFAState>
</DFA>

```

```

</DFAState>
<DFAState Index="1" EdgeCount="1" AcceptSymbol="2">
  <DFAEdge CharSetIndex="0" Target="1"/>
</DFAState>
<DFAState Index="2" EdgeCount="0" AcceptSymbol="4">
</DFAState>
<DFAState Index="3" EdgeCount="0" AcceptSymbol="5">
</DFAState>
<DFAState Index="4" EdgeCount="0" AcceptSymbol="6">
</DFAState>
<DFAState Index="5" EdgeCount="0" AcceptSymbol="8">
</DFAState>
<DFAState Index="6" EdgeCount="0" AcceptSymbol="3">
</DFAState>
<DFAState Index="7" EdgeCount="0" AcceptSymbol="7">
</DFAState>
<DFAState Index="8" EdgeCount="1" AcceptSymbol="10">
  <DFAEdge CharSetIndex="7" Target="8"/>
</DFAState>
<DFAState Index="9" EdgeCount="1" AcceptSymbol="9">
  <DFAEdge CharSetIndex="9" Target="10"/>
</DFAState>
<DFAState Index="10" EdgeCount="1" AcceptSymbol="9">
  <DFAEdge CharSetIndex="9" Target="10"/>
</DFAState>
</DFA>
<LALR Count="20" InitialState="0">
  <LALRState Index="0" ActionCount="8">
    <LALRAction SymbolIndex="3" Action="1" Value="1"/>
    <LALRAction SymbolIndex="4" Action="1" Value="2"/>
    <LALRAction SymbolIndex="9" Action="1" Value="3"/>
    <LALRAction SymbolIndex="10" Action="1" Value="4"/>
    <LALRAction SymbolIndex="11" Action="3" Value="19"/>
    <LALRAction SymbolIndex="12" Action="3" Value="17"/>
    <LALRAction SymbolIndex="13" Action="3" Value="13"/>
    <LALRAction SymbolIndex="14" Action="3" Value="10"/>
  </LALRState>
  <LALRState Index="1" ActionCount="4">
    <LALRAction SymbolIndex="4" Action="1" Value="2"/>
    <LALRAction SymbolIndex="9" Action="1" Value="3"/>
    <LALRAction SymbolIndex="10" Action="1" Value="4"/>
    <LALRAction SymbolIndex="14" Action="3" Value="18"/>
  </LALRState>
  <LALRState Index="2" ActionCount="8">
    <LALRAction SymbolIndex="3" Action="1" Value="1"/>
    <LALRAction SymbolIndex="4" Action="1" Value="2"/>
    <LALRAction SymbolIndex="9" Action="1" Value="3"/>
    <LALRAction SymbolIndex="10" Action="1" Value="4"/>
    <LALRAction SymbolIndex="11" Action="3" Value="5"/>
    <LALRAction SymbolIndex="12" Action="3" Value="17"/>
    <LALRAction SymbolIndex="13" Action="3" Value="13"/>
    <LALRAction SymbolIndex="14" Action="3" Value="10"/>
  </LALRState>
  <LALRState Index="3" ActionCount="6">
    <LALRAction SymbolIndex="0" Action="2" Value="8"/>
    <LALRAction SymbolIndex="3" Action="2" Value="8"/>
    <LALRAction SymbolIndex="5" Action="2" Value="8"/>

```



```

        <LALRAction SymbolIndex="6" Action="2" Value="8"/>
        <LALRAction SymbolIndex="7" Action="2" Value="8"/>
        <LALRAction SymbolIndex="8" Action="2" Value="8"/>
    </LALRState>
    <LALRState Index="4" ActionCount="6">
        <LALRAction SymbolIndex="0" Action="2" Value="9"/>
        <LALRAction SymbolIndex="3" Action="2" Value="9"/>
        <LALRAction SymbolIndex="5" Action="2" Value="9"/>
        <LALRAction SymbolIndex="6" Action="2" Value="9"/>
        <LALRAction SymbolIndex="7" Action="2" Value="9"/>
        <LALRAction SymbolIndex="8" Action="2" Value="9"/>
    </LALRState>
    <LALRState Index="5" ActionCount="3">
        <LALRAction SymbolIndex="3" Action="1" Value="6"/>
        <LALRAction SymbolIndex="5" Action="1" Value="14"/>
        <LALRAction SymbolIndex="8" Action="1" Value="15"/>
    </LALRState>
    <LALRState Index="6" ActionCount="7">
        <LALRAction SymbolIndex="3" Action="1" Value="1"/>
        <LALRAction SymbolIndex="4" Action="1" Value="2"/>
        <LALRAction SymbolIndex="9" Action="1" Value="3"/>
        <LALRAction SymbolIndex="10" Action="1" Value="4"/>
        <LALRAction SymbolIndex="12" Action="3" Value="7"/>
        <LALRAction SymbolIndex="13" Action="3" Value="13"/>
        <LALRAction SymbolIndex="14" Action="3" Value="10"/>
    </LALRState>
    <LALRState Index="7" ActionCount="6">
        <LALRAction SymbolIndex="6" Action="1" Value="8"/>
        <LALRAction SymbolIndex="7" Action="1" Value="11"/>
        <LALRAction SymbolIndex="0" Action="2" Value="1"/>
        <LALRAction SymbolIndex="3" Action="2" Value="1"/>
        <LALRAction SymbolIndex="5" Action="2" Value="1"/>
        <LALRAction SymbolIndex="8" Action="2" Value="1"/>
    </LALRState>
    <LALRState Index="8" ActionCount="6">
        <LALRAction SymbolIndex="3" Action="1" Value="1"/>
        <LALRAction SymbolIndex="4" Action="1" Value="2"/>
        <LALRAction SymbolIndex="9" Action="1" Value="3"/>
        <LALRAction SymbolIndex="10" Action="1" Value="4"/>
        <LALRAction SymbolIndex="13" Action="3" Value="9"/>
        <LALRAction SymbolIndex="14" Action="3" Value="10"/>
    </LALRState>
    <LALRState Index="9" ActionCount="6">
        <LALRAction SymbolIndex="0" Action="2" Value="3"/>
        <LALRAction SymbolIndex="3" Action="2" Value="3"/>
        <LALRAction SymbolIndex="5" Action="2" Value="3"/>
        <LALRAction SymbolIndex="6" Action="2" Value="3"/>
        <LALRAction SymbolIndex="7" Action="2" Value="3"/>
        <LALRAction SymbolIndex="8" Action="2" Value="3"/>
    </LALRState>
    <LALRState Index="10" ActionCount="6">
        <LALRAction SymbolIndex="0" Action="2" Value="7"/>
        <LALRAction SymbolIndex="3" Action="2" Value="7"/>
        <LALRAction SymbolIndex="5" Action="2" Value="7"/>
        <LALRAction SymbolIndex="6" Action="2" Value="7"/>
        <LALRAction SymbolIndex="7" Action="2" Value="7"/>
        <LALRAction SymbolIndex="8" Action="2" Value="7"/>

```



```

</LALRState>
<LALRState Index="18" ActionCount="6">
  <LALRAction SymbolIndex="0" Action="2" Value="6"/>
  <LALRAction SymbolIndex="3" Action="2" Value="6"/>
  <LALRAction SymbolIndex="5" Action="2" Value="6"/>
  <LALRAction SymbolIndex="6" Action="2" Value="6"/>
  <LALRAction SymbolIndex="7" Action="2" Value="6"/>
  <LALRAction SymbolIndex="8" Action="2" Value="6"/>
</LALRState>
<LALRState Index="19" ActionCount="3">
  <LALRAction SymbolIndex="0" Action="4" Value="0"/>
  <LALRAction SymbolIndex="3" Action="1" Value="6"/>
  <LALRAction SymbolIndex="8" Action="1" Value="15"/>
</LALRState>
</LALR>
</Tables>

```

7.2. Website

7.2.1. Introduction

A website was developed for the GOLD Parser Builder so that the application could be made readily available and that a site could be used to share new implementations of the GOLD Engine. This also provides a place to display the latest news about the Builder – including known bugs and new releases.

In addition, the website is designed to serve as a knowledge base for the system. Most of the information found in this document is also located on the website. This includes the theoretical framework, Builder design, Engine design, Compiled Grammar Table design, etc...



Figure 7-13. GOLD Parser Website.

7.2.2. Contributors Section

One of the key features of the website is a section for contributors. Anyone who is kind and generous enough to contribute to the site, deserves recognition. This includes the authors of different implementations of the Engine as well as those who contribute grammars and tools. The section also helps developers contact the correct contributor if they have questions or comments.



Figure 7-14. Contributors Page

7.2.3. Online Documentation

The website also contains an online version of the documentation for Builder, Compiled Grammar Table format, Program Templates and the different implementations of the Engine (if the developer submits documentation).

The GOLD Parser Builder normally opens the online documentation when it is selected from the Help menu. Since the documentation changes from time to time, this makes sure that the latest version is always available to the developer. This also includes known bugs and possible workarounds.

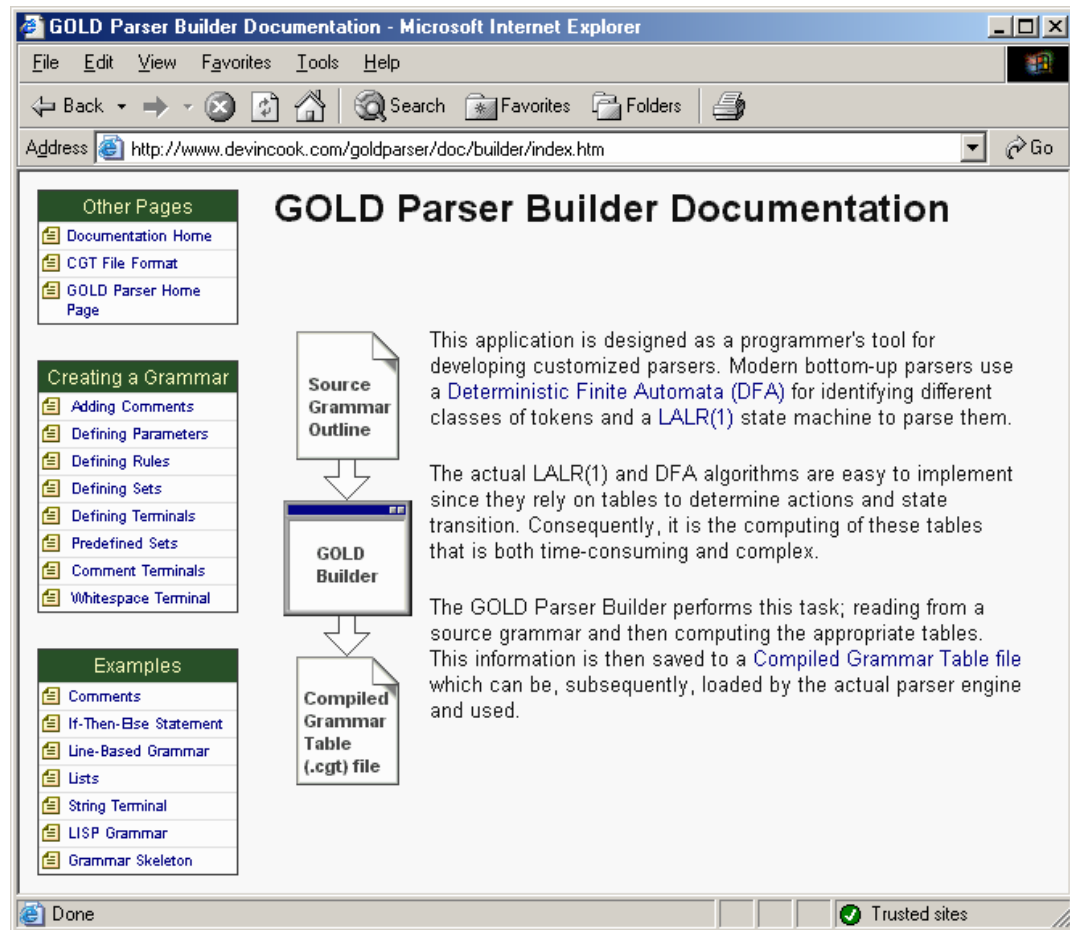


Figure 7-15. Online Documentation

7.2.4. Check for Updates

New versions of the GOLD Builder will be created over time. With each update, bugs are fixed and new features are made available. For those using the application, updates made provide a fix for a bug plaguing the system and may greatly aid the design of their programming language.

The GOLD Builder contains a menu item that loads a web page on the GOLD Parser Website. This page is specific to the version of the application they are using. Essentially the name of the HTML document contains the major version, minor version and revision values.

If a new version of the Builder is available, the web page will alert the developer to this fact and will contain any additional information that is relevant. The following screenshot demonstrates that a new version is available.

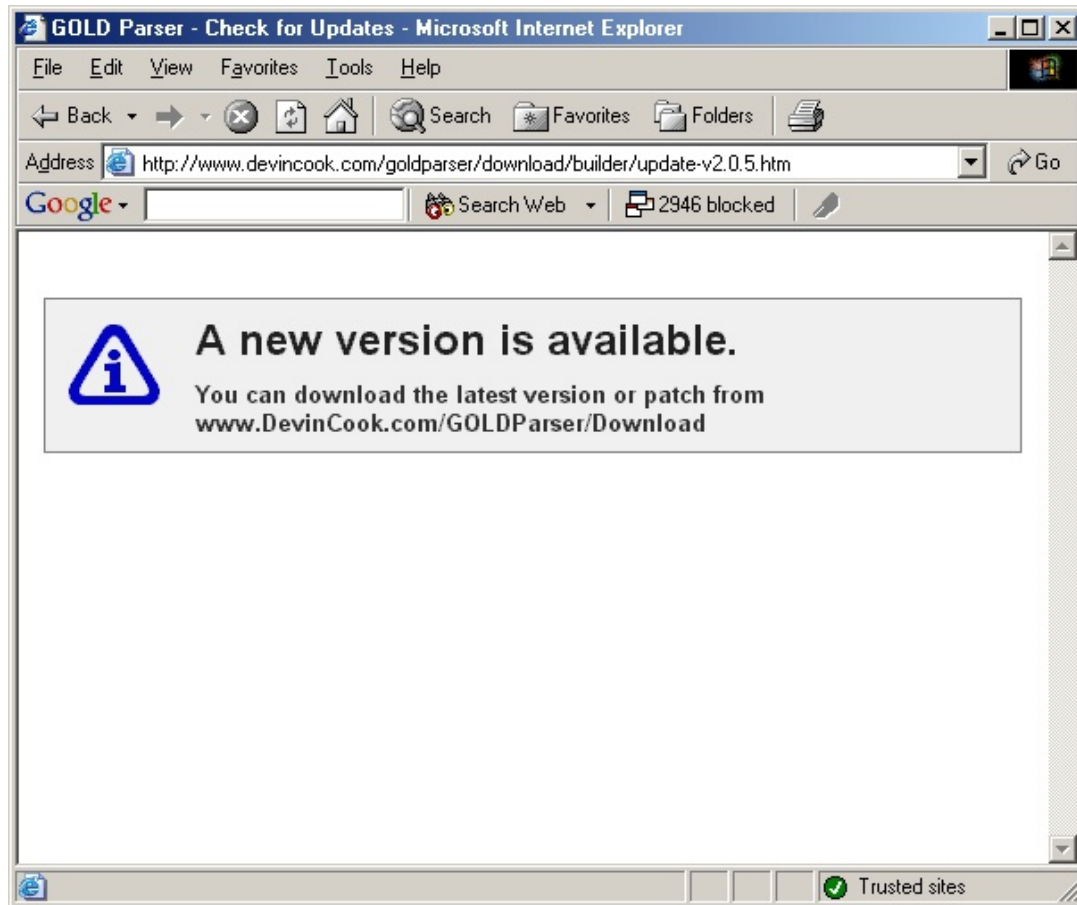


Figure 7-16. Check for Updates - Update Available

If the developer is using the most current version of the Builder, the screen will display the appropriate message.

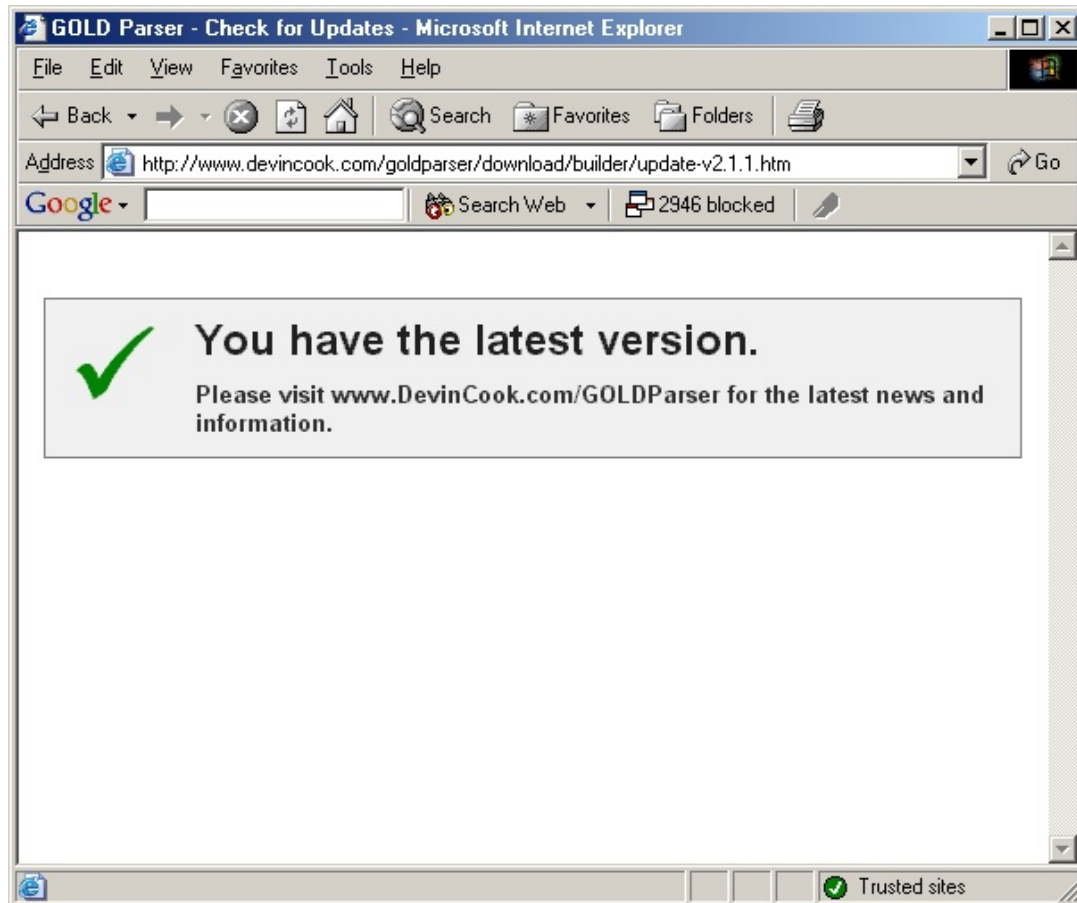


Figure 7-17. Check for Updates - Latest Version

8. Comparison

8.1. Introduction

This chapter contains a short comparison between the GOLD, YACC / Lex and ANTLR meta-languages. One of the most common attributes found in programming languages is the ability to define mathematic expressions. In the following section, a grammar is defined for mathematical expressions using these three different meta-languages.

The syntax between the GOLD Meta-Language and the YACC Meta-Language is almost identical. This is due to the fact that both GOLD and YACC use the LALR parsing algorithm and use Backus-Naur Form rules.

However, since YACC is a compiler-compiler, the system is able to use special notation to shorten the number of rules defined in the grammar. In particular, developers can use the %left, %right and %nonassoc tags to declare operators. YACC assigns precedence level based on the order in which the tags are declared. The earlier the tag is declared, the lower the operator precedence. The %prec tag allows the developer to explicitly declare precedence. To handle this additional logic, the system creates a number of hidden rules. The second example YACC grammar uses this approach.

The left-recursion that is used in the YACC and GOLD examples cannot be used for the ANTLR parser. This is due to the fact that ANTLR uses the LL parsing algorithm which cannot handle left-

recursion. However, since ANTLR contains a large number of classes as part of the meta-language (and the resulting C#, Java or C++ program), the rules can be shortened using Extended Backus-Naur Form. As a result, the following ANTLR grammar contains fewer rules than GOLD or YACC. The definition for the whitespace terminal must be manually implemented by the developer. In the ANTLR version, the definition was copied verbatim from examples on the ANTLR website (www.antlr.org).

8.2. Mathematical Expressions

8.2.1. GOLD Meta-Language

```

"Name"      = 'Operator Precedence Example'
"Author"    = 'Devin Cook'
"About"     = 'This grammar defines simple mathematical expressions'

"Case Sensitive" = False
"Start Symbol"  = <Expression>

! -----
! Terminals
! -----

{ID Tail} = {AlphaNumeric} + [_]

ID        = {Letter}{ID Tail}*
Number    = {Digit}+ ( '.' {Digit}+ )?

! -----
! Rules
! -----

<Expression> ::= <Expression> '+' <Mult Exp>
               | <Expression> '-' <Mult Exp>
               | <Mult Exp>

<Mult Exp>   ::= <Mult Exp> '*' <Negate Exp>
               | <Mult Exp> '/' <Negate Exp>
               | <Negate Exp>

<Negate Exp> ::= '-' <Value> | <Value>

<Value>      ::= ID | Number | '(' <Expression> ')'
```

8.2.2. YACC / Lex Meta-Language

8.2.2.1. Lex File

```

/*
-----
Terminals
-----
*/

digit      [0-9]
letter     [A-Za-z]
id_tail    [0-9A-Za-z\_ ]

%%

{letter}{id_tail}*      { return (ID);      }
{digit}+(\.{digit}*)?  { return (NUMBER); }
"+"                    { return (PLUS);    }
"- "                   { return (MINUS);    }
"* "                   { return (TIMES);    }
"/ "                   { return (DIVIDE);   }
" ("                   { return (LPARAN);   }
") "                   { return (RPARAN);   }
\n                     { } /* Ignore newline */
%%

```

8.2.2.2. YACC File - Approach #1

```
%start expression

%token ID NUMBER LPARAN RPARAN
%token PLUS MINUS
%token TIMES DIVIDE

%%
/*
-----
Rules
-----
*/

expression : expression PLUS mult_exp
           | expression MINUS mult_exp
           | mult_exp
           ;

mult_exp   : mult_exp TIMES negate_exp
           | mult_exp DIVIDE negate_exp
           | negate_exp
           ;

negate_exp : MINUS value
           | value
           ;

value      : ID
           | NUMBER
           | LPARAN expression RPARAN
           ;
```


8.2.2.3. YACC File - Approach #2

```
%start expression

%token ID NUMBER LPARAN RPARAN
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UNARY_MINUS

%%
/*
-----
Rules - This example uses hidden rules
-----
*/

expression : expression PLUS expression
           | expression MINUS expression
           | expression TIMES expression
           | expression DIVIDE expression
           | MINUS expression %prec UNARY_MINUS
           | ID
           | NUMBER
           | LPARAN expression RPARAN
           ;
```

8.2.3. ANTLR Meta-Language

```
// -----
// Terminals
// -----

class TestLexer extends Lexer;

tokens {
    PLUS      = "+" ;
    MINUS     = "-" ;
    TIMES     = "*" ;
    DIVIDE    = "/" ;
    LPARAN    = "(" ;
    RPARAN    = ")" ;
}

// Whitespace
WS : ( ' '
      | '\t'
      | '\f'
      // handle newlines
      | ( "\r\n" // PC
        | '\r'   // Macintosh
        | '\n'   // Unix
        )
      { newline(); }
      )
  { _ttype = Token.SKIP; }
;

ID : ( 'a'..'z' | 'A'..'Z' )
    ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' )* ;

NUMBER : ('0'..'9')+ ( '.' ('0'..'9')+ )? ;

// -----
// Rules
// -----

class TestParser extends Parser;

expression : mult_exp ( ( PLUS | MINUS ) expression )* ;

mult_exp  : negate_exp ( ( TIMES | DIVIDE ) mult_exp )* ;
```

```
negate_exp : ( MINUS )? value
```

```
value      : ID | NUMBER | LPARAN expression RPARAN
```

9. Conclusion

9.1. Results

9.1.1. Multiple Programming Language Support

The Visual Basic source code for the Engine was made available on the GOLD Parser Website in the early stages of the project. It was hoped that other interested computer scientists would convert this source into other programming languages.

The results were better than expected. Over time, a number of talented computer scientists submitted source code for new implementations of the Engine. To create a new implementation, the developers simply needed to implement the DFA and LALR algorithms as well as the code necessary to read the Compiled Grammar Table file produced by the Builder. The Visual Basic 6 source code was provided on the Internet so it could be used as a guide for other programming languages. Naturally, the objects and varying attributes of each language were used for each new implementation.

Currently, the Engine is available in: Visual Basic .NET, Visual Basic 6 (original source), C#, C++, Visual C++, ANSI C, Delphi 5 and Java. Listed below are those developers who were kind and generous enough to share their work. Each name is listed in the order in which source code was received.

9.1.1.1. Sun Java Programming Language

Matthew Hawkins, a master's student in Great Britain, translated the source code to Sun Microsystems' popular Java Programming Language. This implementation of the Engine allows developers to write interpreters, translators and compilers that will work on a myriad of platforms. He was the first contributor to the site.

9.1.1.2. Microsoft C# Programming Language

Marcus Klimstra implemented the Engine on Microsoft's new .NET® Platform. Developers using C#, Visual Basic .NET, J# or any of the .NET compliant programming languages can use this component to develop interpreters, translators and compilers. The source code was written in C#.

Robert van Loenhout submitted a different implementation of the Engine for the Microsoft .NET architecture. This version was written in Microsoft C#®. He also supplied full documentation for using the DLL.

9.1.1.3. Microsoft Visual C++

Manuel Astudillo implemented a C++ Engine from the ground-up. It contains an additional class named "ASTCreator" which is designed to aid in the construction of a specialized parse tree. He also submitted sample projects for Visual C++ 6 and Visual C++ 7.

Eylem Ugurel created another implementation of the GOLD Parser Engine in C++. He submitted a file that contains both the Visual C++ source code and the source code for his "Boethos" scripting engine.

9.1.1.4. Borland Delphi Programming Language

Martin van der Geer ported the Visual Basic 5 source to the popular Delphi® Programming Language. Before this source was written, there was no free or commercially available parser for the Delphi Programming Language. As result, the GOLD Parser Website was quickly listed in a number of Delphi-related forums and web pages.

Ibrahim Khachab later submitted a modified version of the Delphi source code written by Martin van der Geer. All modified sections were marked with `//IBRAHIM`.

Alexandre Rai, implemented the Engine for both Delphi 3 & 5. In this version, most of the functions and classes have been reimplemented and the code is designed to be easier to use.

9.1.1.5. Microsoft Visual Basic .NET

Reggie Wilbanks ported the GOLD Parser Engine source code to the new Visual Basic .NET programming language. He has also submitted an example project which draws a parse tree for a given input stream.

9.1.2. Popularity

On average, approximately 3000 copies of the Builder application are downloaded per month from the GOLD Parser website. This value seems to be heavily affected by the time of year in correspondence with normal academic years.

9.2. Future work

9.2.1. Support Additional Platforms

Currently, the Builder is only available on the Windows 32-bit platforms. Although this makes it accessible to a wide number of students and computer scientists, different versions are needed for Linux, UNIX and a command-line version for Windows. The source code for the Builder is written in Visual Basic 6, and will have to be translated to C++ for this process.

9.2.2. Support "Virtual" terminals

Additional enhancements to the grammar syntax and semantics are needed to handle grammars which are not fully context free. Languages such as Python, do not use symbols to mark the start and end of a block of statements. Instead, the indentation of each statement is used to determine where a block begins and ends.

In this example, content of whitespace is important – or at least the content of a token rather than solely its classification by the tokenizer. If a program has an indent of 10 spaces, the grammar must contain a set of rules for statements at this level. The same is true for all other levels of indentation - requiring an infinite number of rules to parse.

One possible solution would be to add "virtual" terminals to the grammar syntax. These terminals would be entered into the symbol table, but will not be added to the Deterministic Finite Automata. The

developer or specialized implementations of the Engine would create these tokens at runtime and push them onto the token queue. In the case of Python, tokens could be created to represent a increase or decrease in indentation. Of course, the developer could create special tokens for any number of reasons; not just this one alone.

The notation for "virtual" must be easy to read by the developer, since this is not part of standard Backus-Naur Form or regular expression notation. However, for backward compatibility and to allow a consistent development format, the keywords 'virtual', 'empty' or 'null' cannot be used. "Virtual" terminals can possibly be listed in an additional parameter or the notation of terminals can be modified to allow the developer to denote them as such.

Appendix A. Engine Code Listing

A.1. Introduction

The following code declares the Visual Basic 6 implementation of the Engine that was written in conjunction with this project. Although this language supports most modern aspects of object-oriented programming, the file access system still dates back to the language's predecessors. In particular, file access is performed through the use of BASIC commands.

To allow the code to be easily converted to other programming languages, attempts were made to hide Visual Basic 6's file access from the rest of the code. This was accomplished through a "stream" wrapper class.

A.2. Classes

A.2.1. CompiledGrammarTableFile

This class is used to read information stored in the very simple file structure used by the Compiled Grammar Table file.

```

=====
' Class Name:
'     CompiledGrammarTableFile
'
' Instanting:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     This class is used to read information stored in the very simple
file
'     structure used by the Compiled Grammar Table file.
'
'     Record Structure
'         | 1 byte      | 2 byte      | Variable size
'         | Character 'M' | Entry Count | Entries (0 to 32k)
'
'     For more information, please consult
'     http://www.DevinCook.com/GOLDParser/doc/cgt/
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     (None)
'
=====

Option Explicit
Option Compare Text

Private EntryList()           As Variant
Private EntryListCount       As Integer
Private EntryReadPosition    As Integer

```

```

Private Const RecordContentMulti = 77 'M

Private Enum EntryContentConstants
    EntryContentEmpty = 69 'E
    EntryContentInteger = 73 'I - Signed, 2 byte
    EntryContentString = 83 'S - Unicode format
    EntryContentBoolean = 66 'B - 1 Byte, Value is 0 or 1
    EntryContentByte = 98 'b
End Enum

Private pHeader As String
Private iostream As New LookaheadStream

Public Function RetrieveDone() As Boolean

    RetrieveDone = Not (EntryReadPosition < EntryListCount)

End Function

Public Function RetrieveNext() As Variant

    If Not RetrieveDone() Then
        RetrieveNext = EntryList(EntryReadPosition)
        EntryReadPosition = EntryReadPosition + 1
    Else
        RetrieveNext = Empty
    End If

End Function

Public Sub Store(Value As Variant)
    EntryListCount = EntryListCount + 1

    ReDim Preserve EntryList(0 To EntryListCount - 1) 'Change the
size of the array
    EntryList(EntryListCount - 1) = Value
End Sub

Public Sub Clear()

    EntryListCount = 0
    EntryReadPosition = 0
    Erase EntryList

End Sub

Public Sub CloseFile()

    If iostream.FileOpen Then

```

```

        iostream.CloseFile
    End If

End Sub

Public Function EntryCount() As Integer

    EntryCount = EntryListCount

End Function

Public Function Done() As Boolean

    Done = iostream.Done

    '    If iostream.FileOpen And pFileMode = "R" Then
    '        Done = iostream.Done ' Loc(FileNumber) + 1 >= LOF(FileNumber)
    '        ' If LOF(FileNumber) - Loc(FileNumber) < 10 Then Stop
    '        'Done = EOF(FileNumber)
    '    Else
    '        Done = True
    '    End If

End Function

Public Function Entry(ByVal Index As Integer) As Variant

    If Index >= 0 And Index < EntryListCount Then
        Entry = EntryList(Index)
    Else
        Entry = Null
    End If

End Function

Property Get FileType() As String

    FileType = pHeader

End Property

Property Let FileType(Name As String)

    pHeader = Name

End Property

Private Function HasValidHeader() As Boolean

```

```

'This reads characters from the source file until a null
'character is encountered. It then tests to see if it matches
'the FileType

Dim FileHeader As String, Char As Integer, Done As Boolean

FileHeader = iostream.ReadVariable(vbString)

IsValidHeader = (FileHeader = pHeader)

End Function

Public Function OpenFile(FileName As String) As Boolean
    Dim Success As Boolean
    On Error Resume Next

    If iostream.FileOpen Then
        iostream.CloseFile
    End If

    iostream.OpenFile FileName, True

    '===== Check header or react to error
    If iostream.FileOpen Then
        Success = IsValidHeader
    Else
        Success = False
    End If

    OpenFile = Success
End Function

Private Function ReadEntry() As Variant
    Dim Id As Byte, Value As Variant, Result As Variant, b As Byte

    Id = iostream.ReadVariable(vbByte)

    Select Case Id
    Case EntryContentEmpty
        Result = Empty
    Case EntryContentBoolean
        Value = iostream.ReadVariable(vbByte)
        Result = IIf(Value = 1, True, False)
    Case EntryContentInteger
        Result = iostream.ReadVariable(vbInteger)
    Case EntryContentString
        Result = iostream.ReadVariable(vbString)
    Case EntryContentByte
        Result = iostream.ReadVariable(vbByte)
    Case Else

```

```

        MsgBox "Error on CGT read"
        Result = Empty
    End Select

    ReadEntry = Result
End Function

Public Function GetNextRecord() As Boolean

    Dim n As Integer, Count As Integer, Id As Byte, Success As Boolean
    Dim Value As Variant

    If (Not iostream.Done) Then
        Id = iostream.ReadVariable(vbByte)

        Select Case Id
            Case RecordContentMulti
                Clear
                Count = iostream.ReadVariable(vbInteger)
                For n = 1 To Count
                    Store ReadEntry()
                Next
                EntryReadPosition = 0
                Success = True

            Case Else
                Success = False
            End Select

        Else
            Success = False
        End If

        GetNextRecord = Success
    End Function

Private Sub Class_Initialize()

    pHeader = "Simple DataBase"
    EntryReadPosition = 1

    With iostream
        .StringType = StreamStringTypeUnicode
    End With

End Sub

```

```
Private Sub Class_Terminate()

CloseFile

End Sub
```

A.2.2. FAEdge

Each state in the Deterministic Finite Automata contains multiple edges which link to other states in the automata. This class is used internally.

```
'=====
' Class Name:
'     FAEdge
'
' Instantcing:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     Each state in the Determinstic Finite Automata contains multiple
'     edges which link to other states in the automata.
'
'     This class is used to represent an edge.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     (None)
'=====

Public Characters As NumberSet      'Characters to advance on

Public SetIndex As Integer          'Index in the Character Set table
Public Target As Integer
```

A.2.3. FAState

This class represents a state in the Deterministic Finite Automata.

```
'=====
' Class Name:
'     FASState
'
'
' Instancing:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     Represents a state in the Deterministic Finite Automata which
'     is used by the tokenizer.
'
' Author(s):
'     Devin Cook
'     GOLDParser@DevinCook.com
'
' Dependencies:
'     FAEdge
'
'=====

Option Explicit

Private Edges As New ObjectArray

Public AcceptList As New NumberSet
Public TableIndex As Integer

Public Property Get AcceptSymbol() As Integer

    If AcceptList.Count >= 1 Then
        AcceptSymbol = AcceptList.Member(0)
    Else
        AcceptSymbol = -1
    End If

End Property

Public Property Let AcceptSymbol(SymbolIndex As Integer)

    AcceptList.Clear
    If SymbolIndex >= 0 Then
        AcceptList.Add SymbolIndex
    End If

End Property
```


A.2.4. GOLDParse

This is the main class in the GOLD Parser Engine and is used to perform all duties required to the parsing of a source text string. This class contains the LALR(1) State Machine code, the DFA State Machine code, character table (used by the DFA algorithm) and all other structures and methods needed to interact with the developer.

```
'=====
' Class Name:
'   GOLDParse (basic version)
'
' Instantiating:
'   Public; Creatable   (VB Setting: 5 - MultiUse)
'
' Purpose:
'   This is the main class in the GOLD Parser Engine and is used to
'   perform all duties required to the parsing of a source text
'   string. This class contains the LALR(1) State Machine code,
'   the DFA State Machine code, character table (used by the DFA
'   algorithm) and all other structures and methods needed to
'   interact with the developer.
'
'Author(s):
'   Devin Cook
'
'Public Dependencies:
'   Token, Rule, Symbol, Reduction
'
'Private Dependencies:
'   ObjectArray, SimpleDatabase, SymbolList, StringList,
'   VariableList, TokenStack
'
'=====

'=====
'
'           The GOLD Parser Freeware License Agreement
'           =====
'
' this software Is provided 'as-is', without any expressed or
' implied warranty. In no event will the authors be held liable for any
```

```

'damages arising from the use of this software.
'
'Permission is granted to anyone to use this software for any
'purpose. If you use this software in a product, an acknowledgment
'in the product documentation would be deeply appreciated but is
'not required.
'
'In the case of the GOLD Parser Engine source code, permission is
'granted to anyone to alter it and redistribute it freely, subject
'to the following restrictions:
'
'    1. The origin of this software must not be misrepresented; you
'       must not claim that you wrote the original software.
'
'    2. Altered source versions must be plainly marked as such, and
'       must not be misrepresented as being the original software.
'
'    3. This notice may not be removed or altered from any source
'       distribution
'
'=====
Option Explicit

'===== Symbols recognized by the system
Private pSymbolTable As New SymbolList ' ObjectArray

'===== DFA. Contains FASStates.
Private pDFA As New ObjectArray 'FAState

Private pCharacterSetTable As New ObjectArray

'===== Rules. Contains Rule Objects.
Private pRuleTable As New RuleList 'ObjectArray

'===== LALR(1) action table. Contains
LRActionTables.
'This is different from LALR in the Builder which contains LRStates
Private pActionTable As New ObjectArray

'===== Parsing messages
Public Enum GPMessageConstants
    gpMsgTokenRead = 1 'A new token is read
    gpMsgReduction = 2 'A rule is reduced
    gpMsgAccept = 3 'Grammar complete
    gpMsgNotLoadedError = 4 'Now grammar is loaded
    gpMsgLexicalError = 5 'Token not recognized
    gpMsgSyntaxError = 6 'Token is not expected
    gpMsgCommentError = 7 'Reached the end of the file
    gpMsgInternalError = 8 'Something is wrong, very wrong
End Enum

```

```

'===== DFA runtime constants
Private kErrorSymbol As Symbol
Private kEndSymbol As Symbol

'===== DFA runtime variables
Private pInitialDFAState As Integer
Private pLookaheadBuffer As String      'Added 3/7/04

'===== LALR runtime variables
Private pInitialLALRState As Integer
Private pStartSymbol As Long
Private CurrentLALR As Long
Private Stack As New TokenStack

'===== Used for Reductions & Errors
'The set of tokens for 1. Expecting during error, 2. Reduction
Private pTokens      As New TokenStack
Private pHaveReduction As Boolean

Private pTrimReductions As Boolean

'===== Private control variables
Private pTablesLoaded As Boolean
Private pInputTokens As New TokenStack 'Stack of tokens to be
analyzed
Private pSource As New Stream
Private pLineNumber As Long           'Incremented by tokenizer
Private pColumnNumber As Long         'Set by tokenizer
Private pCommentLevel As Integer      'Current level of block
comments (1+)

Private Enum ParseResultConstants
    ParseResultAccept = 1
    ParseResultShift = 2
    ParseResultReduceNormal = 3
    ParseResultReduceEliminated = 4
    ParseResultSyntaxError = 5
    ParseResultInternalError = 6
End Enum

'=====
Private Const RecordIdParameters As Byte = 80 'P
Private Const RecordIdTableCounts As Byte = 84 'T
Private Const RecordIdInitial As Byte = 73 'I
Private Const RecordIdSymbols As Byte = 83 'S
Private Const RecordIdCharSets As Byte = 67 'C
Private Const RecordIdRules As Byte = 82 'R
Private Const RecordIdDFAStates As Byte = 68 'D
Private Const RecordIdLRTables As Byte = 76 'L

```

```

Private Const RecordIdComment As Byte = 33 '!'
Private Const FileHeader = "GOLD Parser Tables/v1.0"

'===== Parameters
Private pParameterName As String
Private pParameterAuthor As String
Private pParameterVersion As String
Private pParameterAbout As String
Private pParameterStartSymbol As String
Private pParameterCaseSensitive As String

Public Function CurrentLineNumber() As Long

    CurrentLineNumber = pLineNumber

End Function

Public Function CurrentColumnNumber() As Long

    CurrentColumnNumber = pColumnNumber

End Function

Public Sub CloseFile()

    pSource.CloseFile

End Sub

Public Function CurrentToken() As Token

    Set CurrentToken = pInputTokens.Top

End Function

Private Sub DiscardRestOfLine()
    Dim EndReached As Boolean, Position As Long

    'Lookahead in the stream (we have buffer and we have to use it)
    'and find the next chr(10) or chr(13). Then discard the lookahead
    'until that point.

    Position = 1
    EndReached = False

```

```

    Do Until EndReached Or pSource.Done()
        Select Case Lookahead(Position)
            Case Chr(10), Chr(13), ""
                EndReached = True
            Case Else
                Position = Position + 1
        End Select
    Loop

    'Discard information - the value returned from this function is
lost
    ReadBuffer Position
End Sub

Public Function PopInputToken() As Token

    Set PopInputToken = pInputTokens.Pop

End Function

Private Sub PrepareToParse()
    Dim Start As New Token

    Start.State = pInitialLALRState
    Set Start.ParentSymbol = pSymbolTable.Member(pStartSymbol)

    Stack.Push Start

End Sub

Public Sub PushInputToken(TheToken As Token)

    pInputTokens.Push TheToken

End Sub

Public Property Get CurrentReduction() As Object
    If pHaveReduction Then
        Set CurrentReduction = Stack.Top.Data
    Else
        Set CurrentReduction = Nothing
    End If
End Property

Public Property Set CurrentReduction(Value As Object)
    If pHaveReduction Then
        Set Stack.Top.Data = Value
    End If
End Property

```

```

Private Function ReadBuffer(ByVal CharCount As Long) As String
    If CharCount <= Len(pLookaheadBuffer) Then
        'Remove the characters from the front of the buffer.
        'This code will
        'be very different in other programming languages

        ReadBuffer = Left(pLookaheadBuffer, CharCount)
        'Remove chars
        pLookaheadBuffer = Mid(pLookaheadBuffer, CharCount + 1)
    Else
        'ERROR - DFA LOGIC DOES NOT ALLOW THIS!
        ReadBuffer = ""
    End If
End Function

Private Function Lookahead(ByVal CharIndex As Long) As String
    Dim NewChars As String, ReadCount As Long

    If CharIndex > Len(pLookaheadBuffer) Then
        '=== We must read characters from the Stream
        ReadCount = CharIndex - Len(pLookaheadBuffer)

        pLookaheadBuffer = pLookaheadBuffer & pSource.Read(ReadCount)
    End If

    'If the buffer is still smaller than the index, we have reached
    'the end of the text. In this case, return a null string - the DFA
    'code will understand

    If CharIndex >= Len(pLookaheadBuffer) Then
        Lookahead = Mid(pLookaheadBuffer, CharIndex, 1)
    Else
        Lookahead = ""
    End If
End Function

Public Sub ShowAboutWindow()

    MsgBox "GOLD Parser Engine" & vbNewLine & App.Major & "." &
    App.Minor & "." & App.Revision

End Sub

Public Sub Clear()
    pSymbolTable.Clear
    pRuleTable.Clear
    pCharacterSetTable.Clear
    pTokens.Clear

```

```

    pInputTokens.Clear
    pActionTable.Clear

    pParameterName = ""
    pParameterVersion = ""
    pParameterAuthor = ""
    pParameterAbout = ""
    pParameterStartSymbol = ""

    Reset
End Sub

Public Property Let TrimReductions(Value As Boolean)
    pTrimReductions = Value
End Property

Public Property Get TrimReductions() As Boolean
    TrimReductions = pTrimReductions
End Property

Public Property Get Parameter(ByVal Name As String) As String

    Select Case UCase(Name)
    Case "NAME"
        Parameter = pParameterName
    Case "VERSION"
        Parameter = pParameterVersion
    Case "AUTHOR"
        Parameter = pParameterAuthor
    Case "ABOUT"
        Parameter = pParameterAbout
    Case "START SYMBOL"
        Parameter = pParameterStartSymbol
    Case "CASE SENSITIVE"
        Parameter = pParameterCaseSensitive
    Case Else
        Parameter = ""
    End Select

End Property

Private Function LoadTables(FileName As String) As Boolean
    On Error GoTo Problem

    Dim File As New SimpleDataBase, ID As Integer, bAccept As Boolean
    Dim n As Integer, SetIndex As Integer, Target As Integer

```



```

Dim ReadSymbol As Symbol, ReadRule As Rule
Dim ReadDFA As FASState, ReadLALR As LRActionTable
Dim Success As Boolean, Text As String

Success = True
File.FileType = FileHeader
If File.OpenFile(FileName, "R") Then
    Do Until File.Done() Or Success = False
        Success = File.GetNextRecord
        ID = File.RetrieveNext()

        Select Case ID
        Case RecordIdParameters
            'Name, Version, Author, About, Case-Sensitive

            pParameterName = File.RetrieveNext
            pParameterVersion = File.RetrieveNext
            pParameterAuthor = File.RetrieveNext
            pParameterAbout = File.RetrieveNext
            pParameterCaseSensitive = CStr(File.RetrieveNext)
            pStartSymbol = Val(File.RetrieveNext)

        Case RecordIdTableCounts
            'Symbol, CharacterSet, Rule, DFA, LALR

            pSymbolTable.ReDimension Val(File.RetrieveNext)
            pCharacterSetTable.ReDimension Val(File.RetrieveNext)
            pRuleTable.ReDimension Val(File.RetrieveNext)
            pDFA.ReDimension Val(File.RetrieveNext)
            pActionTable.ReDimension Val(File.RetrieveNext)

        Case RecordIdInitial
            'DFA, LALR

            pInitialDFAState = File.RetrieveNext
            pInitialLALRState = File.RetrieveNext

        Case RecordIdSymbols
            '#, Name, Kind

            Set ReadSymbol = New Symbol
            n = File.RetrieveNext
            ReadSymbol.Name = File.RetrieveNext
            ReadSymbol.Kind = File.RetrieveNext
            File.RetrieveNext 'Empty

            ReadSymbol.TableIndex = n
            Set pSymbolTable.Member(n) = ReadSymbol

        Case RecordIdCharSets
            '#, Characters

```

```

        n = File.RetrieveNext
        Set pCharacterSetTable.Member(n) =
ToNumberSet(File.RetrieveNext)

    Case RecordIdRules
        '#, ID#, Reserved, (Symbol#, ...)'

        Set ReadRule = New Rule
        n = File.RetrieveNext
        ReadRule.TableIndex = n
        ReadRule.SetRuleNonterminal
pSymbolTable.Member(Val(File.RetrieveNext))
        File.RetrieveNext 'Reserved
        Do Until File.RetrieveDone
            ReadRule.AddItem
pSymbolTable.Member(Val(File.RetrieveNext()))
        Loop
        Set pRuleTable.Member(n) = ReadRule

    Case RecordIdDFAStates
        '#, Accept?, Accept#, Reserved (CharSet#, Target#,
Reserved)...
        Set ReadDFA = New FAState
        n = File.RetrieveNext
        bAccept = File.RetrieveNext

        If bAccept Then
            ReadDFA.AcceptSymbol = File.RetrieveNext
        Else
            ReadDFA.AcceptSymbol = -1
            File.RetrieveNext 'Discard value
        End If
        File.RetrieveNext 'Reserved
'Reserved

        '(Edge chars, Target#, Reserved)...
        Do Until File.RetrieveDone
            SetIndex = File.RetrieveNext 'Char table index
            Target = File.RetrieveNext 'Target
            ReadDFA.AddEdge pCharacterSetTable.Member(SetIndex),
Target, SetIndex
            File.RetrieveNext
'Reserved

        Loop
        Set pDFA.Member(n) = ReadDFA

    Case RecordIdLRTables
        '#, Reserved (Symbol#, Action, Target#, Reserved)...

        Set ReadLALR = New LRActionTable
        n = File.RetrieveNext

```

```

        File.RetrieveNext                'Reserved
    Do Until File.RetrieveDone
        ReadLALR.AddItem
    pSymbolTable.Member(File.RetrieveNext), File.RetrieveNext,
    File.RetrieveNext
        File.RetrieveNext                'Reserved
    Loop
    Set pActionTable.Member(n) = ReadLALR

    Case Else                'RecordIDComment
        Success = False
    End Select

    DoEvents
Loop

pParameterStartSymbol = pSymbolTable.Member(pStartSymbol).Name

File.CloseFile
LoadTables = Success
Else
    LoadTables = False
End If

Exit Function

Problem:

    LoadTables = False
End Function

Public Function SymbolTableCount() As Integer

    SymbolTableCount = SymbolTable.Count

End Function

Public Function RuleTableCount() As Integer

    RuleTableCount = RuleTable.Count

End Function

Public Function SymbolTableEntry(ByVal Index As Integer) As Symbol

    If Index >= 0 And Index < SymbolTable.Count Then
        Set SymbolTableEntry = SymbolTable.Member(Index)
    End If

End Function

```

```

Public Function RuleTableEntry(ByVal Index As Integer) As Rule

    If Index >= 0 And Index < RuleTable.Count Then
        Set RuleTableEntry = RuleTable.Member(Index)
    End If

End Function

Public Function TokenCount() As Integer

    TokenCount = pTokens.Count

End Function

Public Function Tokens(ByVal Index As Integer) As Token

    If Index >= 0 And Index < pTokens.Count Then
        Set Tokens = pTokens.Member(Index)
    Else
        Set Tokens = Nothing
    End If

End Function

Public Function LoadCompiledGrammar(FileName As String) As Boolean
    Reset
    LoadCompiledGrammar = LoadTables(FileName)
End Function
Public Function OpenTextString(Text As String) As Boolean

    Reset
    pSource.Text = Text
    PrepareToParse
    OpenTextString = True

End Function
Public Function Parse() As GPMessageConstants
    '1. If the tables are not setup then report GPM_NotLoadedError
    '2. If parser is in comment mode then read tokens until a
    '   recognized one is found and report it
    '3. Otherwise, parser normal
    '   a. If there are no tokens on the stack
    '       1) Read one and trap error
    '       2) End function with GPM_TokenRead
    '   b. Otherwise, call ParseToken with the top of the stack.
    '       1) If success, then Pop the value
    '       2) Loop if the token was shifted (nothing to report)

    Dim Result As GPMessageConstants, Done As Boolean

```

```

Dim ReadToken As Token, ParseResult As ParseResultConstants

If pActionTable.Count < 1 Or pDFA.Count < 1 Then
    Result = gpMsgNotLoadedError
Else
    Done = False
    Do Until Done
        If pInputTokens.Count = 0 Then
            'We must read a token
            Set ReadToken = RetrieveToken(pSource)
            If ReadToken Is Nothing Then
                Result = gpMsgInternalError
                Done = True
            ElseIf ReadToken.Kind <> SymbolTypeWhitespace Then
                pInputTokens.Push ReadToken
                If pCommentLevel = 0 And ReadToken.Kind <>
SymbolTypeCommentLine And ReadToken.Kind <> SymbolTypeCommentStart Then
                    Result = gpMsgTokenRead
                    Done = True
                End If
            End If
        ElseIf pCommentLevel > 0 Then
            'We are in a block comment
            Set ReadToken = pInputTokens.Pop()

            Select Case ReadToken.Kind
            Case SymbolTypeCommentStart
                pCommentLevel = pCommentLevel + 1
            Case SymbolTypeCommentEnd
                pCommentLevel = pCommentLevel - 1
            Case SymbolTypeEnd
                Result = gpMsgCommentError
                Done = True
            Case Else
                'Do nothing, ignore
                'The 'comment line' symbol is ignored as
                'well
            End Select
        Else
            '=== We are ready to parse
            Set ReadToken = pInputTokens.Top

            Select Case ReadToken.Kind
            Case SymbolTypeCommentStart
                pCommentLevel = pCommentLevel + 1
                pInputTokens.Pop 'Remove it
            Case SymbolTypeCommentLine
                'Remove it and rest of line
                'Procedure also increments the line number
                pInputTokens.Pop
            End Select
        End If
    Loop
End Do

```

```

        DiscardRestOfLine
    Case SymbolTypeError
        Result = gpMsgLexicalError
        Done = True
    Case Else      'FINALLY, we can parse the token
        ParseResult = ParseToken(ReadToken)

        Select Case ParseResult
        Case ParseResultAccept
            Result = gpMsgAccept
            Done = True
        Case ParseResultInternalError
            Result = gpMsgInternalError
            Done = True
        Case ParseResultReduceNormal
            Result = gpMsgReduction
            Done = True
        Case ParseResultShift  'A simple shift, we must
continue
                                pInputTokens.Pop  'Okay, remove the top token,
it is on the stack
        Case ParseResultSyntaxError
            Result = gpMsgSyntaxError
            Done = True
        Case Else
            'Do nothing
        End Select

    End Select
End If
Loop

End If

Parse = Result

End Function

Private Function ParseToken(NextToken As Token) As ParseResultConstants
    'This function analyzes a token and either:
    ' 1. Makes a SINGLE reduction and pushes a complete Reduction
    '    object on the stack
    ' 2. Accepts the token and shifts
    ' 3. Errors and places the expected symbol indexes in the
    '    Tokens list. The Token is assumed to be valid and WILL
    '    be checked. If an action is performed that requires control
    '    to be returned to the user, the function returns true.
    '    The Message parameter is then set to the type of action.

    Dim n As Integer, Found As Boolean, Index As Integer, RuleIndex As
Integer, CurrentRule As Rule

```

```

Dim str As String, Head As Token, NewReduction As Reduction
Dim Result As ParseResultConstants

Index =
pActionTable.Member(CurrentLALR).ActionIndexForSymbol(NextToken.ParentS
ymbol.TableIndex)

If Index <> -1 Then                                'Work - shift or reduce
    pHaveReduction = False                        'Will be set true if a reduction is
made
    pTokens.Count = 0

    Select Case pActionTable.Member(CurrentLALR).Item(Index).Action
    Case ActionAccept
        pHaveReduction = True
        Result = ParseResultAccept

    Case ActionShift
        CurrentLALR =
pActionTable.Member(CurrentLALR).Item(Index).Value
        NextToken.State = CurrentLALR
        Stack.Push NextToken
        Result = ParseResultShift

    Case ActionReduce
        'Produce a reduction - remove as many tokens as members in
the rule & push a nonterminal token

        RuleIndex =
pActionTable.Member(CurrentLALR).Item(Index).Value
        Set CurrentRule = pRuleTable.Member(RuleIndex)

        '==== Create Reduction
        If pTrimReductions And CurrentRule.ContainsOneNonTerminal
Then
            'The current rule only consists of a single nonterminal
            'and can be trimmed from the parse tree. Usually we
            'create a new Reduction, assign it to the Data property
            'of Head and push it on the stack. However, in this
case,
            'the Data property of the Head will be assigned the Data
            'property of the reduced token (i.e. the only one' on the
            'stack). In this case, to save code, the value popped of
            'the stack is changed into the head.

            Set Head = Stack.Pop()
            Set Head.ParentSymbol = CurrentRule.RuleNonterminal

            Result = ParseResultReduceEliminated
        Else
            'Build a

```

```

Reduction
    pHaveReduction = True
    Set NewReduction = New Reduction
    With NewReduction
        Set .ParentRule = CurrentRule
        .TokenCount = CurrentRule.SymbolCount
        For n = .TokenCount - 1 To 0 Step -1
            Set .Tokens(n) = Stack.Pop()
        Next
    End With

    Set Head = New Token
    Set Head.Data = NewReduction
    Set Head.ParentSymbol = CurrentRule.RuleNonterminal

    Result = ParseResultReduceNormal
End If

    'Goto
    Index = Stack.Top().State

    'If n is -1 here, then we have an Internal Table Error!!!!
    n =
pActionTable.Member(Index).ActionIndexForSymbol(CurrentRule.RuleNonterminal.TableIndex)
    If n <> -1 Then
        CurrentLALR = pActionTable.Member(Index).Item(n).Value

        Head.State = CurrentLALR
        Stack.Push Head
    Else
        Result = ParseResultInternalError
    End If
End Select

Else
    '=== Syntax Error! Fill Expected Tokens
    pTokens.Clear
    For n = 0 To pActionTable.Member(CurrentLALR).Count - 1
        Select Case
pActionTable.Member(CurrentLALR).Item(n).Symbol.Kind
            Case SymbolTypeTerminal, SymbolTypeEnd
                Set Head = New Token
                Head.Data = ""
                Set Head.ParentSymbol =
ActionTable.Member(CurrentLALR).Item(n).Symbol
                pTokens.Push Head
            End Select
        Next
    'If pTokens.Count = 0 Then Stop
    Result = ParseResultSyntaxError

```



```

End If

ParseToken = Result      'Very important

End Function

Public Function OpenFile(ByVal FileName As String, Optional ByVal
DetectEncodingFromByteOrderMarks = True) As Boolean
    Dim Success As Boolean
    Reset

    Success = pSource.OpenFile(FileName, StreamFileModeRead,
StreamTransferText, DetectEncodingFromByteOrderMarks) ', StreamType)

    PrepareToParse
    OpenFile = Success
End Function

Public Sub Reset()
    On Error GoTo Problem

    Dim n As Integer

    '===== Setup global variables
    For n = 0 To pSymbolTable.Count - 1
        Select Case pSymbolTable.Member(n).Kind
            Case SymbolTypeError
                Set kErrorSymbol = pSymbolTable.Member(n)
            Case SymbolTypeEnd
                Set kEndSymbol = pSymbolTable.Member(n)
        End Select
    Next

    CurrentLALR = pInitialLALRState
    pLineNumber = 1
    pColumnNumber = 0      'First char makes it column 1
    pSource.CloseFile
    pCommentLevel = 0
    pHaveReduction = False

    pTokens.Clear
    pInputTokens.Clear
    Stack.Clear
    pLookaheadBuffer = ""

    Exit Sub

Problem:
    Err.Clear
End Sub

```

```

Private Function RetrieveToken(Source As Stream) As Token      'Symbol
Index
    'THIS IS THE TOKENIZER!
    '
    'This function implements the DFA algorithm and returns
    'a token to the LALR state machine

    Dim ch As String, n As Integer, Found As Boolean
    Dim Done As Boolean, Target As Integer
    Dim CharSetIndex As Integer, CurrentDFA As Integer
    Dim CurrentPosition As Long
    Dim LastAcceptState As Integer, LastAcceptPosition As Integer
    Dim Result As New Token

    Done = False
    CurrentDFA = pInitialDFAState 'The first state is almost always #1.
    CurrentPosition = 1           'Next byte in the input Stream
    LastAcceptState = -1          'We have not yet accepted a character string
    LastAcceptPosition = -1

    If Lookahead(1) <> "" Then      'Not yet EOF
        Do Until Done
            ' This code searches all the branches of the current DFA
            ' state for the next character in the input Stream. If
            ' found the target state is returned. The InStr()
            ' function searches the string
            ' pCharacterSetTable.Member(CharSetIndex) starting at
            ' position 1 for ch. The pCompareMode variable determines
            ' whether the search is case sensitive.

            ch = Lookahead(CurrentPosition)
            If ch = "" Then          'End reached, do not match
                Found = False
            Else
                n = 0
                Found = False
                Do While n < pDFA.Member(CurrentDFA).EdgeCount And Not
Found
                    CharSetIndex = pDFA.Member(CurrentDFA).Edge(n).SetIndex
                    If
pCharacterSetTable.Member(CharSetIndex).HasMember(AscW(ch)) Then
                        Found = True
                        Target = pDFA.Member(CurrentDFA).Edge(n).Target
                    '.TableIndex
                End If
                n = n + 1
            Loop
        End If

        ' This block-if statement checks whether an edge was found
        ' from the current state. If so, the state and current

```

```

' position advance. Otherwise it is time to exit the main
' loop and report the token found (if there was it fact one).
' If the LastAcceptState is -1, then we never found a match
' and the Error Token is created. Otherwise, a new token
' is created using the Symbol in the Accept State and all
' the characters that comprise it.

If Found Then
    ' This code checks whether the target state accepts a
    ' token. If so, it sets the appropriate variables so when
    ' the algorithm is done, it can return the proper
    ' token and number of characters.

    If pDFA.Member(Target).AcceptSymbol <> -1 Then
        LastAcceptState = Target
        LastAcceptPosition = CurrentPosition
    End If

    CurrentDFA = Target
    CurrentPosition = CurrentPosition + 1
Else
    Done = True
    If LastAcceptState = -1 Then
        'Tokenizer cannot recognize symbol
        Set Result.ParentSymbol = kErrorSymbol
        Result.Data = ReadBuffer(1)
    Else
        'Create Token, read characters
        'The data contains the total number of accept
characters
        Set Result.ParentSymbol =
pSymbolTable.Member(pDFA.Member(LastAcceptState).AcceptSymbol)
        Result.Data = ReadBuffer(LastAcceptPosition)
    End If
End If
DoEvents
Loop
Else
    Result.Data = "" 'End of file reached, create End Token
    Set Result.ParentSymbol = kEndSymbol
End If

' Count Carriage Returns and increment the Line Number. This is
' done for the Developer and is not necessary for the DFA
' algorithm

For n = 1 To Len(Result.Data)
    If Mid(Result.Data, n, 1) = vbCr Then
        pLineNumber = pLineNumber + 1
        pColumnNumber = 0
    
```

```
        Else
            pColumnNumber = pColumnNumber + 1
        End If
    Next

    Set RetrieveToken = Result
End Function

Private Sub Class_Initialize()
    Reset
    pTablesLoaded = False

    pTrimReductions = True
End Sub

Private Sub Class_Terminate()
    Set pSymbolTable = Nothing
    Set pDFA = Nothing
    Set pCharacterSetTable = Nothing
    Set pRuleTable = Nothing
    Set pActionTable = Nothing

    Set kErrorSymbol = Nothing
    Set kEndSymbol = Nothing

    Set Stack = Nothing
    Set pTokens = Nothing

    Set pInputTokens = Nothing
    Set pSource = Nothing
End Sub
```

A.2.5. LRAction

This class represents an action in a LALR State. There is one and only one action for any given symbol.

```
'=====
' Class Name:
'     LRAction
'
' Instanting:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     This class represents an action in a LALR State. There is one and
'     only one action for any given symbol.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     (None)
'
'=====
Option Explicit

Public Enum ActionConstants
    ActionShift = 1      'Shift a symbol and goto a state
    ActionReduce = 2     'Reduce by a specified rule
    ActionGoto = 3       'Goto to a state on reduction
    ActionAccept = 4     'Input successfully parsed
    ActionError = 5      'Programmers see this often!
End Enum

Private pSymbol As Symbol
Public Action As ActionConstants
Public Value As Integer      'shift to state, reduce rule, goto state

Public Property Set Symbol(Sym As Symbol)

    Set pSymbol = Sym
End Property
```

```

Public Property Get Symbol() As Symbol
    Set Symbol = pSymbol
End Property

Public Function SymbolIndex() As Integer
    SymbolIndex = pSymbol.TableIndex
End Function

```

A.2.6. LRActionTable

This class contains the actions (reduce/shift) and goto information for a STATE in a LR parser. Essentially, this is just a row of actions in the LR state transition table. The only data structure is a list of LR Actions.

```

'=====
' Class Name:
'     LRActionTable
'
' Instancing:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     This class contains the actions (reduce/shift) and goto information
'     for a STATE in a LR parser. Essentially, this is just a row of
actions
'     in the LR state transition table. The only data structure is a list
'     of LR Actions.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     LRAction Class

```

```

'
'=====

Option Explicit

Private MemberList() As LRAAction
Private MemberCount As Long

Public Function ActionIndexForSymbol(SymbolIndex As Integer) As Integer
    'Returns the index of SymbolIndex in the table, -1 if not found
    Dim n As Integer, Found As Boolean, Index As Integer

    n = 0
    Found = False
    Do While Not Found And n < MemberCount
        If MemberList(n).Symbol.TableIndex = SymbolIndex Then
            Index = n
            Found = True
        End If
        n = n + 1
    Loop

    If Found Then
        ActionIndexForSymbol = Index
    Else
        ActionIndexForSymbol = -1
    End If

End Function

Public Sub AddItem(TheSymbol As Symbol, Action As ActionConstants, Value As Integer)
    Dim TableEntry As New LRAAction

    Set TableEntry.Symbol = TheSymbol
    TableEntry.Action = Action
    TableEntry.Value = Value

    MemberCount = MemberCount + 1
    ReDim Preserve MemberList(0 To MemberCount - 1)      'Change the size of
the array
    Set MemberList(MemberCount - 1) = TableEntry

End Sub

Public Function Count() As Integer

```

```

        Count = MemberCount
End Function

Public Function Item(ByVal n As Integer) As LRAction

    If n >= 0 And n < MemberCount Then
        Set Item = MemberList(n)
    End If

End Function

```

A.2.7. NumberSet

The NumberSet is used to store list of 2 byte integers. Essentially, this class is used to store the Unicode character sets used by the Deterministic Finite Automata. To optimized the class for speed, all lists are stored in sorted order. This allows the binary search algorithm to be used to check if a number character (number) is present.

```

Option Explicit

Private MemberList() As Integer
Private MemberCount As Long

Public Sub Clear()

    Erase MemberList
    MemberCount = 0

End Sub

Public Function Count() As Long

    Count = MemberCount

End Function

```



```

Public Function HasMember(Number As Integer) As Boolean

    HasMember = (MemberIndex(Number) <> -1)

End Function

Private Function MemberIndex(ByVal Number As Integer) As Long
    Dim Done As Boolean
    Dim Upper As Long, Lower As Long, Middle As Long, Index As Long

    If MemberCount = 0 Then
        Index = -1
    ElseIf Number < MemberList(0) Or Number > MemberList(MemberCount -
1) Then
        Index = -1
    Else
        'THIS IS A BINARY SEARCH - since the list is sorted!
        Upper = MemberCount - 1
        Lower = 0
        Index = -1
        Done = False

        Do
            Middle = (Lower + Upper) / 2

            'Two ends passed each other, fail - member not found
            If Lower > Upper Then Done = True
            ElseIf MemberList(Middle) = Number Then
                Index = Middle
                Done = True
            ElseIf MemberList(Middle) < Number Then
                Lower = Middle + 1
            ElseIf MemberList(Middle) > Number Then
                Upper = Middle - 1
            End If
        Loop Until Done
    End If

    MemberIndex = Index
End Function

Property Get Member(ByVal Index As Long) As Long
Attribute Member.VB_UserMemId = 0

    If Index >= 0 And Index < MemberCount Then
        Member = MemberList(Index)
    End If

End Property

```

```

Property Let Member(ByVal Index As Long, var As Long)

    If Index >= 0 And Index < MemberCount Then
        MemberList(Index) = var
    End If
End Property

Public Sub Add(ByVal Number As Integer)

    'Insert the Number into the list at the correct position -
    maintaining a sorted
    'list

    Dim Index As Integer, n As Integer, Found As Boolean

    '=== Find point of insertion, before Index.
    n = 0
    Index = -1
    Found = False
    Do Until n > MemberCount - 1 Or Index <> -1 Or Found
        If MemberList(n) = Number Then
            Found = True
        ElseIf MemberList(n) > Number Then
            Index = n
        End If
        n = n + 1
    Loop

    If Not Found Then
        '=== Increase array
        MemberCount = MemberCount + 1
        ReDim Preserve MemberList(0 To MemberCount - 1)

        If Index = -1 Then
            'Not found Add last
            MemberList(MemberCount - 1) = Number
        Else
            'Insert, Move items
            For n = MemberCount - 1 To Index + 1 Step -1
                MemberList(n) = MemberList(n - 1)
            Next
            MemberList(Index) = Number
        End If
    End If
End Sub

```

A.2.8. ObjectArray

This class is used to store a list of objects. In Visual Basic, an object can contain any class. This class would have been used universally in the code, but limitations imposed by the Visual Basic programming language required the creation of specialized list classes.

```
'=====
' Class Name:
'     ObjectArray
'
' Instancing:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     This class is used to store a list of objects. In Visual Basic,
'     an object can contain any class. This class would have been used
'     universally in the code, but limitations imposed by the Visual Basic
'     programming language required the creation of specialized list
'     classes.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     (None)
'=====
Option Explicit

Private MemberList() As Object
Private MemberCount As Long

Public Sub Clear()

    Erase MemberList
    MemberCount = 0

End Sub

Public Function Count() As Long

    Count = MemberCount

End Function
```

```

Property Get Member(ByVal Index As Long) As Object

    If Index >= 0 And Index < MemberCount Then
        Set Member = MemberList(Index)
    Else
        Set Member = Nothing
        '    MsgBox "Nothing!"
    End If
End Property

Property Let Member(ByVal Index As Long, Obj As Object)

    If Index >= 0 And Index < MemberCount Then
        MemberList(Index) = Obj
    End If

End Property

Property Set Member(ByVal Index As Long, Obj As Object)

    If Index >= 0 And Index < MemberCount Then
        Set MemberList(Index) = Obj
    End If

End Property

Public Sub Add(Obj As Object)

    MemberCount = MemberCount + 1
    ReDim Preserve MemberList(0 To MemberCount - 1)
    Set MemberList(MemberCount - 1) = Obj

End Sub

Public Function IsEqual(Obj As ObjectArray) As Boolean
    Dim n As Integer
    Dim Equal As Boolean

    If MemberCount = Obj.Count Then
        Equal = True
        n = 0
        Do While Equal And n = Count
            Equal = MemberList(n) = Obj.Member(n)
            n = n + 1
        Loop
    Else
        Equal = False
    End If

    IsEqual = Equal

```

```

End Function

Public Function Pop() As Object

    MsgBox "Pop on ObjectArray"

    'Set Pop = MemberList(MemberCount)

    'MemberCount = MemberCount - 1
    'ReDim Preserve MemberList(1 To MemberCount)

End Function
Public Sub ReDimension(ByVal NewCount As Long)

    MemberCount = NewCount
    If NewCount = 0 Then
        ReDim MemberList(0 To 0)
        Set MemberList(0) = Nothing
    Else
        ReDim Preserve MemberList(0 To MemberCount - 1)
    End If

End Sub

Private Sub Class_Terminate()
    Dim n As Integer
    For n = 0 To MemberCount - 1
        Set MemberList(n) = Nothing
    Next
End Sub

```

A.2.9. Reduction

This class is used by the engine to hold a reduced rule. Rather than contain a list of Symbols, a reduction contains a list of Tokens corresponding to the rule it represents. This class is important since it is used to store the actual source program parsed by the Engine.

```

'=====
' Class Name:
'     Reduction
'
' Instancing:
'     Public; Creatable (VB Setting: 5 - MultiUse)
'
' Purpose:
'     This class is used by the engine to hold a reduced rule. Rather
'     than contain a list of Symbols, a reduction contains a list of
'     Tokens corresponding to the rule it represents. This class is
'     important since it is used to store the
'     actual source program parsed by the Engine.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     Token Class, Rule Class
'=====

Option Explicit

Private pTokens() As Token
Private pTokenCount As Integer

Private pParentRule As Rule

Private pTag As Integer           'General purpose

Friend Property Let TokenCount(Value As Integer)

    If Value < 1 Then
        Erase pTokens
        pTokenCount = 0
    Else
        pTokenCount = Value
        ReDim Preserve pTokens(0 To pTokenCount - 1)    'Change the size of
the array
    End If

End Property

Public Property Get ParentRule() As Rule

```

```
        Set ParentRule = pParentRule
    End Property

    Friend Property Set ParentRule(Value As Rule)

        Set pParentRule = Value
    End Property

    Property Let Tag(Value As Integer)
        pTag = Value
    End Property

    Property Get Tag() As Integer
        Tag = pTag
    End Property

    Public Property Get TokenCount() As Integer
        TokenCount = pTokenCount
    End Property

    Public Property Get Tokens(Index As Integer) As Token

        If Index >= 0 And Index < pTokenCount Then
            Set Tokens = pTokens(Index)
        Else
            Set Tokens = Nothing
        End If
    End Property

    Friend Property Set Tokens(Index As Integer, Value As Token)

        If Index >= 0 And Index < pTokenCount Then
            Set pTokens(Index) = Value
        'Else
        '    MsgBox "Error"
        End If
    End Property
```

A.2.10. Rule

The Rule class is used to represent the logical structures of the grammar. Rules consist of a head containing a nonterminal series of followed by a both nonterminals and terminals.

```

'=====
' Class Name:
'     Rule
'
'
' Instancing:
'     Public; Non-creatable  (VB Setting: 2- PublicNotCreatable)
'
' Purpose:
'     The Rule class is used to represent the logical structures of the
'     grammar. Rules consist of a head containing a nonterminal series
of
'     followed by a both nonterminals and terminals.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     Symbol Class, SymbolList Class
'
'=====
Option Explicit

Private pRuleNonterminal As Symbol
Private pRuleSymbols As New ObjectArray  'This consist the body of the
rule
Private pTableIndex As Integer

Friend Function ContainsOneNonTerminal() As Boolean
    'New 12/2001 - used by the GOLDParse object to TrimReductions

    Dim Result As Boolean

    Result = False

    If pRuleSymbols.Count = 1 Then
        If pRuleSymbols.Member(0).Kind = SymbolTypeNonterminal Then
            Result = True
        End If
    End If

    ContainsOneNonTerminal = Result

```



```

End Function

Friend Function Definition() As String
    Dim n As Integer, str As String

    For n = 0 To pRuleSymbols.Count - 1
        str = str & pRuleSymbols.Member(n).Text & " "
    Next

    Definition = RTrim(str)
End Function

Public Property Get SymbolCount() As Integer

    SymbolCount = pRuleSymbols.Count

End Property

Friend Function Name() As String

    Name = "<" & pRuleNonterminal.Name & ">"

End Function

Friend Sub SetRuleNonterminal(Nonterminal As Symbol)

    Set pRuleNonterminal = Nonterminal

End Sub

Public Property Get RuleNonterminal() As Symbol

    Set RuleNonterminal = pRuleNonterminal

End Property

Public Property Get Symbols(Index As Integer) As Symbol

    If Index >= 0 And Index < pRuleSymbols.Count Then
        Set Symbols = pRuleSymbols.Member(Index)
    End If

End Property

```

```

Friend Sub AddItem(Item As Symbol)

    pRuleSymbols.Add Item

End Sub
Friend Property Let TableIndex(Index As Integer)

    pTableIndex = Index

End Property

Public Property Get TableIndex() As Integer

    TableIndex = pTableIndex

End Property
Public Function Text() As String

    Text = Name() & " ::= " & Definition()

End Function

Private Sub Class_Initialize()
    pTableIndex = -1
End Sub

```

A.2.11. RuleList

This class is used to store a list of Rules.

```

'=====
' Class Name:
'     RuleList
'
' Purpose:
'     This class is used to store a list of Rule objects
'
' Author(s):
'     Devin Cook

```

```

'
' Dependencies:
'     Rule Class
'
'=====

Option Explicit

Private MemberList() As Rule
Private MemberCount As Long

Public Sub ReDimension(ByVal NewCount As Long)

    MemberCount = NewCount
    ReDim Preserve MemberList(0 To MemberCount - 1)

End Sub

Public Sub Clear()

    Erase MemberList
    MemberCount = 0

End Sub

Public Function Count() As Long

    Count = MemberCount

End Function

Property Get Member(ByVal Index As Long) As Rule

    If Index >= 0 And Index < MemberCount Then
        Set Member = MemberList(Index)
    Else
        Set Member = Nothing
    End If

End Property

Property Let Member(ByVal Index As Long, Obj As Rule)

    If Index >= 0 And Index < MemberCount Then
        MemberList(Index) = Obj
    End If

```

```
End Property

Property Set Member(ByVal Index As Long, Obj As Rule)

    If Index >= 0 And Index < MemberCount Then
        Set MemberList(Index) = Obj
    End If

End Property

Public Function Add(Obj As Rule) As Integer

    MemberCount = MemberCount + 1
    ReDim Preserve MemberList(0 To MemberCount - 1)
    Set MemberList(MemberCount - 1) = Obj

    Add = MemberCount - 1

End Function

Private Sub Class_Terminate()
    '== 12/20/2001: Added the set commands
    Dim n As Integer
    For n = 0 To MemberCount - 1
        Set MemberList(n) = Nothing
    Next
End Sub
```

A.2.12. Stream

This class is used to create a very basic version of the stream object such as the one that exists in the C++ programming language. Using streams, the programmer can read data from any number of sources while masking this from the code using the stream. Visual Basic .NET adds this functionality.

```
'=====
' Class Name:
'     Stream (Basic Version)
'
' Instancing:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     This class is used to create a very basic version of the
'     stream object that exists in the C++ programming language.
'     Using streams, the programmer can read data from any number
'     of sources while masking this from the code using the
'     stream. Visual Basic .NET will add this functionality.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     (None)
'=====

Option Explicit

Public Enum StreamTransferConstants
    StreamTransferBinary = 1
    StreamTransferText = 2
End Enum

'==== For deciding whether to write strings in Binary mode
'==== as C++ (null end) or with a 2-byte predecessor field
Public Enum StreamStringFormConstants
    StreamStringFormNull = 1           '== Null character delimited
```

```

    StreamStringFormNone = 2           '== Raw data
    StreamStringFormField = 3          '== 2-Byte prefix integer
End Enum

Public Enum StreamStringTypeConstants
    StreamStringTypeASCII = 1
    StreamStringTypeUnicode = 2
End Enum

Public Enum StreamEncodingConstants
    StreamEncodingLittleEndian = 1     '== For those x86 guys
    StreamEncodingBigEndian = 2         '== For those Motorola guys
End Enum

Private Enum StreamTargetConstants
    StreamTargetString
    StreamTargetFile
End Enum

'===== File Streaming
Private m_FileNumber As Integer
Private m_FileOpen As Boolean
Private m_StreamTransfer As StreamTransferConstants

'===== The main data type and control
Private m_Buffer As String             'Up to 2 billion chars

Private m_StreamTarget As StreamTargetConstants
Private m_StreamCanWrite As Boolean
Private m_StreamCanRead As Boolean

'===== Public data types
Private m_StringForm As StreamStringFormConstants
Private m_StringType As StreamStringTypeConstants
Private m_Encoding As StreamEncodingConstants

'===== This is a very useful ADT that organizes very complex binary
data
Private Type ByteSequence
    Length As Integer                  'In bytes, not bits
    Bytes(0 To 15) As Byte            '0 is LSB
End Type

Public Property Get Encoding() As StreamEncodingConstants
    Encoding = m_Encoding
End Property

Public Property Let Encoding(ByVal Value As StreamEncodingConstants)
    m_Encoding = Value

```

```

End Property

Public Property Get StringType() As StreamStringTypeConstants
    StringType = m_StringType
End Property

Public Property Let StringType(ByVal Value As
StreamStringTypeConstants)
    m_StringType = Value
End Property

Public Property Get StringForm() As StreamStringFormConstants
    StringForm = m_StringForm
End Property

Public Property Let StringForm(ByVal Value As
StreamStringFormConstants)
    m_StringForm = Value
End Property

Private Sub CloseStream()

    If m_StreamTarget = StreamTargetFile And m_FileOpen Then
        Close m_FileNumber
        m_FileOpen = False
    End If

End Sub

Public Function EOF() As Boolean
    EOF = Done()
End Function

Public Function FileOpen() As Boolean
    FileOpen = m_FileOpen
End Function

Private Function ReadByteSequence(Length As Integer) As ByteSequence

    'This function is where I will put the Big/Little endian code

    Dim str As String, Result As ByteSequence, n As Integer
    str = StreamRead(Length)

    Result.Length = Length

    If m_Encoding = StreamEncodingBigEndian Then
        For n = 1 To Length
            Result.Bytes(n - 1) = Asc(Mid(str, n, 1))
        Next
    End If

```



```

        CharNumber = DecodeInt16(ReadByteSequence(2))
        Variable = ""
        'First char is assigned in first loop
        Do Until CharNumber = 0
            Variable = Variable & ChrW(CharNumber)
            CharNumber = DecodeInt16(ReadByteSequence(2))
        Loop
    End Select
End Select

    End Select
Else
    Variable = ReadUntil(Chr(0))
End If

ReadVariable = Variable

End Function

Private Function EncodeInt16(ByVal Value As Integer) As ByteSequence
    Dim Temp As ByteSequence

    If Value < 0 Then
        'Negative - two's compliment: -1
        & negate
        Temp = CByteSequence(Abs(Value + 1), 2) 'Note: Abs(Value+1) ==
        Abs(Value)-1 when Value<0
        EncodeInt16 = Negate(Temp)
    Else
        'Positive or zero
        EncodeInt16 = CByteSequence(Value, 2)
    End If

End Function

Private Function DecodeInt16(Source As ByteSequence) As Integer

    Dim Temp As ByteSequence

    If GetBit(Source, 15) = 1 Then
        'Negative
        Temp = Negate(Source)
        DecodeInt16 = -(CNumber(Temp) + 1)
    Else
        'Positive or zero
        DecodeInt16 = CNumber(Source)
    End If

End Function

Private Function DecodeInt32(Source As ByteSequence) As Long

    Dim Temp As ByteSequence

    If GetBit(Source, 31) = 1 Then
        'Negative

```

```

        Temp = Negate(Source)
        DecodeInt32 = -(CNumber(Temp) + 1)
    Else
        DecodeInt32 = CNumber(Source)
    End If

End Function

Private Function EncodeInt32(ByVal Value As Double) As ByteSequence
    Dim Temp As ByteSequence

    If Value < 0 Then
        Temp = CByteSequence(Abs(Value + 1), 4)
        'Negative - two's compliment: -1 & negate
        'Note: Abs(Value+1) == Abs(Value)-1 when Value<0
        EncodeInt32 = Negate(Temp)
    Else
        EncodeInt32 = CByteSequence(Value, 4)
        'Positive or zero
    End If

End Function

End Function

Private Function CByteSequence(ByVal Value As Variant, ByteCount As Integer) As ByteSequence

    Dim TheByte As Byte, i As Integer, Result As ByteSequence

    '===Copy whole bytes
    For i = 0 To ByteCount - 1
        TheByte = Int(Value) - Int(Value / 256) * 256
        Result.Bytes(i) = TheByte
        Value = Fix(Value / 256)
    Next

    Result.Length = ByteCount

    CByteSequence = Result

End Function

Private Function CNumber(Sequence As ByteSequence) As Double

    Dim TheByte As Byte, i As Integer, Result As Double

    Result = 0
    '===Copy whole bytes
    For i = Sequence.Length - 1 To 0 Step -1
        Result = Result * 256 + Sequence.Bytes(i)
    Next
    'Move

```

```

    CNumber = Result

End Function

Private Function Negate(Source As ByteSequence) As ByteSequence

    Dim Result As ByteSequence, n As Integer

    Result = Source
    For n = 0 To Result.Length - 1
        Result.Bytes(n) = Not Result.Bytes(n)
    Next

    Negate = Result

End Function

Private Function GetBit(Source As ByteSequence, BitNumber As Integer)
As Byte

    'Returns the bit found in the BinaryBuffer starting at index
    '0 in last byte and moving to the left

    Dim ByteIndex As Integer, BitIndex As Integer

    ByteIndex = BitNumber \ 8           'Whole bytes
    BitIndex = BitNumber Mod 8

    If ByteIndex < Source.Length Then
        GetBit = IIf((Source.Bytes(ByteIndex) And (2 ^ BitIndex)) = 0, 0,
1)
    Else
        'Outside buffer = assume 0
        GetBit = 0
    End If

End Function

Private Sub WriteByteSequence(Sequence As ByteSequence)
    'This function is where I will put the Big/Little endian code

    Dim n As Integer, str As String

    If m_Encoding = StreamEncodingLittleEndian Then           '0=LSB
        For n = 0 To Sequence.Length - 1
            str = str & Chr(Sequence.Bytes(n))
        Next
    End Sub

```

```

Else
    For n = Sequence.Length - 1 To 0 Step -1
        str = str & Chr(Sequence.Bytes(n))
    Next
End If

StreamWrite str
End Sub

Private Sub StreamWrite(ByVal Text As String)

    'This is the sub that adds info to the buffer and then
    'flushes it if necessary. Information is converted
    'automatically based
    Dim n As Integer

    Select Case m_StreamTarget
    Case StreamTargetFile
        If m_FileOpen And m_StreamCanWrite Then
            Print #m_FileNumber, Text;
        End If

        Case StreamTargetString
            m_Buffer = m_Buffer & Text
        End Select
    End Sub

Public Sub CloseFile()

    If m_FileOpen Then
        CloseStream
    End If

End Sub

Public Function Done() As Boolean

    Select Case m_StreamTarget
    Case StreamTargetString
        Done = Len(m_Buffer) = 0
    Case StreamTargetFile
        If Not m_FileOpen Then
            Done = True
        Else
            Done = VBA.EOF(m_FileNumber)
        End If
    End Select

End Function

Public Function OpenFile(Filename As String, Mode As String) As Boolean

```

```

On Error Resume Next
Dim Success As Boolean, n As Long

If m_FileOpen Then
    Close m_FileNumber
End If

m_StreamCanWrite = False
m_StreamCanRead = False

If LCase(Right(Mode, 1)) = "b" Then
    m_StreamTransfer = StreamTransferBinary
Else
    m_StreamTransfer = StreamTransferText
End If

Success = True 'Unless determined otherwise
Select Case UCase(Left(Mode, 1))
Case "W"
    m_StreamCanWrite = True
    m_FileNumber = FreeFile
    Open FileName For Output As m_FileNumber
Case "R"
    m_StreamCanRead = True
    m_FileNumber = FreeFile
    Open FileName For Binary Access Read As m_FileNumber
Case "A"
    m_StreamCanWrite = True
    m_FileNumber = FreeFile
    Open FileName For Append As m_FileNumber
Case Else
    Success = False
End Select

'===== Check header or react to error
If Err.Number = 0 And Success = True Then
    m_FileOpen = True
    m_StreamTarget = StreamTargetFile
    m_StringForm = StreamStringFormNone
Else
    Err.Clear
    m_FileOpen = False
    Success = False
End If

OpenFile = Success
End Function

```

```

Public Function WriteLine(Text As String) As Boolean

    StreamWrite Text & vbNewLine

End Function

Public Function Read(Optional ByVal Length As Integer = 1) As String

    'This function reads the appropriate amount of characters from the
    'buffer and constructs the requested datatype

    If m_StreamCanRead Then
        Read = StreamRead(Length)
    Else
        Read = ""
    End If

End Function

Public Function ReadUntil(ByVal EndChar As String) As String

    Dim EndReached As Boolean, TextSegment As String, ch As String

    If m_StreamCanRead Then
        If EndChar = "" Then EndChar = Left(vbNewLine, 1)
        If Len(EndChar) > 1 Then EndChar = Left(EndChar, 1)

        EndReached = False
        Do Until EndReached Or Done()
            ch = NextCharacter()
            If StrComp(ch, EndChar, vbBinaryCompare) <> 0 Then
                TextSegment = TextSegment & ch
            Else
                EndReached = True
            End If
        Loop
        ReadUntil = TextSegment

    Else
        ReadUntil = ""
    End If

End Function

Private Function StreamRead(Optional ByVal Size As Integer = 1) As
String

```

```

'This function takes data from the buffer and creates a string
'of the appropriate size.

Dim StreamLeft As Long, Result As String

If Size < 0 Then
    Size = 0
End If

Select Case m_StreamTarget
Case StreamTargetFile
    If m_FileOpen Then          'Read from file
        '=== The total bytes left
        If VBA.EOF(m_FileNumber) Then
            StreamLeft = 0
        Else
            StreamLeft = LOF(m_FileNumber) - Seek(m_FileNumber) + 1
        End If

        '=== Read chars
        If Size >= StreamLeft Then
            'This read will finish the rest of the file
            Result = Input(StreamLeft, m_FileNumber)
            CloseFile
        Else
            Result = Input(Size, m_FileNumber)
        End If
    Else
        Result = ""            'File closed!!!!
    End If

Case StreamTargetString
    StreamLeft = Len(m_Buffer)

    If Size > StreamLeft Then
        Result = m_Buffer      'Rest of Buffer
        m_Buffer = ""          'Kill rest of buffer
    Else
        Result = Left(m_Buffer, Size)
        'Rest of Buffer is beginning
        m_Buffer = Mid(m_Buffer, Size + 1)
    End If
End Select

StreamRead = Result
End Function

Public Function NextCharacter() As String
    'Get the next character in the stream, but DO NOT READ IT!
    Dim Result As String, CurrentPos As Long

```

```

Select Case m_StreamTarget
Case StreamTargetFile
    If m_FileOpen Then          'Read from file
        If VBA.EOF(m_FileNumber) Then
            Result = ""
        Else
            CurrentPos = Seek(m_FileNumber)

            Result = Input(1, m_FileNumber)
            Seek m_FileNumber, CurrentPos
            'Rewind 1 character
        End If
    Else
        Result = ""            'File closed!!!!
    End If

Case StreamTargetString
    Result = Left(m_Buffer, 1)
End Select

NextCharacter = Result
End Function

Property Get Text() As String

    If m_StreamTarget = StreamTargetString Then
        Text = m_Buffer
    Else
        Text = ""
    End If

End Property

Property Let Text(NewString As String)

    If m_StreamTarget = StreamTargetString Then
        m_Buffer = NewString
    End If

End Property

Public Sub WriteValue(ByVal Value As Variant)
    'This procedure writes a variable to the buffer/file.
    'If in binary mode, the buffer is flushed and the binary variable
    'is saved. If in text mode, the string equivalent of the
    'variable is added to the buffer

    Dim CharNumber As Integer, n As Integer

```



```

If m_StreamTransfer = StreamTransferText Then
    'Convert variable to string
    Value = CStr(Value)
End If

Select Case VarType(Value)
Case vbSingle, vbDouble
    'Not supported in the basic version
Case vbInteger
    WriteByteSequence EncodeInt16(Value)
Case vbLong
    WriteByteSequence EncodeInt32(Value)
Case vbByte
    StreamWrite Chr(Value)
Case vbString
    Select Case m_StringType
    Case StreamStringTypeASCII
        Select Case m_StringForm
        Case StreamStringFormNull
            StreamWrite CStr(Value) & Chr(0)
        Case StreamStringFormField
            WriteByteSequence EncodeInt16(Len(Value))
            '2-Byte length field
            StreamWrite CStr(Value)
        Case StreamStringFormNone
            StreamWrite CStr(Value)
        End Select
    Case StreamStringTypeUnicode
        Select Case m_StringForm
        Case StreamStringFormNull
            For n = 1 To Len(Value)
                CharNumber = AscW(Mid(Value, n, 1))
                WriteByteSequence EncodeInt16(CharNumber)
            Next
            WriteByteSequence EncodeInt16(0)
        Case StreamStringFormField
            WriteByteSequence EncodeInt16(Len(Value))
            '2-Byte length field
            For n = 1 To Len(Value)
                CharNumber = AscW(Mid(Value, n, 1))
                WriteByteSequence EncodeInt16(CharNumber)
            Next
        Case StreamStringFormNone
            For n = 1 To Len(Value)
                CharNumber = AscW(Mid(Value, n, 1))
                WriteByteSequence EncodeInt16(CharNumber)
            Next
        End Select
    End Select
End Select

```

```

End Sub

Private Sub Class_Initialize()

    m_Buffer = ""
    m_StreamTransfer = StreamTransferText
    m_StreamCanRead = True
    m_StreamCanWrite = True
    m_StreamTarget = StreamTargetString
    m_StringType = StreamStringTypeASCII
    m_StringForm = StreamStringFormNull
    m_Encoding = StreamEncodingLittleEndian
End Sub

Public Function ReadLine() As String

    Dim EndReached As Boolean, Text As String, ch As String

    If m_StreamCanRead Then
        EndReached = False
        Do Until EndReached Or Done()
            ch = StreamRead(1)
            If ch = Chr(10) Or ch = Chr(13) Then      'End char
                ch = NextCharacter()
                If ch = Chr(10) Or ch = Chr(13) Then  'Discard
second of line-feed, carriage return pair
                    StreamRead 1
                End If
                EndReached = True
            Else
                Text = Text & ch
            End If
        Loop
        ReadLine = Text
    Else
        ReadLine = ""
    End If

End Function

Private Sub Class_Terminate()

    If m_FileOpen Then
        CloseFile
    End If

End Sub

```

A.2.13. Symbol

This class is used to store of the nonterminals used by the Deterministic Finite Automata (DFA) and LALR Parser. Symbols can be either terminals (which represent a class of tokens - such as identifiers) or nonterminals (which represent the rules and structures of the grammar). Terminal symbols fall into several categories for use by the GOLD Parser Engine which are enumerated below.

```
'=====
' Class Name:
'     Symbol
'
' Instancing:
'     Public; Non-creatable  (VB Setting: 2- PublicNotCreatable)
'
' Purpose:
'     This class is used to store of the nonterminals used by the
'     Deterministic
'     Finite Automata (DFA) and LALR Parser. Symbols can be either
'     terminals (which represent a class of tokens - such as
identifiers) or
'     nonterminals (which represent the rules and structures of the
grammar).
'     Terminal symbols fall into several categories for use by the
GOLD
'     Parser Engine which are enumerated below.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     (None)
'
'=====

Option Explicit
```

```

Private pName As String
Private pKind As SymbolTypeConstants

Private pPattern As String
Private pVariableLength As Boolean

Public Enum SymbolTypeConstants
    SymbolTypeNonterminal = 0      'Normal nonterminal
    SymbolTypeTerminal = 1         'Normal terminal
    SymbolTypeWhitespace = 2       'Type of terminal
    SymbolTypeEnd = 3              'End character (EOF)
    SymbolTypeCommentStart = 4     'Comment start
    SymbolTypeCommentEnd = 5       'Comment end
    SymbolTypeCommentLine = 6     'Comment line
    SymbolTypeError = 7            'Error symbol
End Enum

Private pTableIndex As Integer

Private Const kQuotedChars = "|-+*?()[\]{}<>!"

Friend Property Let Kind(TheType As SymbolTypeConstants)

    pKind = TheType
End Property

Public Property Get Kind() As SymbolTypeConstants

    Kind = pKind
End Property

Friend Property Let TableIndex(Index As Integer)

    pTableIndex = Index
End Property

Public Property Get TableIndex() As Integer

    TableIndex = pTableIndex
End Property

Friend Property Let Name(NewName As String)

    pName = NewName

```

```

End Property

Public Property Get Name() As String

    Name = pName

End Property

Public Property Get Text() As String
    Dim str As String

    Select Case Kind
    Case SymbolTypeNonterminal
        str = "<" & Name & ">"
    Case SymbolTypeTerminal
        str = PatternFormat(Name)
    Case Else
        str = "(" & Name & ")"
    End Select

    Text = str
End Property

Private Function PatternFormat(Source As String) As String

    '=== Create a valid Regular Expression for a source string
    '=== Put all special characters in single quotes

    Dim c As Integer, Result As String, ch As String

    For c = 1 To Len(Source)
        ch = Mid(Source, c, 1)
        If ch = "'" Then
            ch = "''"
        ElseIf InStr(kQuotedChars, ch) <> 0 Or ch = Chr(34) Then
            ch = "'" & ch & "'"
        End If

        Result = Result & ch
    Next

    PatternFormat = Result
End Function

Private Sub Class_Initialize()
    pTableIndex = -1
End Sub

```

A.2.14. SymbolList

This class stores a list of Symbol objects.

```
'=====
' Class Name:
'     SymbolList
'
' Purpose:
'     This class is used to store a list of Symbols.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     Symbol class
'=====

Option Explicit

'This is a necessary class to organize symbols (given the Friend
property)

'Version 1.0

Private MemberList() As Symbol
Private MemberCount As Long

Public Sub ReDimension(ByVal NewCount As Long)

    MemberCount = NewCount
    If NewCount = 0 Then
        ReDim MemberList(0 To 0)
        Set MemberList(0) = Nothing
    Else
        ReDim Preserve MemberList(0 To MemberCount - 1)
    End If
End Sub
```

```
Public Sub Clear()  
    Erase MemberList  
    MemberCount = 0  
End Sub  
  
Public Function Count() As Long  
    Count = MemberCount  
End Function  
  
Property Get Member(ByVal Index As Long) As Symbol  
    If Index >= 0 And Index < MemberCount Then  
        Set Member = MemberList(Index)  
    Else  
        Set Member = Nothing  
    End If  
End Property  
  
Property Set Member(ByVal Index As Long, Obj As Symbol)  
    If Index >= 0 And Index < MemberCount Then  
        Set MemberList(Index) = Obj  
    End If  
End Property  
  
Public Function Add(NewItem As Symbol) As Long  
    MemberCount = MemberCount + 1  
    ReDim Preserve MemberList(0 To MemberCount - 1)  
    Set MemberList(MemberCount - 1) = NewItem  
    Add = MemberCount - 1  
End Function
```

```

Private Sub Class_Terminate()
    '== 12/20/2001: Added the set commands
    Dim n As Integer
    For n = 0 To MemberCount - 1
        Set MemberList(n) = Nothing
    Next
End Sub

```

A.2.15. Token

While the Symbol represents a class of terminals and nonterminals, the Token represents an individual piece of information. Ideally, the token would inherit directly from the Symbol Class, but do to the fact that Visual Basic 5/6 does not support this aspect of Object Oriented Programming, a Symbol is created as a member and its methods are mimicked.

```

'=====
' Class Name:
'     Token
'
' Instancing:
'     Public; Creatable (VB Setting: 5 - MultiUse)
'
' Purpose:
'     While the Symbol represents a class of terminals and nonterminals,
'     the Token represents an individual piece of information.
'     Ideally, the token would inherit directly from the Symbol Class,
'     but do to the fact that Visual Basic 5/6 does not support this
aspect
'     of Object Oriented Programming, a Symbol is created as a member
and
'     its methods are mimicked.
'
' Author(s):
'     Devin Cook
'     GOLDParse@DevinCook.com
'

```



```

' Dependencies:
'     Symbol class
'
'=====

Option Explicit

Private pState As Integer
Private pData As Variant
Private pParentSymbol As Symbol

Public Property Get Kind() As SymbolTypeConstants
    Kind = ParentSymbol.Kind
End Property

Public Property Get Name() As String
    Name = ParentSymbol.Name
End Property
Public Property Get ParentSymbol() As Symbol
    Set ParentSymbol = pParentSymbol
End Property

Public Property Get Data() As Variant
    'The ugliness of this is caused by that fact that Visual Basic 5/6
    treats
    'object different from scalar data types. This will be resolved in VB
    .NET

    If VarType(pData) = vbObject Then
        Set Data = pData
    Else
        Data = pData
    End If
End Property

Public Property Let Data(Value As Variant)
    pData = Value
End Property

Public Property Set Data(Value As Variant)
    Set pData = Value
End Property

Public Property Set ParentSymbol(TheSymbol As Symbol)
    Set pParentSymbol = TheSymbol

```

```
End Property

Public Property Get TableIndex() As Integer
    TableIndex = ParentSymbol.TableIndex
End Property

Public Property Get Text() As String
    Text = ParentSymbol.Text
End Property

Friend Property Let State(Value As Integer)
    pState = Value
End Property

Friend Property Get State() As Integer
    State = pState
End Property

Private Sub Class_Initialize()
    Data = Empty
End Sub
```

A.2.16. TokenStack

This class is used by the GOLDParse class to store tokens during parsing. In particular, this class is used in the LALR(1) state machine.

```
'=====
' Class Name:
'     TokenStack
'
' Instancing:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     This class is used by the GOLDParse class to store tokens
'     during parsing. In particular, this class is used in the LALR(1)
'     state machine.
'
' Author(s):
'     Devin Cook
'
' Dependencies:
'     Token Class
'
'=====
Option Explicit

Private MemberList() As Token
Private MemberCount As Long

Friend Property Let Count(Value As Long)

If Value < 1 Then
    Erase MemberList
    MemberCount = 0
Else
    ReDim Preserve MemberList(0 To Value - 1)      'Change the size of
the array
    MemberCount = Value
End If

End Property

Public Sub Clear()
```

```

    Erase MemberList
    MemberCount = 0

End Sub

Property Get Count() As Long

    Count = MemberCount

End Property

Property Get Member(ByVal Index As Long) As Token

    If Index >= 0 And Index < MemberCount Then
        Set Member = MemberList(Index)
    Else
        Set Member = Nothing
        ' MsgBox "Nothing!"
    End If

End Property

Property Set Member(ByVal Index As Long, TheToken As Token)

    If Index >= 0 And Index < MemberCount Then
        Set MemberList(Index) = TheToken
    End If

End Property

Public Sub Push(TheToken As Token)

    MemberCount = MemberCount + 1
    ReDim Preserve MemberList(0 To MemberCount - 1)    'Change the size
of the array
    Set MemberList(MemberCount - 1) = TheToken

End Sub

Public Function Pop() As Token

    'Modified 12/11/2001
    If MemberCount >= 1 Then
        Set Pop = MemberList(MemberCount - 1)    'Indexing from 0
    End If

End Function

```

```
        Set MemberList(MemberCount - 1) = Nothing    'List no longer
points to the token
        MemberCount = MemberCount - 1
    Else
        Set Pop = Nothing
    End If

End Function

Public Function Top() As Token

    If MemberCount >= 1 Then
        Set Top = MemberList(MemberCount - 1)
    End If

End Function
```

A.2.17. Variable

The variable class is a simple structure used to store the value and name of parameter.

```
Option Explicit

Public Name As String
Public Value As Variant
Public Comment As String
```

A.2.18. VariableList

This is a very simple class used to store a list of variables.

```
'=====
' Class Name:
'     VariableList
'
' Instancing:
'     Private; Internal  (VB Setting: 1 - Private)
'
' Purpose:
'     This is a very simple class that stores a list of "variables".
'     The GOLDParse class uses a this class to store the parameter
'     fields.
'
' Author(s):
'     Devin Cook
'     GOLDParse@DevinCook.com
'
' Dependencies:
'     (None)
'
'=====

Option Explicit
Option Compare Text
```

```

Private MemberList As New ObjectArray

Public Function Add(ByVal Name As String, Optional ByVal Value As String
= "", Optional ByVal Comment As String) As Boolean
    Dim Index As Integer, NewVar As New Variable

    Index = VariableIndex(Name)

    If Index = -1 Then
        NewVar.Name = Name
        NewVar.Value = Value
        NewVar.Comment = Comment

        MemberList.Add NewVar
    End If

    Add = (Index = -1)
End Function

Public Sub ClearValues()

    Dim n As Integer

    For n = 0 To MemberList.Count - 1
        MemberList.Member(n).Value = ""
    Next

End Sub

Public Function Count() As Integer
    Count = MemberList.Count
End Function

Public Function Member(ByVal ID As Variant) As Variable
    Dim Index As Integer, NewVar As Variable

    If VarType(ID) = VBString Then
        Index = VariableIndex(ID)
    Else
        Index = Val(ID)
    End If

    If Index >= 0 And Index < MemberList.Count Then
        Set Member = MemberList.Member(Index)
    Else
        Set NewVar = New Variable
        NewVar.Name = CStr(ID)

        MemberList.Add NewVar
        Set Member = NewVar
    End If
End Function

```

```
End If
End Function

Public Function VariableIndex(ByVal Name As String) As Integer
    Dim Index As Integer, n As Integer

    Index = -1
    n = 0
    Do While n < MemberList.Count And Index = -1
        If MemberList.Member(n).Name = Name Then
            Index = n
        Else
            n = n + 1
        End If
    Loop

    VariableIndex = Index
End Function
```


Appendix B. Example Program Template

The following is an example template that creates a skeleton Visual Basic program for the ActiveX version of the Engine.

```
##TEMPLATE-NAME 'Visual Basic - ActiveX (With Separate Procedure)'  
##LANGUAGE 'Visual Basic'  
##ENGINE-NAME 'ActiveX DLL'  
##AUTHOR 'Devin Cook'  
##FILE-EXTENSION 'bas'  
##NOTES  
Unlike the other template, this version defines separate procedures  
for handling reductions and loading the compiled grammar table file.  
  
This version is better for interpreters/compiler that use  
customized objects.  
##END-NOTES  
##ID-CASE Propercase  
##ID-SEPARATOR '_'  
##ID-SYMBOL-PREFIX 'Symbol'  
##ID-RULE-PREFIX 'Rule'  
Option Explicit  
  
Public Parser As New GOLDParseEngine.GOLDParse  
  
Public Enum SymbolConstants  
##SYMBOLS  
    %ID% = %Value% ' %Description%  
##END-SYMBOLS  
End Enum  
  
Public Enum RuleConstants  
##RULES  
    %ID% = %Value% ' %Description%  
##END-RULES  
End Enum  
  
public Sub DoParse(ByVal Source As String)  
    'This procedure starts the GOLD Parser Engine and handles each  
    'of the messages it returns. Each time a reduction is made,  
    'a new custom object can be created and used to store the rule.
```

```

'Otherwise, the system will use the Reduction object that was
'returned.
'
'The resulting tree will be a pure representation of the language
'and will be ready to implement.

Dim Response As GOLDParseEngine.GPMessageConstants
Dim Done As Boolean
Dim Success As Boolean      'Controls when we leave the loop

Success = False    'Unless the program is accepted by the parser

With Parser
    .OpenTextString Source
    .TrimReductions = True

    Done = False
    Do Until Done
        Response = .Parse()

        Select Case Response
            Case gpMsgLexicalError
                'Cannot recognize token
                Done = True

            Case gpMsgSyntaxError
                'Expecting a different token
                Done = True

            Case gpMsgReduction
                'Create a customized object to store the reduction
                Set .CurrentReduction = _
                    CreateNewObject(.CurrentReduction)

            Case gpMsgAccept
                'Success!
                'Set Program = .CurrentReduction    'The root node!
                Done = True
                Success = True

            Case gpMsgTokenRead
                'You don't have to do anything here.

            Case gpMsgInternalError
                'INTERNAL ERROR! Something is horribly wrong.
                Done = True

            Case gpMsgNotLoadedError
                'Due to the if-statement above, this case
                'statement should never be true
                Done = True

```

```

        Case gpMsgCommentError
            'COMMENT ERROR! Unexpected end of file
            Done = True
        End Select
    Loop
End With

End Sub

Public Function CreateNewObject _
    (TheReduction as GOLDParseEngine.Reduction) As Object

    Dim Result As Object

    With TheReduction
        Select Case .ParentRule.TableIndex
##RULES
            Case %ID%
                '%Description%'
##END-RULES
        End Select
    End With

    Set CreateNewObject = Result
End Function

Public Sub Setup
    Parser.LoadCompiledGrammar(App.Path & "\grammar.cgt")
End Sub

```

Appendix C. GOLD Meta-Grammar

C.1. Introduction

This is the very simple meta-grammar used to define GOLD Meta-Language. The terminal definitions are very complex. Each definition allows for an "Override Sequence" such as the backslash that is used in C. In this case, it is single quotes. Not all the terminals have overrides. They were only added where their use could be justified.

C.2. Grammar

```
"Name"      = 'GOLD Parser - Grammar Outline'
"Version"    = '2.2'
"Author"     = 'Devin Cook'
"About"      = 'This is the very simple grammar for defining grammars'

! All Printable characters in the first 256 of Unicode
{Printable Full} = {Printable} + {Printable Extended}

! The token definitions are very complex. Each definition allows
! for an "Override Sequence" such as the backslash in C. In this
! case, it is single quotes. Not all the tokens have overrides. I
! only added them where their use could be justified.

{Parameter Ch}  = {Printable}      - ["] - ['']
{Set Name Ch}   = {Printable}      - [{}]
{Literal Ch}    = {Printable Full} - ['']
{Nonterminal Ch} = {Printable}     - [<>] - ['']
```

```

{Terminal Ch}      = {Alphanumeric} + [_.]
{Set Literal Ch} = {Printable Full} - ['[ '' ]'] - ['']

ParameterName = ''' {Parameter Ch}+ '''
SetName       = '{' {Set Name Ch}+ '}'
Nonterminal   = '<' {Nonterminal Ch}+ '>'
Terminal      = ( {Terminal Ch} | ' ' {Literal Ch}* ' ' )+
SetLiteral     = '[' ({Set Literal Ch} | ' ' {Literal Ch}* ' ' )+ ']'

! This is a line based grammar
{Whitespace Ch} = {Whitespace} - {CR} - {LF}

Whitespace = {Whitespace Ch}+
Newline    = {CR}{LF} | {CR} | {LF}

Comment Line = '!'
Comment Start = '!*'
Comment End   = '*!'

"Start Symbol" = <Grammar>

! ----- Basics
! The <nl opt> here removes all newlines before the
! first definition

<Grammar> ::= <nl opt> <Content>

<Content> ::= <Definition> <Content>
           | <Definition>

<Definition> ::= <Parameter>
               | <Set Decl>
               | <Terminal Decl>
               | <Rule Decl>

! Optional series of New Line - use below is restricted
<nl opt> ::= NewLine <nl opt>
           |

! One or more New Lines
<nl> ::= NewLine <nl>
       | NewLine

! ----- Parameter Definition

<Parameter> ::= ParameterName <nl opt> '=' <Parameter Body>

```

```

<Parameter Body> ::= <Parameter Items> <nl opt> '|' <Parameter Body>
                    | <Parameter Items> <nl>

<Parameter Items> ::= <Parameter Item> <Parameter Items>
                    | <Parameter Item>

<Parameter Item> ::= ParameterName
                    | Terminal
                    | SetLiteral
                    | SetName
                    | Nonterminal

! ----- Set Definition

<Set Decl> ::= SetName <nl opt> '=' <Set Body>

<Set Body> ::= <Set Item> <nl opt> '+' <Set Body>
              | <Set Item> <nl opt> '-' <Set Body>
              | <Set Item> <nl>

<Set Item> ::= SetLiteral
             | SetName

! ----- Terminal Definition

<Terminal Decl> ::= <Terminal Name> <nl opt> '=' <Terminal Body>

<Terminal Name> ::= Terminal <Terminal Name>
                 | Terminal

<Terminal Body> ::= <Expression> <nl opt> '|' <Terminal Body>
                  | <Expression> <nl>

<Expression> ::= <Expression Item> <Expression>
                | <Expression Item>

<Expression Item> ::= Terminal <Kleene Opt>
                   | SetName <Kleene Opt>
                   | SetLiteral <Kleene Opt>
                   | '(' <Terminal Body 2> ')' <Kleene Opt>

<Terminal Body 2> ::= <Expression> '|' <Terminal Body 2>
                   | <Expression>

<Kleene Opt> ::= '+'
               | '?'

```

```

      | '*'
      |

! ----- Rule Definition

! Note the third rule of <Rule Body>. A rule is nullable if a blank
! entry exists. This entry can only be last in the list.

<Rule Decl> ::= Nonterminal <nl opt> '::=' <Rule Body>

<Rule Body> ::= <Production> <nl opt> '|' <Rule Body>
               | <Production> <nl>

<Production> ::= <Symbols>
                |

<Symbols>    ::= <Symbol> <Symbols>
               | <Symbol>

<Symbol>    ::= Terminal
               | Nonterminal

```


Appendix D. LALR Construction Test

D.1. Introduction

This appendix displays the tables created by both GOLD and YACC for the ANSI C grammar. To make sure that the results were directly comparable, the ANSI C grammar was defined using both the YACC and GOLD meta-languages. Both versions of the grammar contain the same number of rules. Each rule is also identical in both format and order.

Since the rules are listed side-by-side in this appendix, the width, in characters, of each can be problematic. If the rule is too long, the text will wrap and the resulting information would be hard to interpret. In addition, this would be costly on the number of pages required for printing. To help alleviate this problem, the actual names used for terminals and nonterminals were intentionally shortened. Unfortunately, the text will still have to be wrapped in many cases.

In the table displayed in Appendix D.3, the rules were "smart wrapped" by a throw-away utility program. If a rule reaches the end of the column, the text is wrapped and continued where the body of the rule started. This allows the text to be easily interpreted. To save additional space, the indentation from the right column was decreased to 2 characters. Other than these two modifications, the LALR state table exported from YACC and GOLD are unchanged.

There is some notational differences which are important to understand to interpret the results. The YACC export text follows each configuration (a rule and a cursor position) by the rule's index delimited by parenthesis. The GOLD export, on the other hand, does not display this information. While helpful, this information does not change the interpretation of the tables.

In many cases, a state contains a single complete configuration. In other words, the configuration has no remaining terminals or nonterminals to be read. In this case, the only action that can be performed is a reduction.

While the GOLD table export displays each of the valid terminals that the system expects to see, YACC tersely displays a period followed by the reduce action. This is interpreted literally as "any terminal". Whether this is simply a result of the notation or whether YACC stores an "any terminal" entry in the LALR tables is unknown. However, given how states are combined during LALR table construction, this is irrelevant. The expected tokens for these "single reduce" states will be supersets of the expected tokens of any state the system will "go to". In other words, any parse errors caught on these "single reduce" states will be caught when the next token is read. As a result, YACC's use of the "any terminal" in the exported tables is inconsequential for comparison of correctness.

D.2. ANSI C Grammar in YACC

The following is the grammar for ANSI C defined using the YACC meta-language. The GOLD version of this grammar can be located in Appendix E.

```
%token MINUS MINUSMINUS EXCLAM EXCLAMEQ
%token PERCENT AMP AMPAMP AMPEQ LPARAN RPARAN
%token TIMES TIMESEQ COMMA DOT DIV DIVEQ COLON
%token SEMI QUESTION LBRACKET RBRACKET CARET
%token CARETEQ LBRACE PIPE PIPEPIPE PIPEEQ RBRACE
%token TILDE PLUS PLUSPLUS PLUSEQ LT LTLT LTEQ EQ
%token MINUSEQ EQEQ GT MINUSGT GTEQ GTGT AUTO
%token BREAK CASE CHAR CHARLITERAL CONST CONTINUE
%token DECLITERAL DEFAULT DO DOUBLE ELSE ENUM
%token EXTERN FLOAT FLOATLITERAL FOR GOTO HEXLITERAL
%token ID IF INT LONG OCTLITERAL REGISTER RETURN
%token SHORT SIGNED SIZEOF STATIC STRINGLITERAL
%token STRUCT SWITCH TYPEDEF UNION UNSIGNED
%token VOID VOLATILE WHILE

%%

decls : decl decls
      |
      ;

decl : func_decl
      | func_proto
      | struct_decl
      | union_decl
      | enum_decl
      | var_decl
      | typedef_decl
      ;

func_proto : func_id LPARAN types RPARAN SEMI
            | func_id LPARAN params RPARAN SEMI
            | func_id LPARAN RPARAN SEMI
            ;

func_decl : func_id LPARAN params RPARAN block
```

```

        | func_id LPARAN id_list RPARAN struct_def block
        | func_id LPARAN RPARAN block
    ;

params : param COMMA params
    | param
    ;

param : CONST type ID
    | type ID
    ;

types : type COMMA types
    | type
    ;

id_list : ID COMMA id_list
    | ID
    ;

func_id : type ID
    | ID
    ;

typedef_decl : TYPEDEF type ID SEMI
    ;

struct_decl : STRUCT ID LBRACE struct_def RBRACE SEMI
    ;

union_decl : UNION ID LBRACE struct_def RBRACE SEMI
    ;

struct_def : var_decl struct_def
    | var_decl
    ;

var_decl : mod type var var_list SEMI
    | type var var_list SEMI
    | mod var var_list SEMI
    ;

var : ID array
    | ID array EQ op_if
    ;

array : LBRACKET expr RBRACKET
    | LBRACKET RBRACKET
    ;

```

```

var_list : COMMA var_item var_list
        |
        ;

var_item : pointers var
        ;

mod : EXTERN
    | STATIC
    | REGISTER
    | AUTO
    | VOLATILE
    | CONST
    ;

enum_decl : ENUM ID LBRACE enum_def RBRACE SEMI
          ;

enum_def : enum_val COMMA enum_def
         | enum_val
         ;

enum_val : ID
         | ID EQ OCTLITERAL
         | ID EQ HEXLITERAL
         | ID EQ DECLITERAL
         ;

type : base pointers
     ;

base : sign scalar
     | STRUCT ID
     | STRUCT LBRACE struct_def RBRACE
     | UNION ID
     | UNION LBRACE struct_def RBRACE
     | ENUM ID
     ;

sign : SIGNED
     | UNSIGNED
     ;

scalar : CHAR
       | INT
       | SHORT
       | LONG
       | SHORT INT
       | LONG INT
       | FLOAT

```

```

        | DOUBLE
        | VOID
        ;

pointers : TIMES pointers
        |
        ;

stm : var_decl
    | ID COLON
    | IF LPARAN expr RPARAN stm
    | IF LPARAN expr RPARAN then_stm ELSE stm
    | WHILE LPARAN expr RPARAN stm
    | FOR LPARAN arg SEMI arg SEMI arg RPARAN stm
    | normal_stm
    ;

then_stm : IF LPARAN expr RPARAN then_stm ELSE then_stm
        | WHILE LPARAN expr RPARAN then_stm
        | FOR LPARAN arg SEMI arg SEMI arg RPARAN then_stm
        | normal_stm
        ;

normal_stm : DO stm WHILE LPARAN expr RPARAN
           | SWITCH LPARAN expr RPARAN LBRACE case_stms RBRACE
           | block
           | expr SEMI
           | GOTO ID SEMI
           | BREAK SEMI
           | CONTINUE SEMI
           | RETURN expr SEMI
           | SEMI
           ;

arg : expr
    |
    ;

case_stms : CASE value COLON stm_list case_stms
          | DEFAULT COLON stm_list
          ;

block : LBRACE stm_list RBRACE
      ;

stm_list : stm stm_list
         |
         ;

expr : expr COMMA op_assign

```

```

    | op_assign
    ;

op_assign : op_if EQ op_assign
    | op_if PLUSEQ op_assign
    | op_if MINUSEQ op_assign
    | op_if TIMESEQ op_assign
    | op_if DIVEQ op_assign
    | op_if CARETEQ op_assign
    | op_if AMPEQ op_assign
    | op_if PIPEEQ op_assign
    | op_if GTGTEQ op_assign
    | op_if LTLTEQ op_assign
    | op_if
    ;

op_if : op_or QUESTION op_if COLON op_if
    | op_or
    ;

op_or : op_or PIPEPIPE op_and
    | op_and
    ;

op_and : op_and AMPAMP op_binor
    | op_binor
    ;

op_binor : op_binor PIPE op_binxor
    | op_binxor
    ;

op_binxor : op_binxor CARET op_binand
    | op_binand
    ;

op_binand : op_binand AMP op_equate
    | op_equate
    ;

op_equate : op_equate EQEQ op_compare
    | op_equate EXCLAMEQ op_compare
    | op_compare
    ;

op_compare : op_compare LT op_shift
    | op_compare GT op_shift
    | op_compare LTEQ op_shift
    | op_compare GTEQ op_shift
    | op_shift
    ;

```

```

op_shift : op_shift LTLT op_add
          | op_shift GTGT op_add
          | op_add
          ;

op_add : op_add PLUS op_mult
        | op_add MINUS op_mult
        | op_mult
        ;

op_mult : op_mult TIMES op_unary
          | op_mult DIV op_unary
          | op_mult PERCENT op_unary
          | op_unary
          ;

op_unary : EXCLAM op_unary
           | TILDE op_unary
           | MINUS op_unary
           | TIMES op_unary
           | AMP op_unary
           | PLUSPLUS op_unary
           | MINUSMINUS op_unary
           | op_pointer PLUSPLUS
           | op_pointer MINUSMINUS
           | LPARAN type RPARAN op_unary
           | SIZEOF LPARAN type RPARAN
           | SIZEOF LPARAN ID pointers RPARAN
           | op_pointer
           ;

op_pointer : op_pointer DOT value
            | op_pointer MINUSGT value
            | op_pointer LBRACKET expr RBRACKET
            | value
            ;

value : OCTLITERAL
       | HEXLITERAL
       | DECLITERAL
       | STRINGLITERAL
       | CHARLITERAL
       | FLOATLITERAL
       | ID LPARAN expr RPARAN
       | ID LPARAN RPARAN
       | ID
       | LPARAN expr RPARAN
       ;

```


D.3. Side-By-Side Comparison

In the table below, the order of the YACC states was left unchanged. The corresponding GOLD states were reordered so that equivalent states would be displayed on the same row.

<pre> state 0 \$accept : . decls \$end (0) decls : . (2) sign : . (64) AUTO shift 1 CONST shift 2 ENUM shift 3 EXTERN shift 4 ID shift 5 REGISTER shift 6 SIGNED shift 7 STATIC shift 8 STRUCT shift 9 TYPEDEF shift 10 UNION shift 11 UNSIGNED shift 12 VOLATILE shift 13 \$end reduce 2 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 decls goto 14 decl goto 15 func_decl goto 16 func_proto goto 17 struct_decl goto 18 union_decl goto 19 enum_decl goto 20 var_decl goto 21 typedef_decl goto 22 func_id goto 23 type goto 24 mod goto 25 base goto 26 sign goto 27 </pre>	<pre> State 0 <S'> ::= • <Decls> (EOF) <Decls> ::= • <Sign> ::= • auto Shift 1 const Shift 2 enum Shift 3 extern Shift 17 Id Shift 18 register Shift 19 signed Shift 20 static Shift 21 struct Shift 22 typedef Shift 209 union Shift 213 unsigned Shift 30 volatile Shift 31 (EOF) Reduce 1 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Decl> Goto 219 <Decls> Goto 350 <Enum Decl> Goto 221 <Func Decl> Goto 222 <Func ID> Goto 223 <Func Proto> Goto 343 <Mod> Goto 36 <Sign> Goto 60 <Struct Decl> Goto 344 <Type> Goto 345 <Typedef Decl> Goto 347 <Union Decl> Goto 348 <Var Decl> Goto 349 </pre>
<pre> state 1 mod : AUTO . (45) . reduce 45 </pre>	<pre> State 1 <Mod> ::= auto • char Reduce 44 double Reduce 44 enum Reduce 44 </pre>

	float Reduce 44 Id Reduce 44 int Reduce 44 long Reduce 44 short Reduce 44 signed Reduce 44 struct Reduce 44 union Reduce 44 unsigned Reduce 44 void Reduce 44
state 2 mod : CONST . (47) . reduce 47	State 2 <Mod> ::= const • char Reduce 46 double Reduce 46 enum Reduce 46 float Reduce 46 Id Reduce 46 int Reduce 46 long Reduce 46 short Reduce 46 signed Reduce 46 struct Reduce 46 union Reduce 46 unsigned Reduce 46 void Reduce 46
state 3 enum_decl : ENUM . ID LBRACE enum_def RBRACE SEMI (48) base : ENUM . ID (61) ID shift 28 . error	State 3 <Enum Decl> ::= enum • Id '{' <Enum Def> '}' ';' ; <Base> ::= enum • Id Id Shift 4
state 4 mod : EXTERN . (42) . reduce 42	State 17 <Mod> ::= extern • char Reduce 41 double Reduce 41 enum Reduce 41 float Reduce 41 Id Reduce 41 int Reduce 41 long Reduce 41 short Reduce 41 signed Reduce 41 struct Reduce 41 union Reduce 41 unsigned Reduce 41 void Reduce 41
state 5 func_id : ID . (25) . reduce 25	State 18 <Func ID> ::= Id • '(' Reduce 24
state 6 mod : REGISTER . (44) . reduce 44	State 19 <Mod> ::= register • char Reduce 43 double Reduce 43 enum Reduce 43 float Reduce 43 Id Reduce 43 int Reduce 43 long Reduce 43

	short Reduce 43 signed Reduce 43 struct Reduce 43 union Reduce 43 unsigned Reduce 43 void Reduce 43
state 7 sign : SIGNED . (62) . reduce 62	State 20 <Sign> ::= signed • char Reduce 61 double Reduce 61 float Reduce 61 int Reduce 61 long Reduce 61 short Reduce 61 void Reduce 61
state 8 mod : STATIC . (43) . reduce 43	State 21 <Mod> ::= static • char Reduce 42 double Reduce 42 enum Reduce 42 float Reduce 42 Id Reduce 42 int Reduce 42 long Reduce 42 short Reduce 42 signed Reduce 42 struct Reduce 42 union Reduce 42 unsigned Reduce 42 void Reduce 42
state 9 struct_decl : STRUCT . ID LBRACE struct_def RBRACE SEMI (27) base : STRUCT . ID (57) base : STRUCT . LBRACE struct_def RBRACE (58) LBRACE shift 29 ID shift 30 . error	State 22 <Struct Decl> ::= struct • Id '{' <Struct Def> '}' ';' ; <Base> ::= struct • Id <Base> ::= struct • '{' <Struct Def> '}' '{' Shift 23 Id Shift 204
state 10 typedef_decl : TYPEDEF . type ID SEMI (26) sign : . (64) ENUM shift 31 SIGNED shift 7 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 type goto 34 base goto 26	State 209 <Typedef Decl> ::= typedef • <Type> Id ';' <Sign> ::= • enum Shift 24 signed Shift 20 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Sign> Goto 60

sign goto 27	<Type> Goto 210
state 11 union_decl : UNION . ID LBRACE struct_def RBRACE SEMI (28) base : UNION . ID (59) base : UNION . LBRACE struct_def RBRACE (60) LBRACE shift 35 ID shift 36 . error	State 213 <Union Decl> ::= union • Id '{' <Struct Def> '}' ';' ; <Base> ::= union • Id <Base> ::= union • '{' <Struct Def> '}' '{' Shift 29 Id Shift 214
state 12 sign : UNSIGNED . (63) . reduce 63	State 30 <Sign> ::= unsigned • char Reduce 62 double Reduce 62 float Reduce 62 int Reduce 62 long Reduce 62 short Reduce 62 void Reduce 62
state 13 mod : VOLATILE . (46) . reduce 46	State 31 <Mod> ::= volatile • char Reduce 45 double Reduce 45 enum Reduce 45 float Reduce 45 Id Reduce 45 int Reduce 45 long Reduce 45 short Reduce 45 signed Reduce 45 struct Reduce 45 union Reduce 45 unsigned Reduce 45 void Reduce 45
state 14 \$accept : decls . \$end (0) \$end accept	State 350 <S'> ::= <Decls> • (EOF) (EOF) Accept
state 15 decls : decl . decls (1) decls : . (2) sign : . (64) AUTO shift 1 CONST shift 2 ENUM shift 3 EXTERN shift 4 ID shift 5 REGISTER shift 6 SIGNED shift 7 STATIC shift 8 STRUCT shift 9 TYPEDEF shift 10 UNION shift 11 UNSIGNED shift 12 VOLATILE shift 13 \$end reduce 2 CHAR reduce 64 DOUBLE reduce 64	State 219 <Decls> ::= <Decl> • <Decls> <Decls> ::= • <Sign> ::= • auto Shift 1 const Shift 2 enum Shift 3 extern Shift 17 Id Shift 18 register Shift 19 signed Shift 20 static Shift 21 struct Shift 22 typedef Shift 209 union Shift 213 unsigned Shift 30 volatile Shift 31 (EOF) Reduce 1 char Reduce 63 double Reduce 63

<pre> FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 decls goto 37 decl goto 15 func_decl goto 16 func_proto goto 17 struct_decl goto 18 union_decl goto 19 enum_decl goto 20 var_decl goto 21 typedef_decl goto 22 func_id goto 23 type goto 24 mod goto 25 base goto 26 sign goto 27 </pre>	<pre> float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Decl> Goto 219 <Decls> Goto 220 <Enum Decl> Goto 221 <Func Decl> Goto 222 <Func ID> Goto 223 <Func Proto> Goto 343 <Mod> Goto 36 <Sign> Goto 60 <Struct Decl> Goto 344 <Type> Goto 345 <Typedef Decl> Goto 347 <Union Decl> Goto 348 <Var Decl> Goto 349 </pre>
<pre> state 16 decl : func_decl . (3) . reduce 3 </pre>	<pre> State 222 <Decl> ::= <Func Decl> • (EOF) Reduce 2 auto Reduce 2 char Reduce 2 const Reduce 2 double Reduce 2 enum Reduce 2 extern Reduce 2 float Reduce 2 Id Reduce 2 int Reduce 2 long Reduce 2 register Reduce 2 short Reduce 2 signed Reduce 2 static Reduce 2 struct Reduce 2 typedef Reduce 2 union Reduce 2 unsigned Reduce 2 void Reduce 2 volatile Reduce 2 </pre>
<pre> state 17 decl : func_proto . (4) . reduce 4 </pre>	<pre> State 343 <Decl> ::= <Func Proto> • (EOF) Reduce 3 auto Reduce 3 char Reduce 3 const Reduce 3 double Reduce 3 enum Reduce 3 extern Reduce 3 float Reduce 3 Id Reduce 3 int Reduce 3 long Reduce 3 register Reduce 3 short Reduce 3 signed Reduce 3 static Reduce 3 struct Reduce 3 typedef Reduce 3 </pre>

	union Reduce 3 unsigned Reduce 3 void Reduce 3 volatile Reduce 3
state 18 decl : struct_decl . (5) . reduce 5	State 344 <Decl> ::= <Struct Decl> • (EOF) Reduce 4 auto Reduce 4 char Reduce 4 const Reduce 4 double Reduce 4 enum Reduce 4 extern Reduce 4 float Reduce 4 Id Reduce 4 int Reduce 4 long Reduce 4 register Reduce 4 short Reduce 4 signed Reduce 4 static Reduce 4 struct Reduce 4 typedef Reduce 4 union Reduce 4 unsigned Reduce 4 void Reduce 4 volatile Reduce 4
state 19 decl : union_decl . (6) . reduce 6	State 348 <Decl> ::= <Union Decl> • (EOF) Reduce 5 auto Reduce 5 char Reduce 5 const Reduce 5 double Reduce 5 enum Reduce 5 extern Reduce 5 float Reduce 5 Id Reduce 5 int Reduce 5 long Reduce 5 register Reduce 5 short Reduce 5 signed Reduce 5 static Reduce 5 struct Reduce 5 typedef Reduce 5 union Reduce 5 unsigned Reduce 5 void Reduce 5 volatile Reduce 5
state 20 decl : enum_decl . (7) . reduce 7	State 221 <Decl> ::= <Enum Decl> • (EOF) Reduce 6 auto Reduce 6 char Reduce 6 const Reduce 6 double Reduce 6 enum Reduce 6 extern Reduce 6 float Reduce 6

	Id Reduce 6 int Reduce 6 long Reduce 6 register Reduce 6 short Reduce 6 signed Reduce 6 static Reduce 6 struct Reduce 6 typedef Reduce 6 union Reduce 6 unsigned Reduce 6 void Reduce 6 volatile Reduce 6
state 21 decl : var_decl . (8) . reduce 8	State 349 <Decl> ::= <Var Decl> • (EOF) Reduce 7 auto Reduce 7 char Reduce 7 const Reduce 7 double Reduce 7 enum Reduce 7 extern Reduce 7 float Reduce 7 Id Reduce 7 int Reduce 7 long Reduce 7 register Reduce 7 short Reduce 7 signed Reduce 7 static Reduce 7 struct Reduce 7 typedef Reduce 7 union Reduce 7 unsigned Reduce 7 void Reduce 7 volatile Reduce 7
state 22 decl : typedef_decl . (9) . reduce 9	State 347 <Decl> ::= <Typedef Decl> • (EOF) Reduce 8 auto Reduce 8 char Reduce 8 const Reduce 8 double Reduce 8 enum Reduce 8 extern Reduce 8 float Reduce 8 Id Reduce 8 int Reduce 8 long Reduce 8 register Reduce 8 short Reduce 8 signed Reduce 8 static Reduce 8 struct Reduce 8 typedef Reduce 8 union Reduce 8 unsigned Reduce 8 void Reduce 8 volatile Reduce 8
state 23 func_proto : func_id . LPARAN types	State 223 <Func Proto> ::= <Func ID> • '('

<pre> RPARAN SEMI (10) func_proto : func_id . LPARAN params RPARAN SEMI (11) func_proto : func_id . LPARAN RPARAN SEMI (12) func_decl : func_id . LPARAN params RPARAN block (13) func_decl : func_id . LPARAN id_list RPARAN struct_def block (14) func_decl : func_id . LPARAN RPARAN block (15) LPARAN shift 38 . error </pre>	<pre> <Types> ')' ';' <Func Proto> ::= <Func ID> • '(' <Params> ')' ';' <Func Proto> ::= <Func ID> • '(' ')' ';' <Func Decl> ::= <Func ID> • '(' <Params> ')' <Block> <Func Decl> ::= <Func ID> • '(' <Id List> ')' <Struct Def> <Block> <Func Decl> ::= <Func ID> • '(' ')' <Block> '(' Shift 224 </pre>
<pre> state 24 func_id : type . ID (24) var_decl : type . var var_list SEMI (32) ID shift 39 . error var goto 40 </pre>	<pre> State 345 <Func ID> ::= <Type> • Id <Var Decl> ::= <Type> • <Var> <Var List> ';' Id Shift 346 <Var> Goto 196 </pre>
<pre> state 25 var_decl : mod . type var var_list SEMI (31) var_decl : mod . var var_list SEMI (33) sign : . (64) ENUM shift 31 ID shift 41 SIGNED shift 7 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 type goto 42 var goto 43 base goto 26 sign goto 27 </pre>	<pre> State 36 <Var Decl> ::= <Mod> • <Type> <Var> <Var List> ';' <Var Decl> ::= <Mod> • <Var> <Var List> ';' <Sign> ::= • enum Shift 24 Id Shift 37 signed Shift 20 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Sign> Goto 60 <Type> Goto 181 <Var> Goto 190 </pre>
<pre> state 26 type : base . pointers (55) pointers : . (75) TIMES shift 44 RPARAN reduce 75 COMMA reduce 75 ID reduce 75 pointers goto 45 </pre>	<pre> State 32 <Type> ::= <Base> • <Pointers> <Pointers> ::= • '*' Shift 33 ')' Reduce 74 ',' Reduce 74 Id Reduce 74 <Pointers> Goto 35 </pre>
<pre> state 27 base : sign . scalar (56) CHAR shift 46 </pre>	<pre> State 60 <Base> ::= <Sign> • <Scalar> char Shift 61 </pre>

DOUBLE shift 47 FLOAT shift 48 INT shift 49 LONG shift 50 SHORT shift 51 VOID shift 52 . error scalar goto 53	double Shift 62 float Shift 63 int Shift 64 long Shift 65 short Shift 67 void Shift 69 <Scalar> Goto 70
state 28 enum_decl : ENUM ID . LBRACE enum_def RBRACE SEMI (48) base : ENUM ID . (61) LBRACE shift 54 TIMES reduce 61 ID reduce 61	State 4 <Enum Decl> ::= enum Id • '{' <Enum Def> '}' ';' ; <Base> ::= enum Id • '{' Shift 5 '*' Reduce 60 Id Reduce 60
state 29 base : STRUCT LBRACE . struct_def RBRACE (58) sign : . (64) AUTO shift 1 CONST shift 2 ENUM shift 31 EXTERN shift 4 REGISTER shift 6 SIGNED shift 7 STATIC shift 8 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 55 struct_def goto 56 type goto 57 mod goto 25 base goto 26 sign goto 27	State 23 <Base> ::= struct '{' • <Struct Def> '}' <Sign> ::= • auto Shift 1 const Shift 2 enum Shift 24 extern Shift 17 register Shift 19 signed Shift 20 static Shift 21 struct Shift 26 union Shift 28 unsigned Shift 30 volatile Shift 31 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Mod> Goto 36 <Sign> Goto 60 <Struct Def> Goto 202 <Type> Goto 195 <Var Decl> Goto 199
state 30 struct_decl : STRUCT ID . LBRACE struct_def RBRACE SEMI (27) base : STRUCT ID . (57) LBRACE shift 58 TIMES reduce 57 ID reduce 57	State 204 <Struct Decl> ::= struct Id • '{' <Struct Def> '}' ';' ; <Base> ::= struct Id • '{' Shift 205 '*' Reduce 56 Id Reduce 56
state 31 base : ENUM . ID (61) ID shift 59 . error	State 24 <Base> ::= enum • Id Id Shift 25
state 32 base : STRUCT . ID (57)	State 26 <Base> ::= struct • Id

<pre> base : STRUCT . LBRACE struct_def RBRACE (58) LBRACE shift 29 ID shift 60 . error </pre>	<pre> <Base> ::= struct • '{' <Struct Def> '}' '{' Shift 23 Id Shift 27 </pre>
<pre> state 33 base : UNION . ID (59) base : UNION . LBRACE struct_def RBRACE (60) LBRACE shift 35 ID shift 61 . error </pre>	<pre> State 28 <Base> ::= union • Id <Base> ::= union • '{' <Struct Def> '}' '{' Shift 29 Id Shift 201 </pre>
<pre> state 34 typedef_decl : TYPEDEF type . ID SEMI (26) ID shift 62 . error </pre>	<pre> State 210 <Typedef Decl> ::= typedef <Type> • Id ';' Id Shift 211 </pre>
<pre> state 35 base : UNION LBRACE . struct_def RBRACE (60) sign : . (64) AUTO shift 1 CONST shift 2 ENUM shift 31 EXTERN shift 4 REGISTER shift 6 SIGNED shift 7 STATIC shift 8 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 55 struct_def goto 63 type goto 57 mod goto 25 base goto 26 sign goto 27 </pre>	<pre> State 29 <Base> ::= union '{' • <Struct Def> '}' <Sign> ::= • auto Shift 1 const Shift 2 enum Shift 24 extern Shift 17 register Shift 19 signed Shift 20 static Shift 21 struct Shift 26 union Shift 28 unsigned Shift 30 volatile Shift 31 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Mod> Goto 36 <Sign> Goto 60 <Struct Def> Goto 193 <Type> Goto 195 <Var Decl> Goto 199 </pre>
<pre> state 36 union_decl : UNION ID . LBRACE struct_def RBRACE SEMI (28) base : UNION ID . (59) LBRACE shift 64 TIMES reduce 59 ID reduce 59 </pre>	<pre> State 214 <Union Decl> ::= union Id • '{' <Struct Def> '}' ';' <Base> ::= union Id • '{' Shift 215 '*' Reduce 58 Id Reduce 58 </pre>
<pre> state 37 decls : decl decls . (1) . reduce 1 </pre>	<pre> State 220 <Decls> ::= <Decl> <Decls> • (Eof) Reduce 0 </pre>

<p>state 38</p> <p>func_proto : func_id LPARAN . types RPARAN SEMI (10)</p> <p>func_proto : func_id LPARAN . params RPARAN SEMI (11)</p> <p>func_proto : func_id LPARAN . RPARAN SEMI (12)</p> <p>func_decl : func_id LPARAN . params RPARAN block (13)</p> <p>func_decl : func_id LPARAN . id_list RPARAN struct_def block (14)</p> <p>func_decl : func_id LPARAN . RPARAN block (15)</p> <p>sign : . (64)</p> <p>RPARAN shift 65 CONST shift 66 ENUM shift 31 ID shift 67 SIGNED shift 7 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64</p> <p>types goto 68 params goto 69 id_list goto 70 param goto 71 type goto 72 base goto 26 sign goto 27</p>	<p>State 224</p> <p><Func Proto> ::= <Func ID> '(' • <Types> ')' ';' ;</p> <p><Func Proto> ::= <Func ID> '(' • <Params> ')' ';' ;</p> <p><Func Proto> ::= <Func ID> '(' • ')' ; ';' ;</p> <p><Func Decl> ::= <Func ID> '(' • <Params> ')' <Block></p> <p><Func Decl> ::= <Func ID> '(' • <Id List> ')' <Struct Def> <Block></p> <p><Func Decl> ::= <Func ID> '(' • ')' ; <Block></p> <p><Sign> ::= •</p> <p>')' Shift 225 const Shift 317 enum Shift 24 Id Shift 320 signed Shift 20 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63</p> <p><Base> Goto 32 <Id List> Goto 323 <Param> Goto 327 <Params> Goto 332 <Sign> Goto 60 <Type> Goto 336 <Types> Goto 340</p>
<p>state 39</p> <p>func_id : type ID . (24)</p> <p>var : ID . array (34)</p> <p>var : ID . array EQ op_if (35)</p> <p>array : . (38)</p> <p>LBRACKET shift 73 LPARAN reduce 24 COMMA reduce 38 SEMI reduce 38 EQ reduce 38</p> <p>array goto 74</p>	<p>State 346</p> <p><Func ID> ::= <Type> Id •</p> <p><Var> ::= Id • <Array></p> <p><Var> ::= Id • <Array> '=' <Op If></p> <p><Array> ::= •</p> <p>'[' Shift 38 '(' Reduce 23 ' ,' Reduce 37 ' ;' Reduce 37 ' =' Reduce 37</p> <p><Array> Goto 178</p>
<p>state 40</p> <p>var_decl : type var . var_list SEMI (32)</p> <p>var_list : . (40)</p> <p>COMMA shift 75 SEMI reduce 40</p> <p>var_list goto 76</p>	<p>State 196</p> <p><Var Decl> ::= <Type> <Var> • <Var List> ';' ;</p> <p><Var List> ::= •</p> <p>' ,' Shift 183 ' ;' Reduce 39</p> <p><Var List> Goto 197</p>
<p>state 41</p> <p>var : ID . array (34)</p>	<p>State 37</p> <p><Var> ::= Id • <Array></p>

var : ID . array EQ op_if (35) array : . (38) LBRACKET shift 73 COMMA reduce 38 SEMI reduce 38 EQ reduce 38 array goto 74	<Var> ::= Id • <Array> '=' <Op If> <Array> ::= • '[' Shift 38 ',' Reduce 37 ';' Reduce 37 '=' Reduce 37 <Array> Goto 178
state 42 var_decl : mod type . var var_list SEMI (31) ID shift 41 . error var goto 77	State 181 <Var Decl> ::= <Mod> <Type> • <Var> <Var List> ';' ; Id Shift 37 <Var> Goto 182
state 43 var_decl : mod var . var_list SEMI (33) var_list : . (40) COMMA shift 75 SEMI reduce 40 var_list goto 78	State 190 <Var Decl> ::= <Mod> <Var> • <Var List> ';' <Var List> ::= • ',' Shift 183 ';' Reduce 39 <Var List> Goto 191
state 44 pointers : TIMES . pointers (74) pointers : . (75) TIMES shift 44 RPARAN reduce 75 COMMA reduce 75 ID reduce 75 pointers goto 79	State 33 <Pointers> ::= '*' • <Pointers> <Pointers> ::= • '*' Shift 33 ')' Reduce 74 ',' Reduce 74 Id Reduce 74 <Pointers> Goto 34
state 45 type : base pointers . (55) . reduce 55	State 35 <Type> ::= <Base> <Pointers> • ')' Reduce 54 ',' Reduce 54 Id Reduce 54
state 46 scalar : CHAR . (65) . reduce 65	State 61 <Scalar> ::= char • ')' Reduce 64 '*' Reduce 64 ',' Reduce 64 Id Reduce 64
state 47 scalar : DOUBLE . (72) . reduce 72	State 62 <Scalar> ::= double • ')' Reduce 71 '*' Reduce 71 ',' Reduce 71 Id Reduce 71
state 48 scalar : FLOAT . (71) . reduce 71	State 63 <Scalar> ::= float • ')' Reduce 70 '*' Reduce 70 ',' Reduce 70

	Id Reduce 70
state 49 scalar : INT . (66) . reduce 66	State 64 <Scalar> ::= int •)' Reduce 65 '* Reduce 65 ' , ' Reduce 65 Id Reduce 65
state 50 scalar : LONG . (68) scalar : LONG . INT (70) INT shift 80 RPARAN reduce 68 TIMES reduce 68 COMMA reduce 68 ID reduce 68	State 65 <Scalar> ::= long • <Scalar> ::= long • int int Shift 66)' Reduce 67 '* Reduce 67 ' , ' Reduce 67 Id Reduce 67
state 51 scalar : SHORT . (67) scalar : SHORT . INT (69) INT shift 81 RPARAN reduce 67 TIMES reduce 67 COMMA reduce 67 ID reduce 67	State 67 <Scalar> ::= short • <Scalar> ::= short • int int Shift 68)' Reduce 66 '* Reduce 66 ' , ' Reduce 66 Id Reduce 66
state 52 scalar : VOID . (73) . reduce 73	State 69 <Scalar> ::= void •)' Reduce 72 '* Reduce 72 ' , ' Reduce 72 Id Reduce 72
state 53 base : sign scalar . (56) . reduce 56	State 70 <Base> ::= <Sign> <Scalar> •)' Reduce 55 '* Reduce 55 ' , ' Reduce 55 Id Reduce 55
state 54 enum_decl : ENUM ID LBRACE . enum_def RBRACE SEMI (48) ID shift 82 . error enum_def goto 83 enum_val goto 84	State 5 <Enum Decl> ::= enum Id '{' • <Enum Def> '}' ';' ; Id Shift 6 <Enum Def> Goto 11 <Enum Val> Goto 14
state 55 struct_def : var_decl . struct_def (29) struct_def : var_decl . (30) sign : . (64) AUTO shift 1 CONST shift 2 ENUM shift 31 EXTERN shift 4 REGISTER shift 6 SIGNED shift 7 STATIC shift 8	State 199 <Struct Def> ::= <Var Decl> • <Struct Def> <Struct Def> ::= <Var Decl> • <Sign> ::= • auto Shift 1 const Shift 2 enum Shift 24 extern Shift 17 register Shift 19 signed Shift 20 static Shift 21

<pre> STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 LBRACE reduce 30 RBRACE reduce 30 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 55 struct_def goto 85 type goto 57 mod goto 25 base goto 26 sign goto 27 </pre>	<pre> struct Shift 26 union Shift 28 unsigned Shift 30 volatile Shift 31 '{' Reduce 29 '}' Reduce 29 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Mod> Goto 36 <Sign> Goto 60 <Struct Def> Goto 200 <Type> Goto 195 <Var Decl> Goto 199 </pre>
<pre> state 56 base : STRUCT LBRACE struct_def . RBRACE (58) RBRACE shift 86 . error </pre>	<pre> State 202 <Base> ::= struct '{' <Struct Def> • '}' '}' Shift 203 </pre>
<pre> state 57 var_decl : type . var var_list SEMI (32) ID shift 41 . error var goto 40 </pre>	<pre> State 195 <Var Decl> ::= <Type> • <Var> <Var List> ';' Id Shift 37 <Var> Goto 196 </pre>
<pre> state 58 struct_decl : STRUCT ID LBRACE . struct_def RBRACE SEMI (27) sign : . (64) AUTO shift 1 CONST shift 2 ENUM shift 31 EXTERN shift 4 REGISTER shift 6 SIGNED shift 7 STATIC shift 8 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 55 struct_def goto 87 type goto 57 mod goto 25 </pre>	<pre> State 205 <Struct Decl> ::= struct Id '{' • <Struct Def> '}' ';' <Sign> ::= • auto Shift 1 const Shift 2 enum Shift 24 extern Shift 17 register Shift 19 signed Shift 20 static Shift 21 struct Shift 26 union Shift 28 unsigned Shift 30 volatile Shift 31 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Mod> Goto 36 <Sign> Goto 60 <Struct Def> Goto 206 <Type> Goto 195 </pre>

mod goto 25 base goto 26 sign goto 27	<Type> Goto 195 <Var Decl> Goto 199
state 65 func_proto : func_id LPARAN RPARAN . SEMI (12) func_decl : func_id LPARAN RPARAN . block (15) SEMI shift 91 LBRACE shift 92 . error block goto 93	State 225 <Func Proto> ::= <Func ID> '(' ')' • ';' <Func Decl> ::= <Func ID> '(' ')' • <Block> ';' Shift 226 '{' Shift 227 <Block> Goto 316
state 66 param : CONST . type ID (18) sign : . (64) ENUM shift 31 SIGNED shift 7 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 type goto 94 base goto 26 sign goto 27	State 317 <Param> ::= const • <Type> Id <Sign> ::= • enum Shift 24 signed Shift 20 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Sign> Goto 60 <Type> Goto 318
state 67 id_list : ID . COMMA id_list (22) id_list : ID . (23) COMMA shift 95 RPARAN reduce 23	State 320 <Id List> ::= Id • ',' <Id List> <Id List> ::= Id • ',' Shift 321 ')' Reduce 22
state 68 func_proto : func_id LPARAN types . RPARAN SEMI (10) RPARAN shift 96 . error	State 340 <Func Proto> ::= <Func ID> '(' <Types> • ')' ';' ' ')' Shift 341
state 69 func_proto : func_id LPARAN params . RPARAN SEMI (11) func_decl : func_id LPARAN params . RPARAN block (13) RPARAN shift 97 . error	State 332 <Func Proto> ::= <Func ID> '(' <Params> • ')' ';' ' <Func Decl> ::= <Func ID> '(' <Params> • ')' <Block> ')' Shift 333
state 70 func_decl : func_id LPARAN id_list . RPARAN struct_def block (14) RPARAN shift 98 . error	State 323 <Func Decl> ::= <Func ID> '(' <Id List> • ')' <Struct Def> <Block> ')' Shift 324
state 71	State 327

<pre> params : param . COMMA params (16) params : param . (17) COMMA shift 99 RPARAN reduce 17 </pre>	<pre> <Params> ::= <Param> • ',' <Params> <Params> ::= <Param> • ',' Shift 328 ')' Reduce 16 </pre>
<pre> state 72 param : type . ID (19) types : type . COMMA types (20) types : type . (21) COMMA shift 100 ID shift 101 RPARAN reduce 21 </pre>	<pre> State 336 <Param> ::= <Type> • Id <Types> ::= <Type> • ',' <Types> <Types> ::= <Type> • ',' Shift 337 Id Shift 331 ')' Reduce 20 </pre>
<pre> state 73 array : LBRACKET . expr RBRACKET (36) array : LBRACKET . RBRACKET (37) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 RBRACKET shift 108 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 expr goto 120 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 38 <Array> ::= '[' • <Expr> ']' <Array> ::= '[' • ']' '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ']' Shift 175 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Expr> Goto 176 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 74 var : ID array . (34) var : ID array . EQ op_if (35) EQ shift 135 COMMA reduce 34 SEMI reduce 34 </pre>	<pre> State 178 <Var> ::= Id <Array> • <Var> ::= Id <Array> • '=' <Op If> '=' Shift 179 ',' Reduce 33 ';' Reduce 33 </pre>
<pre> state 75 var_list : COMMA . var_item var_list (39) pointers : . (75) </pre>	<pre> State 183 <Var List> ::= ',' • <Var Item> <Var List> <Pointers> ::= • </pre>

TIMES shift 44 ID reduce 75 var_item goto 136 pointers goto 137	'*' Shift 33 Id Reduce 74 <Pointers> Goto 184 <Var Item> Goto 186
state 76 var_decl : type var var_list . SEMI (32) SEMI shift 138 . error	State 197 <Var Decl> ::= <Type> <Var> <Var List> • ';' ';' Shift 198
state 77 var_decl : mod type var . var_list SEMI (31) var_list : . (40) COMMA shift 75 SEMI reduce 40 var_list goto 139	State 182 <Var Decl> ::= <Mod> <Type> <Var> • <Var List> ';' ',' Shift 183 ';' Reduce 39 <Var List> Goto 188
state 78 var_decl : mod var var_list . SEMI (33) SEMI shift 140 . error	State 191 <Var Decl> ::= <Mod> <Var> <Var List> • ';' ';' Shift 192
state 79 pointers : TIMES pointers . (74) . reduce 74	State 34 <Pointers> ::= '*' <Pointers> • ')' Reduce 73 ',' Reduce 73 Id Reduce 73
state 80 scalar : LONG INT . (70) . reduce 70	State 66 <Scalar> ::= long int • ')' Reduce 69 '*' Reduce 69 ',' Reduce 69 Id Reduce 69
state 81 scalar : SHORT INT . (69) . reduce 69	State 68 <Scalar> ::= short int • ')' Reduce 68 '*' Reduce 68 ',' Reduce 68 Id Reduce 68
state 82 enum_val : ID . (51) enum_val : ID . EQ OCTLITERAL (52) enum_val : ID . EQ HEXLITERAL (53) enum_val : ID . EQ DECLITERAL (54) EQ shift 141 COMMA reduce 51 RBRACE reduce 51	State 6 <Enum Val> ::= Id • <Enum Val> ::= Id • '=' OctLiteral <Enum Val> ::= Id • '=' HexLiteral <Enum Val> ::= Id • '=' Decliteral '=' Shift 7 ',' Reduce 50 '}' Reduce 50
state 83 enum_decl : ENUM ID LBRACE enum_def . RBRACE SEMI (48) RBRACE shift 142 . error	State 11 <Enum Decl> ::= enum Id '{' <Enum Def> • '}' ';' '}' Shift 12

state 84 enum_def : enum_val . COMMA enum_def (49) enum_def : enum_val . (50) COMMA shift 143 RBRACE reduce 50	State 14 <Enum Def> ::= <Enum Val> • ',' <Enum Def> <Enum Def> ::= <Enum Val> • ',' Shift 15 '}' Reduce 49
state 85 struct_def : var_decl struct_def . (29) . reduce 29	State 200 <Struct Def> ::= <Var Decl> <Struct Def> • {' Reduce 28 }' Reduce 28
state 86 base : STRUCT LBRACE struct_def RBRACE . (58) . reduce 58	State 203 <Base> ::= struct {' <Struct Def> '}' • ')' Reduce 57 '*' Reduce 57 ',' Reduce 57 Id Reduce 57
state 87 struct_decl : STRUCT ID LBRACE struct_def . RBRACE SEMI (27) RBRACE shift 144 . error	State 206 <Struct Decl> ::= struct Id {' <Struct Def> • '}' ';' ; '}' Shift 207
state 88 typedef_decl : TYPEDEF type ID SEMI . (26) . reduce 26	State 212 <Typedef Decl> ::= typedef <Type> Id ';' • (EOF) Reduce 25 auto Reduce 25 char Reduce 25 const Reduce 25 double Reduce 25 enum Reduce 25 extern Reduce 25 float Reduce 25 Id Reduce 25 int Reduce 25 long Reduce 25 register Reduce 25 short Reduce 25 signed Reduce 25 static Reduce 25 struct Reduce 25 typedef Reduce 25 union Reduce 25 unsigned Reduce 25 void Reduce 25 volatile Reduce 25
state 89 base : UNION LBRACE struct_def RBRACE . (60) . reduce 60	State 194 <Base> ::= union {' <Struct Def> '}' • ')' Reduce 59 '*' Reduce 59 ',' Reduce 59 Id Reduce 59
state 90 union_decl : UNION ID LBRACE struct_def . RBRACE SEMI (28)	State 216 <Union Decl> ::= union Id {' <Struct Def> • '}' ';' ;

RBRACE shift 145 . error	'}' Shift 217
state 91 func_proto : func_id LPARAN RPARAN SEMI . (12) . reduce 12	State 226 <Func Proto> ::= <Func ID> '(' ' ' ')' ';' • (EOF) Reduce 11 auto Reduce 11 char Reduce 11 const Reduce 11 double Reduce 11 enum Reduce 11 extern Reduce 11 float Reduce 11 Id Reduce 11 int Reduce 11 long Reduce 11 register Reduce 11 short Reduce 11 signed Reduce 11 static Reduce 11 struct Reduce 11 typedef Reduce 11 union Reduce 11 unsigned Reduce 11 void Reduce 11 volatile Reduce 11
state 92 block : LBRACE . stm_list RBRACE (101) sign : . (64) stm_list : . (103) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 150 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 153 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8	State 227 <Block> ::= '{' • <Stm List> '}' <Sign> ::= • <Stm List> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 DecLiteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 234 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 247 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21

<p> STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 156 RBRACE reduce 103 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 160 normal_stm goto 161 value goto 121 stm_list goto 162 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </p>	<p> StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 274 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 '}' Reduce 102 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 281 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 284 <Stm List> Goto 314 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283 </p>
<p> state 93 func_decl : func_id LPARAN RPARAN block . (15) . reduce 15 </p>	<p> State 316 <Func Decl> ::= <Func ID> '(' ' ')' <Block> • (EOF) Reduce 14 auto Reduce 14 char Reduce 14 const Reduce 14 double Reduce 14 enum Reduce 14 extern Reduce 14 float Reduce 14 Id Reduce 14 int Reduce 14 long Reduce 14 register Reduce 14 short Reduce 14 signed Reduce 14 static Reduce 14 struct Reduce 14 typedef Reduce 14 union Reduce 14 unsigned Reduce 14 </p>

	void Reduce 14 volatile Reduce 14
state 94 param : CONST type . ID (18) ID shift 163 . error	State 318 <Param> ::= const <Type> • Id Id Shift 319
state 95 id_list : ID COMMA . id_list (22) ID shift 67 . error id_list goto 164	State 321 <Id List> ::= Id ',' • <Id List> Id Shift 320 <Id List> Goto 322
state 96 func_proto : func_id LPARAN types RPARAN . SEMI (10) SEMI shift 165 . error	State 341 <Func Proto> ::= <Func ID> '(' <Types> ')' • ';' ';' Shift 342
state 97 func_proto : func_id LPARAN params RPARAN . SEMI (11) func_decl : func_id LPARAN params RPARAN . block (13) SEMI shift 166 LBRACE shift 92 . error block goto 167	State 333 <Func Proto> ::= <Func ID> '(' <Params> ')' • ';' <Func Decl> ::= <Func ID> '(' <Params> ')' • <Block> ';' Shift 334 '{' Shift 227 <Block> Goto 335
state 98 func_decl : func_id LPARAN id_list RPARAN . struct_def block (14) sign : . (64) AUTO shift 1 CONST shift 2 ENUM shift 31 EXTERN shift 4 REGISTER shift 6 SIGNED shift 7 STATIC shift 8 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 55 struct_def goto 168 type goto 57 mod goto 25 base goto 26 sign goto 27	State 324 <Func Decl> ::= <Func ID> '(' <Id List> ')' • <Struct Def> <Block> <Sign> ::= • auto Shift 1 const Shift 2 enum Shift 24 extern Shift 17 register Shift 19 signed Shift 20 static Shift 21 struct Shift 26 union Shift 28 unsigned Shift 30 volatile Shift 31 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Mod> Goto 36 <Sign> Goto 60 <Struct Def> Goto 325 <Type> Goto 195 <Var Decl> Goto 199
state 99	State 328

<pre> params : param COMMA . params (16) sign : . (64) CONST shift 66 ENUM shift 31 SIGNED shift 7 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 params goto 169 param goto 71 type goto 170 base goto 26 sign goto 27 </pre>	<pre> <Params> ::= <Param> ',' • <Params> <Sign> ::= • const Shift 317 enum Shift 24 signed Shift 20 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Param> Goto 327 <Params> Goto 329 <Sign> Goto 60 <Type> Goto 330 </pre>
<pre> state 100 types : type COMMA . types (20) sign : . (64) ENUM shift 31 SIGNED shift 7 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 types goto 171 type goto 172 base goto 26 sign goto 27 </pre>	<pre> State 337 <Types> ::= <Type> ',' • <Types> <Sign> ::= • enum Shift 24 signed Shift 20 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Sign> Goto 60 <Type> Goto 338 <Types> Goto 339 </pre>
<pre> state 101 param : type ID . (19) . reduce 19 </pre>	<pre> State 331 <Param> ::= <Type> Id • ')' Reduce 18 ',' Reduce 18 </pre>
<pre> state 102 op_unary : MINUS . op_unary (149) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 </pre>	<pre> State 39 <Op Unary> ::= '-' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 </pre>

ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 173 op_pointer goto 134	Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 174 <Value> Goto 96
state 103 op_unary : MINUSMINUS . op_unary (153) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 174 op_pointer goto 134	State 40 <Op Unary> ::= '--' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 173 <Value> Goto 96
state 104 op_unary : EXCLAM . op_unary (147) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 175 op_pointer goto 134	State 41 <Op Unary> ::= '!' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 172 <Value> Goto 96
state 105 op_unary : AMP . op_unary (151) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104	State 42 <Op Unary> ::= '&' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41

<pre> AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 176 op_pointer goto 134 </pre>	<pre> '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 171 <Value> Goto 96 </pre>
<pre> state 106 op_unary : LPARAN . type RPARAN op_unary (156) value : LPARAN . expr RPARAN (173) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 ENUM shift 31 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIGNED shift 7 SIZEOF shift 117 STRINGLITERAL shift 118 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 type goto 177 op_if goto 119 expr goto 178 base goto 26 sign goto 27 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 </pre>	<pre> State 43 <Op Unary> ::= '(' • <Type> ')' <Op Unary> <Value> ::= '(' • <Expr> ')' <Sign> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 enum Shift 24 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 signed Shift 20 sizeof Shift 55 StringLiteral Shift 73 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Expr> Goto 85 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 </pre>

op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	<Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Type> Goto 168 <Value> Goto 96
state 107 op_unary : TIMES . op_unary (150) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 179 op_pointer goto 134	State 44 <Op Unary> ::= '*' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 167 <Value> Goto 96
state 108 array : LBRACKET RBRACKET . (37) . reduce 37	State 175 <Array> ::= '[' ']' • ',' Reduce 36 ';' Reduce 36 '=' Reduce 36
state 109 op_unary : TILDE . op_unary (148) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 180 op_pointer goto 134	State 45 <Op Unary> ::= '~' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 166 <Value> Goto 96
state 110	State 46

<pre> op_unary : PLUSPLUS . op_unary (152) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 181 op_pointer goto 134 </pre>	<pre> <Op Unary> ::= '++' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 165 <Value> Goto 96 </pre>
<pre> state 111 value : CHARLITERAL . (168) . reduce 168 </pre>	<pre> State 47 <Value> ::= CharLiteral • '-' Reduce 167 '--' Reduce 167 '!=' Reduce 167 '%' Reduce 167 '&' Reduce 167 '&&' Reduce 167 '&=' Reduce 167 ')' Reduce 167 '*' Reduce 167 '*=' Reduce 167 ',' Reduce 167 '.' Reduce 167 '/' Reduce 167 '/' Reduce 167 ':' Reduce 167 ';' Reduce 167 '?' Reduce 167 '[' Reduce 167 ']' Reduce 167 '^' Reduce 167 '^=' Reduce 167 ' ' Reduce 167 ' ' Reduce 167 ' =' Reduce 167 '+' Reduce 167 '++' Reduce 167 '+=' Reduce 167 '<' Reduce 167 '<<' Reduce 167 '<=' Reduce 167 '=' Reduce 167 '==' Reduce 167 '>' Reduce 167 '>' Reduce 167 '>=' Reduce 167 '>>' Reduce 167 </pre>

	'>>=' Reduce 167
state 112 value : DECLITERAL . (166) . reduce 166	State 48 <Value> ::= DeclLiteral • '-' Reduce 165 '--' Reduce 165 '!=' Reduce 165 '%' Reduce 165 '&' Reduce 165 '&&' Reduce 165 '&=' Reduce 165 ')' Reduce 165 '*' Reduce 165 '*=' Reduce 165 ',' Reduce 165 '.' Reduce 165 '/' Reduce 165 '/' Reduce 165 ':' Reduce 165 ';' Reduce 165 '?' Reduce 165 '[' Reduce 165 ']' Reduce 165 '^' Reduce 165 '^=' Reduce 165 ' ' Reduce 165 ' ' Reduce 165 ' =' Reduce 165 '+' Reduce 165 '++' Reduce 165 '+=' Reduce 165 '<' Reduce 165 '<<' Reduce 165 '<=' Reduce 165 '<=' Reduce 165 '=' Reduce 165 '-' Reduce 165 '==' Reduce 165 '>' Reduce 165 '>-' Reduce 165 '>=' Reduce 165 '>>' Reduce 165 '>>=' Reduce 165
state 113 value : FLOATLITERAL . (169) . reduce 169	State 49 <Value> ::= FloatLiteral • '-' Reduce 168 '--' Reduce 168 '!=' Reduce 168 '%' Reduce 168 '&' Reduce 168 '&&' Reduce 168 '&=' Reduce 168 ')' Reduce 168 '*' Reduce 168 '*=' Reduce 168 ',' Reduce 168 '.' Reduce 168 '/' Reduce 168 '/' Reduce 168 ':' Reduce 168 ';' Reduce 168 '?' Reduce 168 '[' Reduce 168

	']' Reduce 168 '^' Reduce 168 '^=' Reduce 168 ' ' Reduce 168 ' ' Reduce 168 ' =' Reduce 168 '+' Reduce 168 '++' Reduce 168 '+=' Reduce 168 '<' Reduce 168 '<<' Reduce 168 '<=' Reduce 168 '<=' Reduce 168 '=' Reduce 168 '-=' Reduce 168 '==' Reduce 168 '>' Reduce 168 '->' Reduce 168 '>=' Reduce 168 '>>' Reduce 168 '>>=' Reduce 168
state 114 value : HEXLITERAL . (165) . reduce 165	State 50 <Value> ::= HexLiteral • '-' Reduce 164 '--' Reduce 164 '!=' Reduce 164 '%' Reduce 164 '&' Reduce 164 '&&' Reduce 164 '&=' Reduce 164 ')' Reduce 164 '*' Reduce 164 '*=' Reduce 164 ',' Reduce 164 '.' Reduce 164 '/' Reduce 164 '/=' Reduce 164 ':' Reduce 164 ';' Reduce 164 '?' Reduce 164 '[' Reduce 164 ']' Reduce 164 '^' Reduce 164 '^=' Reduce 164 ' ' Reduce 164 ' ' Reduce 164 ' =' Reduce 164 '+' Reduce 164 '++' Reduce 164 '+=' Reduce 164 '<' Reduce 164 '<<' Reduce 164 '<=' Reduce 164 '<=' Reduce 164 '=' Reduce 164 '-=' Reduce 164 '==' Reduce 164 '>' Reduce 164 '->' Reduce 164 '>=' Reduce 164 '>>' Reduce 164 '>>=' Reduce 164
state 115	State 51

<pre> value : ID . LPARAN expr RPARAN (170) value : ID . LPARAN RPARAN (171) value : ID . (172) LPARAN shift 182 MINUS reduce 172 MINUSMINUS reduce 172 EXCLAMEQ reduce 172 PERCENT reduce 172 AMP reduce 172 AMPAMP reduce 172 AMPEQ reduce 172 RPARAN reduce 172 TIMES reduce 172 TIMESEQ reduce 172 COMMA reduce 172 DOT reduce 172 DIV reduce 172 DIVEQ reduce 172 COLON reduce 172 SEMI reduce 172 QUESTION reduce 172 LBRACKET reduce 172 RBRACKET reduce 172 CARET reduce 172 CARETEQ reduce 172 PIPE reduce 172 PIPEPIPE reduce 172 PIPEEQ reduce 172 PLUS reduce 172 PLUSPLUS reduce 172 PLUSEQ reduce 172 LT reduce 172 LTLT reduce 172 LTLTEQ reduce 172 LTEQ reduce 172 EQ reduce 172 MINUSEQ reduce 172 EQEQ reduce 172 GT reduce 172 MINUSGT reduce 172 GTEQ reduce 172 GTGT reduce 172 GTGTEQ reduce 172 </pre>	<pre> <Value> ::= Id • '(' <Expr> ')' <Value> ::= Id • '(' ')' <Value> ::= Id • '(' Shift 52 '-' Reduce 171 '--' Reduce 171 '!=' Reduce 171 '%' Reduce 171 '&' Reduce 171 '&&' Reduce 171 '&=' Reduce 171 ')' Reduce 171 '*' Reduce 171 '*=' Reduce 171 ',' Reduce 171 '.' Reduce 171 '/' Reduce 171 '/' Reduce 171 ':' Reduce 171 ';' Reduce 171 '?' Reduce 171 '[' Reduce 171 ']' Reduce 171 '^' Reduce 171 '^=' Reduce 171 ' ' Reduce 171 ' ' Reduce 171 ' =' Reduce 171 '+' Reduce 171 '++' Reduce 171 '+=' Reduce 171 '<' Reduce 171 '<<' Reduce 171 '<=' Reduce 171 '<=' Reduce 171 '=' Reduce 171 '==' Reduce 171 '==' Reduce 171 '>' Reduce 171 '>' Reduce 171 '>=' Reduce 171 '>=' Reduce 171 '>=' Reduce 171 </pre>
<pre> state 116 value : OCTLITERAL . (164) . reduce 164 </pre>	<pre> State 54 <Value> ::= OctLiteral • '-' Reduce 163 '--' Reduce 163 '!=' Reduce 163 '%' Reduce 163 '&' Reduce 163 '&&' Reduce 163 '&=' Reduce 163 ')' Reduce 163 '*' Reduce 163 '*=' Reduce 163 ',' Reduce 163 '.' Reduce 163 '/' Reduce 163 '/' Reduce 163 ':' Reduce 163 ';' Reduce 163 '?' Reduce 163 </pre>

	'[' Reduce 163 ']' Reduce 163 '^' Reduce 163 '^=' Reduce 163 ' ' Reduce 163 ' ' Reduce 163 ' =' Reduce 163 '+' Reduce 163 '++' Reduce 163 '+=' Reduce 163 '<' Reduce 163 '<<' Reduce 163 '<=' Reduce 163 '<=' Reduce 163 '=' Reduce 163 '-=' Reduce 163 '==' Reduce 163 '>' Reduce 163 '->' Reduce 163 '>=' Reduce 163 '>>' Reduce 163 '>=' Reduce 163
state 117 op_unary : SIZEOF . LPARAN type RPARAN (157) op_unary : SIZEOF . LPARAN ID pointers RPARAN (158) LPARAN shift 183 . error	State 55 <Op Unary> ::= sizeof • '(' <Type> ')' <Op Unary> ::= sizeof • '(' Id <Pointers> ')' '(' Shift 56
state 118 value : STRINGLITERAL . (167) . reduce 167	State 73 <Value> ::= StringLiteral • '-' Reduce 166 '--' Reduce 166 '!=' Reduce 166 '%' Reduce 166 '&' Reduce 166 '&&' Reduce 166 '&=' Reduce 166 ')' Reduce 166 '*' Reduce 166 '*=' Reduce 166 ',' Reduce 166 '.' Reduce 166 '/' Reduce 166 '/=' Reduce 166 ':' Reduce 166 ';' Reduce 166 '?' Reduce 166 '[' Reduce 166 ']' Reduce 166 '^' Reduce 166 '^=' Reduce 166 ' ' Reduce 166 ' ' Reduce 166 ' =' Reduce 166 '+' Reduce 166 '++' Reduce 166 '+=' Reduce 166 '<' Reduce 166 '<<' Reduce 166 '<=' Reduce 166 '<=' Reduce 166

	'=' Reduce 166 '-=' Reduce 166 '==' Reduce 166 '>' Reduce 166 '->' Reduce 166 '>=' Reduce 166 '>>' Reduce 166 '>>=' Reduce 166
state 119 op_assign : op_if . EQ op_assign (106) op_assign : op_if . PLUSEQ op_assign (107) op_assign : op_if . MINUSEQ op_assign (108) op_assign : op_if . TIMESEQ op_assign (109) op_assign : op_if . DIVEQ op_assign (110) op_assign : op_if . CARETEQ op_assign (111) op_assign : op_if . AMPEQ op_assign (112) op_assign : op_if . PIPEEQ op_assign (113) op_assign : op_if . GTGTEQ op_assign (114) op_assign : op_if . LTLTEQ op_assign (115) op_assign : op_if . (116) AMPEQ shift 184 TIMESEQ shift 185 DIVEQ shift 186 CARETEQ shift 187 PIPEEQ shift 188 PLUSEQ shift 189 LTLTEQ shift 190 EQ shift 191 MINUSEQ shift 192 GTGTEQ shift 193 RPARAN reduce 116 COMMA reduce 116 SEMI reduce 116 RBRACKET reduce 116	State 128 <Op Assign> ::= <Op If> • '=' <Op Assign> <Op Assign> ::= <Op If> • '+=' <Op Assign> <Op Assign> ::= <Op If> • '-=' <Op Assign> <Op Assign> ::= <Op If> • '*=' <Op Assign> <Op Assign> ::= <Op If> • '/=' <Op Assign> <Op Assign> ::= <Op If> • '^=' <Op Assign> <Op Assign> ::= <Op If> • '&=' <Op Assign> <Op Assign> ::= <Op If> • ' =' <Op Assign> <Op Assign> ::= <Op If> • '>>=' <Op Assign> <Op Assign> ::= <Op If> • '<=<' <Op Assign> <Op Assign> ::= <Op If> • '&=' Shift 129 '*=' Shift 138 '/=' Shift 140 '^=' Shift 142 ' =' Shift 144 '+=' Shift 146 '<=<' Shift 148 '=' Shift 150 '-=' Shift 152 '>>=' Shift 154 ')' Reduce 115 ',' Reduce 115 ';' Reduce 115 ']' Reduce 115
state 120 array : LBRACKET expr . RBRACKET (36) expr : expr . COMMA op_assign (104) COMMA shift 194 RBRACKET shift 195 . error	State 176 <Array> ::= '[' <Expr> • ']' <Expr> ::= <Expr> • ',' <Op Assign> ',' Shift 76 ']' Shift 177
state 121 op_pointer : value . (163) . reduce 163	State 96 <Op Pointer> ::= <Value> • '-' Reduce 162 '--' Reduce 162 '!=' Reduce 162 '%' Reduce 162 '&' Reduce 162 '&&' Reduce 162 '&=' Reduce 162 ')' Reduce 162

	'*' Reduce 162 '*=' Reduce 162 ',' Reduce 162 '.' Reduce 162 '/' Reduce 162 '/=' Reduce 162 ':' Reduce 162 ';' Reduce 162 '?' Reduce 162 '[' Reduce 162 ']' Reduce 162 '^' Reduce 162 '^=' Reduce 162 ' ' Reduce 162 ' ' Reduce 162 ' =' Reduce 162 '+' Reduce 162 '++' Reduce 162 '+=' Reduce 162 '<' Reduce 162 '<<' Reduce 162 '<=' Reduce 162 '<=' Reduce 162 '=' Reduce 162 '-=' Reduce 162 '==' Reduce 162 '>' Reduce 162 '->' Reduce 162 '>=' Reduce 162 '>>' Reduce 162 '>>=' Reduce 162
state 122 expr : op_assign . (105) . reduce 105	State 126 <Expr> ::= <Op Assign> • ')' Reduce 104 ',' Reduce 104 ';' Reduce 104 ']' Reduce 104
state 123 op_if : op_or . QUESTION op_if COLON op_if (117) op_if : op_or . (118) op_or : op_or . PIPEPIPE op_and (119) QUESTION shift 196 PIPEPIPE shift 197 AMPEQ reduce 118 RPARAN reduce 118 TIMESEQ reduce 118 COMMA reduce 118 DIVEQ reduce 118 COLON reduce 118 SEMI reduce 118 RBRACKET reduce 118 CARETEQ reduce 118 PIPEEQ reduce 118 PLUSEQ reduce 118 LTLTEQ reduce 118 EQ reduce 118 MINUSEQ reduce 118 GTGTEQ reduce 118	State 131 <Op If> ::= <Op Or> • '?' <Op If> ':' <Op If> <Op If> ::= <Op Or> • <Op Or> ::= <Op Or> • ' ' <Op And> '?' Shift 132 ' ' Shift 136 '&=' Reduce 117 ')' Reduce 117 '*=' Reduce 117 ',' Reduce 117 '/=' Reduce 117 ':' Reduce 117 ';' Reduce 117 ']' Reduce 117 '^=' Reduce 117 ' =' Reduce 117 '+=' Reduce 117 '<=' Reduce 117 '=' Reduce 117 '-=' Reduce 117 '>>=' Reduce 117
state 124 op_or : op_and . (120)	State 87 <Op Or> ::= <Op And> •

<pre> op_and : op_and . AMPAMP op_binor (121) AMPAMP shift 198 AMPEQ reduce 120 RPARAN reduce 120 TIMESEQ reduce 120 COMMA reduce 120 DIVEQ reduce 120 COLON reduce 120 SEMI reduce 120 QUESTION reduce 120 RBRACKET reduce 120 CARETEQ reduce 120 PIPEPIPE reduce 120 PIPEEQ reduce 120 PLUSEQ reduce 120 LTLTEQ reduce 120 EQ reduce 120 MINUSEQ reduce 120 GTGTEQ reduce 120 </pre>	<pre> <Op And> ::= <Op And> • '&&' <Op BinOR> '&&' Shift 88 '&=' Reduce 119 ')' Reduce 119 '*=' Reduce 119 ',' Reduce 119 '/' Reduce 119 ':' Reduce 119 ';' Reduce 119 '?' Reduce 119 ']' Reduce 119 '^=' Reduce 119 ' ' Reduce 119 ' =' Reduce 119 '+' Reduce 119 '<=' Reduce 119 '=' Reduce 119 '-' Reduce 119 '>=' Reduce 119 </pre>
<pre> state 125 op_and : op_binor . (122) op_binor : op_binor . PIPE op_binxor (123) PIPE shift 199 AMPAMP reduce 122 AMPEQ reduce 122 RPARAN reduce 122 TIMESEQ reduce 122 COMMA reduce 122 DIVEQ reduce 122 COLON reduce 122 SEMI reduce 122 QUESTION reduce 122 RBRACKET reduce 122 CARETEQ reduce 122 PIPEPIPE reduce 122 PIPEEQ reduce 122 PLUSEQ reduce 122 LTLTEQ reduce 122 EQ reduce 122 MINUSEQ reduce 122 GTGTEQ reduce 122 </pre>	<pre> State 127 <Op And> ::= <Op BinOR> • <Op BinOR> ::= <Op BinOR> • ' ' <Op BinXOR> ' ' Shift 120 '&&' Reduce 121 '&=' Reduce 121 ')' Reduce 121 '*=' Reduce 121 ',' Reduce 121 '/' Reduce 121 ':' Reduce 121 ';' Reduce 121 '?' Reduce 121 ']' Reduce 121 '^=' Reduce 121 ' ' Reduce 121 ' =' Reduce 121 '+' Reduce 121 '<=' Reduce 121 '=' Reduce 121 '-' Reduce 121 '>=' Reduce 121 </pre>
<pre> state 126 op_binor : op_binxor . (124) op_binxor : op_binxor . CARET op_binand (125) CARET shift 200 AMPAMP reduce 124 AMPEQ reduce 124 RPARAN reduce 124 TIMESEQ reduce 124 COMMA reduce 124 DIVEQ reduce 124 COLON reduce 124 SEMI reduce 124 QUESTION reduce 124 RBRACKET reduce 124 CARETEQ reduce 124 PIPE reduce 124 PIPEPIPE reduce 124 </pre>	<pre> State 125 <Op BinOR> ::= <Op BinXOR> • <Op BinXOR> ::= <Op BinXOR> • '^' <Op BinAND> '^' Shift 122 '&&' Reduce 123 '&=' Reduce 123 ')' Reduce 123 '*=' Reduce 123 ',' Reduce 123 '/' Reduce 123 ':' Reduce 123 ';' Reduce 123 '?' Reduce 123 ']' Reduce 123 '^=' Reduce 123 ' ' Reduce 123 ' =' Reduce 123 </pre>

<pre> op_equate : op_compare . (131) op_compare : op_compare . LT op_shift (132) op_compare : op_compare . GT op_shift (133) op_compare : op_compare . LTEQ op_shift (134) op_compare : op_compare . GTEQ op_shift (135) LT shift 204 LTEQ shift 205 GT shift 206 GTEQ shift 207 EXCLAMEQ reduce 131 AMP reduce 131 AMPAMP reduce 131 AMPEQ reduce 131 RPARAN reduce 131 TIMESEQ reduce 131 COMMA reduce 131 DIVEQ reduce 131 COLON reduce 131 SEMI reduce 131 QUESTION reduce 131 RBRACKET reduce 131 CARET reduce 131 CARETEQ reduce 131 PIPE reduce 131 PIPEPIPE reduce 131 PIPEEQ reduce 131 PLUSEQ reduce 131 LTLTEQ reduce 131 EQ reduce 131 MINUSEQ reduce 131 EQEQ reduce 131 GTGTEQ reduce 131 </pre>	<pre> <Op Equate> ::= <Op Compare> • <Op Compare> ::= <Op Compare> • '<' <Op Shift> <Op Compare> ::= <Op Compare> • '>' <Op Shift> <Op Compare> ::= <Op Compare> • '<=' <Op Shift> <Op Compare> ::= <Op Compare> • '>=' <Op Shift> '<' Shift 92 '<=' Shift 107 '>' Shift 109 '>=' Shift 111 '!=' Reduce 130 '&' Reduce 130 '&&' Reduce 130 '&=' Reduce 130 ')' Reduce 130 '*=' Reduce 130 ',' Reduce 130 '/' Reduce 130 ':' Reduce 130 ';' Reduce 130 '?' Reduce 130 ']' Reduce 130 '^' Reduce 130 '^=' Reduce 130 ' ' Reduce 130 ' ' Reduce 130 ' =' Reduce 130 '+=' Reduce 130 '<<=' Reduce 130 '=' Reduce 130 '-' Reduce 130 '==' Reduce 130 '>>=' Reduce 130 </pre>
<pre> state 130 op_compare : op_shift . (136) op_shift : op_shift . LTLT op_add (137) op_shift : op_shift . GTGT op_add (138) LTLT shift 208 GTGT shift 209 EXCLAMEQ reduce 136 AMP reduce 136 AMPAMP reduce 136 AMPEQ reduce 136 RPARAN reduce 136 TIMESEQ reduce 136 COMMA reduce 136 DIVEQ reduce 136 COLON reduce 136 SEMI reduce 136 QUESTION reduce 136 RBRACKET reduce 136 CARET reduce 136 CARETEQ reduce 136 PIPE reduce 136 PIPEPIPE reduce 136 PIPEEQ reduce 136 PLUSEQ reduce 136 LT reduce 136 LTLTEQ reduce 136 </pre>	<pre> State 116 <Op Compare> ::= <Op Shift> • <Op Shift> ::= <Op Shift> • '<<' <Op Add> <Op Shift> ::= <Op Shift> • '>>' <Op Add> '<<' Shift 100 '>>' Shift 105 '!=' Reduce 135 '&' Reduce 135 '&&' Reduce 135 '&=' Reduce 135 ')' Reduce 135 '*=' Reduce 135 ',' Reduce 135 '/' Reduce 135 ':' Reduce 135 ';' Reduce 135 '?' Reduce 135 ']' Reduce 135 '^' Reduce 135 '^=' Reduce 135 ' ' Reduce 135 ' ' Reduce 135 ' =' Reduce 135 '+=' Reduce 135 '<' Reduce 135 '<<=' Reduce 135 </pre>

LTEQ reduce 136 EQ reduce 136 MINUSEQ reduce 136 EQEQ reduce 136 GT reduce 136 GTEQ reduce 136 GTGTEQ reduce 136	'<=' Reduce 135 '=' Reduce 135 '-=' Reduce 135 '==' Reduce 135 '>' Reduce 135 '>=' Reduce 135 '>>=' Reduce 135
state 131 op_shift : op_add . (139) op_add : op_add . PLUS op_mult (140) op_add : op_add . MINUS op_mult (141) MINUS shift 210 PLUS shift 211 EXCLAMEQ reduce 139 AMP reduce 139 AMPAMP reduce 139 AMPEQ reduce 139 RPARAN reduce 139 TIMESEQ reduce 139 COMMA reduce 139 DIVEQ reduce 139 COLON reduce 139 SEMI reduce 139 QUESTION reduce 139 RBRACKET reduce 139 CARET reduce 139 CARETEQ reduce 139 PIPE reduce 139 PIPEPIPE reduce 139 PIPEEQ reduce 139 PLUSEQ reduce 139 LT reduce 139 LTLT reduce 139 LTLTEQ reduce 139 LTEQ reduce 139 EQ reduce 139 MINUSEQ reduce 139 EQEQ reduce 139 GT reduce 139 GTEQ reduce 139 GTGT reduce 139 GTGTEQ reduce 139	State 77 <Op Shift> ::= <Op Add> • <Op Add> ::= <Op Add> • '+' <Op Mult> <Op Add> ::= <Op Add> • '-' <Op Mult> '-' Shift 78 '+' Shift 102 '!=' Reduce 138 '&' Reduce 138 '&&' Reduce 138 '&=' Reduce 138 ')' Reduce 138 '*' Reduce 138 ',' Reduce 138 '/' Reduce 138 ':' Reduce 138 ';' Reduce 138 '?' Reduce 138 ']' Reduce 138 '^' Reduce 138 '^=' Reduce 138 ' ' Reduce 138 ' ' Reduce 138 ' =' Reduce 138 '+=' Reduce 138 '<' Reduce 138 '<<' Reduce 138 '<=<' Reduce 138 '<=' Reduce 138 '=' Reduce 138 '-=' Reduce 138 '==' Reduce 138 '>' Reduce 138 '>=' Reduce 138 '>>' Reduce 138 '>>=' Reduce 138
state 132 op_add : op_mult . (142) op_mult : op_mult . TIMES op_unary (143) op_mult : op_mult . DIV op_unary (144) op_mult : op_mult . PERCENT op_unary (145) PERCENT shift 212 TIMES shift 213 DIV shift 214 MINUS reduce 142 EXCLAMEQ reduce 142 AMP reduce 142 AMPAMP reduce 142 AMPEQ reduce 142 RPARAN reduce 142 TIMESEQ reduce 142 COMMA reduce 142 DIVEQ reduce 142 COLON reduce 142	State 93 <Op Add> ::= <Op Mult> • <Op Mult> ::= <Op Mult> • '*' <Op Unary> <Op Mult> ::= <Op Mult> • '/' <Op Unary> <Op Mult> ::= <Op Mult> • '%' <Op Unary> '%' Shift 80 '*' Shift 94 '/' Shift 97 '-' Reduce 141 '!=' Reduce 141 '&' Reduce 141 '&&' Reduce 141 '&=' Reduce 141 ')' Reduce 141 '*' Reduce 141 ',' Reduce 141 '/' Reduce 141

SEMI reduce 142 QUESTION reduce 142 RBRACKET reduce 142 CARET reduce 142 CARETEQ reduce 142 PIPE reduce 142 PIPEPIPE reduce 142 PIPEEQ reduce 142 PLUS reduce 142 PLUSEQ reduce 142 LT reduce 142 LTLT reduce 142 LTLTEQ reduce 142 LTEQ reduce 142 EQ reduce 142 MINUSEQ reduce 142 EQEQ reduce 142 GT reduce 142 GTEQ reduce 142 GTGT reduce 142 GTGTEQ reduce 142	':' Reduce 141 ';' Reduce 141 '?' Reduce 141 ']' Reduce 141 '^' Reduce 141 '^=' Reduce 141 ' ' Reduce 141 ' ' Reduce 141 ' =' Reduce 141 '+' Reduce 141 '+=' Reduce 141 '<' Reduce 141 '<<' Reduce 141 '<=' Reduce 141 '<=' Reduce 141 '=' Reduce 141 '==' Reduce 141 '==' Reduce 141 '>' Reduce 141 '>=' Reduce 141 '>>' Reduce 141 '>>=' Reduce 141
state 133 op_mult : op_unary . (146) . reduce 146	State 104 <Op Mult> ::= <Op Unary> • '-' Reduce 145 '!=' Reduce 145 '%' Reduce 145 '&' Reduce 145 '&&' Reduce 145 '&=' Reduce 145 ')' Reduce 145 '*' Reduce 145 '*=' Reduce 145 ',' Reduce 145 '/' Reduce 145 '/' Reduce 145 ':' Reduce 145 ';' Reduce 145 '?' Reduce 145 ']' Reduce 145 '^' Reduce 145 '^=' Reduce 145 ' ' Reduce 145 ' ' Reduce 145 ' =' Reduce 145 '+' Reduce 145 '+=' Reduce 145 '<' Reduce 145 '<<' Reduce 145 '<=' Reduce 145 '<=' Reduce 145 '=' Reduce 145 '==' Reduce 145 '==' Reduce 145 '>' Reduce 145 '>=' Reduce 145 '>>' Reduce 145 '>>=' Reduce 145
state 134 op_unary : op_pointer . PLUSPLUS (154) op_unary : op_pointer . MINUSMINUS (155) op_unary : op_pointer . (159)	State 81 <Op Unary> ::= <Op Pointer> • '++' <Op Unary> ::= <Op Pointer> • '--' <Op Unary> ::= <Op Pointer> • <Op Pointer> ::= <Op Pointer> • '.'

<pre> op_pointer : op_pointer . DOT value (160) op_pointer : op_pointer . MINUSGT value (161) op_pointer : op_pointer . LBRACKET expr RBRACKET (162) MINUSMINUS shift 215 DOT shift 216 LBRACKET shift 217 PLUSPLUS shift 218 MINUSGT shift 219 MINUS reduce 159 EXCLAMEQ reduce 159 PERCENT reduce 159 AMP reduce 159 AMPAMP reduce 159 AMPEQ reduce 159 RPARAN reduce 159 TIMES reduce 159 TIMESEQ reduce 159 COMMA reduce 159 DIV reduce 159 DIVEQ reduce 159 COLON reduce 159 SEMI reduce 159 QUESTION reduce 159 RBRACKET reduce 159 CARET reduce 159 CARETEQ reduce 159 PIPE reduce 159 PIPEPIPE reduce 159 PIPEEQ reduce 159 PLUS reduce 159 PLUSEQ reduce 159 LT reduce 159 LTLT reduce 159 LTLTEQ reduce 159 LTEQ reduce 159 EQ reduce 159 MINUSEQ reduce 159 EQEQ reduce 159 GT reduce 159 GTEQ reduce 159 GTGT reduce 159 GTGTEQ reduce 159 </pre>	<pre> <Value> <Op Pointer> ::= <Op Pointer> • '->' <Value> <Op Pointer> ::= <Op Pointer> • '[' <Expr> ']' '--' Shift 82 '.' Shift 83 '[' Shift 157 '++' Shift 160 '-->' Shift 161 '-' Reduce 158 '!=' Reduce 158 '%' Reduce 158 '&' Reduce 158 '&&' Reduce 158 '&=' Reduce 158 ')' Reduce 158 '*' Reduce 158 '*=' Reduce 158 ',' Reduce 158 '/' Reduce 158 '/' Reduce 158 '/' Reduce 158 ':' Reduce 158 ';' Reduce 158 '?' Reduce 158 ']' Reduce 158 '^' Reduce 158 '^=' Reduce 158 ' ' Reduce 158 ' ' Reduce 158 ' =' Reduce 158 '+' Reduce 158 '+=' Reduce 158 '<' Reduce 158 '<<' Reduce 158 '<=' Reduce 158 '=' Reduce 158 '==' Reduce 158 '>' Reduce 158 '>=' Reduce 158 '>>' Reduce 158 '>=' Reduce 158 </pre>
<pre> state 135 var : ID array EQ . op_if (35) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 </pre>	<pre> State 179 <Var> ::= Id <Array> '=' • <Op If> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 </pre>

<pre> STRINGLITERAL shift 118 . error op_if goto 220 value goto 121 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 180 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 136 var_list : COMMA var_item . var_list (39) var_list : . (40) COMMA shift 75 SEMI reduce 40 var_list goto 221 </pre>	<pre> State 186 <Var List> ::= ',' <Var Item> • <Var List> <Var List> ::= • ',' Shift 183 ';' Reduce 39 <Var List> Goto 187 </pre>
<pre> state 137 var_item : pointers . var (41) ID shift 41 . error var goto 222 </pre>	<pre> State 184 <Var Item> ::= <Pointers> • <Var> Id Shift 37 <Var> Goto 185 </pre>
<pre> state 138 var_decl : type var var_list SEMI . (32) . reduce 32 </pre>	<pre> State 198 <Var Decl> ::= <Type> <Var> <Var List> ';' • (EOF) Reduce 31 '-' Reduce 31 '--' Reduce 31 '!' Reduce 31 '&' Reduce 31 '(' Reduce 31 '*' Reduce 31 ';' Reduce 31 '{' Reduce 31 '}' Reduce 31 '~' Reduce 31 '++' Reduce 31 auto Reduce 31 break Reduce 31 case Reduce 31 char Reduce 31 CharLiteral Reduce 31 const Reduce 31 continue Reduce 31 Decliteral Reduce 31 default Reduce 31 do Reduce 31 double Reduce 31 enum Reduce 31 extern Reduce 31 float Reduce 31 </pre>

	FloatLiteral Reduce 31 for Reduce 31 goto Reduce 31 HexLiteral Reduce 31 Id Reduce 31 if Reduce 31 int Reduce 31 long Reduce 31 OctLiteral Reduce 31 register Reduce 31 return Reduce 31 short Reduce 31 signed Reduce 31 sizeof Reduce 31 static Reduce 31 StringLiteral Reduce 31 struct Reduce 31 switch Reduce 31 typedef Reduce 31 union Reduce 31 unsigned Reduce 31 void Reduce 31 volatile Reduce 31 while Reduce 31
state 139 var_decl : mod type var var_list . SEMI (31) SEMI shift 223 . error	State 188 <Var Decl> ::= <Mod> <Type> <Var> <Var List> • ';' ';' Shift 189
state 140 var_decl : mod var var_list SEMI . (33) . reduce 33	State 192 <Var Decl> ::= <Mod> <Var> <Var List> ';' • (EOF) Reduce 32 '-' Reduce 32 '--' Reduce 32 '!' Reduce 32 '&' Reduce 32 '(' Reduce 32 '*' Reduce 32 ';' Reduce 32 '{' Reduce 32 '}' Reduce 32 '~' Reduce 32 '++' Reduce 32 auto Reduce 32 break Reduce 32 case Reduce 32 char Reduce 32 CharLiteral Reduce 32 const Reduce 32 continue Reduce 32 DecLiteral Reduce 32 default Reduce 32 do Reduce 32 double Reduce 32 enum Reduce 32 extern Reduce 32 float Reduce 32 FloatLiteral Reduce 32 for Reduce 32 goto Reduce 32 HexLiteral Reduce 32

	'&' Reduce 94 '(' Reduce 94 '*' Reduce 94 ';' Reduce 94 '{' Reduce 94 '}' Reduce 94 '~' Reduce 94 '++' Reduce 94 auto Reduce 94 break Reduce 94 case Reduce 94 char Reduce 94 CharLiteral Reduce 94 const Reduce 94 continue Reduce 94 DecLiteral Reduce 94 default Reduce 94 do Reduce 94 double Reduce 94 else Reduce 94 enum Reduce 94 extern Reduce 94 float Reduce 94 FloatLiteral Reduce 94 for Reduce 94 goto Reduce 94 HexLiteral Reduce 94 Id Reduce 94 if Reduce 94 int Reduce 94 long Reduce 94 OctLiteral Reduce 94 register Reduce 94 return Reduce 94 short Reduce 94 signed Reduce 94 sizeof Reduce 94 static Reduce 94 StringLiteral Reduce 94 struct Reduce 94 switch Reduce 94 union Reduce 94 unsigned Reduce 94 void Reduce 94 volatile Reduce 94 while Reduce 94
state 147 normal_stm : BREAK . SEMI (92) SEMI shift 231 . error	State 229 <Normal Stm> ::= break • ';'
state 148 normal_stm : CONTINUE . SEMI (93) SEMI shift 232 . error	State 231 <Normal Stm> ::= continue • ';'
state 149 normal_stm : DO . stm WHILE LPARAN expr RPARAN (87) sign : . (64) MINUS shift 102 MINUSMINUS shift 103	State 233 <Normal Stm> ::= do • <Stm> while '(' <Expr> ')'
	<Sign> ::= • '-' Shift 39 '--' Shift 40

EXCLAM shift 104	'!' Shift 41
AMP shift 105	'&' Shift 42
LPARAN shift 106	'(' Shift 43
TIMES shift 107	'*' Shift 44
SEMI shift 146	';' Shift 228
LBRACE shift 92	'{' Shift 227
TILDE shift 109	'~' Shift 45
PLUSPLUS shift 110	'++' Shift 46
AUTO shift 1	auto Shift 1
BREAK shift 147	break Shift 229
CHARLITERAL shift 111	CharLiteral Shift 47
CONST shift 2	const Shift 2
CONTINUE shift 148	continue Shift 231
DECLITERAL shift 112	DecLiteral Shift 48
DO shift 149	do Shift 233
ENUM shift 31	enum Shift 24
EXTERN shift 4	extern Shift 17
FLOATLITERAL shift 113	FloatLiteral Shift 49
FOR shift 150	for Shift 234
GOTO shift 151	goto Shift 242
HEXLITERAL shift 114	HexLiteral Shift 50
ID shift 152	Id Shift 245
IF shift 153	if Shift 247
OCTLITERAL shift 116	OctLiteral Shift 54
REGISTER shift 6	register Shift 19
RETURN shift 154	return Shift 263
SIGNED shift 7	signed Shift 20
SIZEOF shift 117	sizeof Shift 55
STATIC shift 8	static Shift 21
STRINGLITERAL shift 118	StringLiteral Shift 73
STRUCT shift 32	struct Shift 26
SWITCH shift 155	switch Shift 266
UNION shift 33	union Shift 28
UNSIGNED shift 12	unsigned Shift 30
VOLATILE shift 13	volatile Shift 31
WHILE shift 156	while Shift 274
CHAR reduce 64	char Reduce 63
DOUBLE reduce 64	double Reduce 63
FLOAT reduce 64	float Reduce 63
INT reduce 64	int Reduce 63
LONG reduce 64	long Reduce 63
SHORT reduce 64	short Reduce 63
VOID reduce 64	void Reduce 63
var_decl goto 157	<Base> Goto 32
block goto 158	<Block> Goto 278
type goto 57	<Expr> Goto 279
mod goto 25	<Mod> Goto 36
op_if goto 119	<Normal Stm> Goto 281
expr goto 159	<Op Add> Goto 77
base goto 26	<Op And> Goto 87
sign goto 27	<Op Assign> Goto 126
stm goto 233	<Op BinAND> Goto 89
normal_stm goto 161	<Op BinOR> Goto 127
value goto 121	<Op BinXOR> Goto 125
op_assign goto 122	<Op Compare> Goto 91
op_or goto 123	<Op Equate> Goto 124
op_and goto 124	<Op If> Goto 128
op_binor goto 125	<Op Mult> Goto 93
op_binxor goto 126	<Op Or> Goto 131
op_binand goto 127	<Op Pointer> Goto 81
op_equate goto 128	<Op Shift> Goto 116
op_compare goto 129	<Op Unary> Goto 104
op_shift goto 130	<Sign> Goto 60

op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	<Stm> Goto 309 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283
state 150 stm : FOR . LPARAN arg SEMI arg SEMI arg RPARAN stm (81) LPARAN shift 234 . error	State 234 <Stm> ::= for • '(' <Arg> ';' <Arg> ';' <Arg> ')' <Stm> '(' Shift 235
state 151 normal_stm : GOTO . ID SEMI (91) ID shift 235 . error	State 242 <Normal Stm> ::= goto • Id ';' <Id> Id Shift 243
state 152 stm : ID . COLON (77) value : ID . LPARAN expr RPARAN (170) value : ID . LPARAN RPARAN (171) value : ID . (172) LPARAN shift 182 COLON shift 236 MINUS reduce 172 MINUSMINUS reduce 172 EXCLAMEQ reduce 172 PERCENT reduce 172 AMP reduce 172 AMPAMP reduce 172 AMPEQ reduce 172 TIMES reduce 172 TIMESEQ reduce 172 COMMA reduce 172 DOT reduce 172 DIV reduce 172 DIVEQ reduce 172 SEMI reduce 172 QUESTION reduce 172 LBRACKET reduce 172 CARET reduce 172 CARETEQ reduce 172 PIPE reduce 172 PIPEPIPE reduce 172 PIPEEQ reduce 172 PLUS reduce 172 PLUSPLUS reduce 172 PLUSEQ reduce 172 LT reduce 172 LTLT reduce 172 LTLTEQ reduce 172 LTEQ reduce 172 EQ reduce 172 MINUSEQ reduce 172 EQEQ reduce 172 GT reduce 172 MINUSGT reduce 172 GTEQ reduce 172 GTGT reduce 172 GTGTEQ reduce 172	State 245 <Stm> ::= Id • ':' <Value> ::= Id • '(' <Expr> ')' <Value> <Value> ::= Id • '(' ')' <Value> <Value> ::= Id • '(' Shift 52 ':' Shift 246 '-' Reduce 171 '--' Reduce 171 '!=' Reduce 171 '%' Reduce 171 '&' Reduce 171 '&&' Reduce 171 '&=' Reduce 171 '*' Reduce 171 '*=' Reduce 171 ',' Reduce 171 '.' Reduce 171 '/' Reduce 171 '/' Reduce 171 ';' Reduce 171 '? ' Reduce 171 '[' Reduce 171 '^' Reduce 171 '^=' Reduce 171 ' ' Reduce 171 ' ' Reduce 171 ' =' Reduce 171 '+' Reduce 171 '++' Reduce 171 '+=' Reduce 171 '<' Reduce 171 '<<' Reduce 171 '<=' Reduce 171 '=' Reduce 171 '-' Reduce 171 '==' Reduce 171 '>' Reduce 171 '>=' Reduce 171 '>>' Reduce 171 '>=' Reduce 171
state 153 stm : IF . LPARAN expr RPARAN stm (78) stm : IF . LPARAN expr RPARAN then_stm ELSE stm (79)	State 247 <Stm> ::= if • '(' <Expr> ')' <Stm> <Stm> ::= if • '(' <Expr> ')' <Then Stm> else <Stm>

LPARAN shift 237 . error	'(' Shift 248
state 154 normal_stm : RETURN . expr SEMI (94) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 expr goto 238 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 263 <Normal Stm> ::= return • <Expr> ';' '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DeclLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Expr> Goto 264 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 155 normal_stm : SWITCH . LPARAN expr RPARAN LBRACE case_stms RBRACE (88) LPARAN shift 239 . error	State 266 <Normal Stm> ::= switch • '(' <Expr> ')' '{' <Case Stms> '}' '(' Shift 267
state 156 stm : WHILE . LPARAN expr RPARAN stm (80) LPARAN shift 240 . error	State 274 <Stm> ::= while • '(' <Expr> ')' <Stm> '(' Shift 275
state 157 stm : var_decl . (76) . reduce 76	State 283 <Stm> ::= <Var Decl> • '-' Reduce 75 '--' Reduce 75 '!' Reduce 75 '&' Reduce 75 '(' Reduce 75 '*' Reduce 75 ';' Reduce 75

	'{' Reduce 75 '}' Reduce 75 '~' Reduce 75 '++' Reduce 75 auto Reduce 75 break Reduce 75 case Reduce 75 char Reduce 75 CharLiteral Reduce 75 const Reduce 75 continue Reduce 75 DecLiteral Reduce 75 default Reduce 75 do Reduce 75 double Reduce 75 enum Reduce 75 extern Reduce 75 float Reduce 75 FloatLiteral Reduce 75 for Reduce 75 goto Reduce 75 HexLiteral Reduce 75 Id Reduce 75 if Reduce 75 int Reduce 75 long Reduce 75 OctLiteral Reduce 75 register Reduce 75 return Reduce 75 short Reduce 75 signed Reduce 75 sizeof Reduce 75 static Reduce 75 StringLiteral Reduce 75 struct Reduce 75 switch Reduce 75 union Reduce 75 unsigned Reduce 75 void Reduce 75 volatile Reduce 75 while Reduce 75
state 158 normal_stm : block . (89) . reduce 89	State 278 <Normal Stm> ::= <Block> • '-' Reduce 88 '--' Reduce 88 '!' Reduce 88 '&' Reduce 88 '(' Reduce 88 '*' Reduce 88 ';' Reduce 88 '{' Reduce 88 '}' Reduce 88 '~' Reduce 88 '++' Reduce 88 auto Reduce 88 break Reduce 88 case Reduce 88 char Reduce 88 CharLiteral Reduce 88 const Reduce 88 continue Reduce 88 DecLiteral Reduce 88 default Reduce 88

	do Reduce 88 double Reduce 88 else Reduce 88 enum Reduce 88 extern Reduce 88 float Reduce 88 FloatLiteral Reduce 88 for Reduce 88 goto Reduce 88 HexLiteral Reduce 88 Id Reduce 88 if Reduce 88 int Reduce 88 long Reduce 88 OctLiteral Reduce 88 register Reduce 88 return Reduce 88 short Reduce 88 signed Reduce 88 sizeof Reduce 88 static Reduce 88 StringLiteral Reduce 88 struct Reduce 88 switch Reduce 88 union Reduce 88 unsigned Reduce 88 void Reduce 88 volatile Reduce 88 while Reduce 88
state 159 normal_stm : expr . SEMI (90) expr : expr . COMMA op_assign (104) COMMA shift 194 SEMI shift 241 . error	State 279 <Normal Stm> ::= <Expr> • ';' ; <Expr> ::= <Expr> • ',' <Op Assign> ',' Shift 76 ';' Shift 280
state 160 stm_list : stm . stm_list (102) sign : . (64) stm_list : . (103) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 150 GOTO shift 151 HEXLITERAL shift 114	State 284 <Stm List> ::= <Stm> • <Stm List> <Sign> ::= • <Stm List> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 DecLiteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 234 goto Shift 242 HexLiteral Shift 50

ID shift 152 IF shift 153 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 156 RBRACE reduce 103 CASE reduce 103 CHAR reduce 64 DEFAULT reduce 103 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 160 normal_stm goto 161 value goto 121 stm_list goto 242 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	Id Shift 245 if Shift 247 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 274 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 '}' Reduce 102 case Reduce 102 default Reduce 102 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 281 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 284 <Stm List> Goto 285 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283
state 161 stm : normal_stm . (82) . reduce 82	State 281 <Stm> ::= <Normal Stm> • '-' Reduce 81 '--' Reduce 81 '!' Reduce 81 '&' Reduce 81 '(' Reduce 81 '*' Reduce 81 ';' Reduce 81 '{' Reduce 81 '}' Reduce 81 '~' Reduce 81

	'++' Reduce 81 auto Reduce 81 break Reduce 81 case Reduce 81 char Reduce 81 CharLiteral Reduce 81 const Reduce 81 continue Reduce 81 DecLiteral Reduce 81 default Reduce 81 do Reduce 81 double Reduce 81 enum Reduce 81 extern Reduce 81 float Reduce 81 FloatLiteral Reduce 81 for Reduce 81 goto Reduce 81 HexLiteral Reduce 81 Id Reduce 81 if Reduce 81 int Reduce 81 long Reduce 81 OctLiteral Reduce 81 register Reduce 81 return Reduce 81 short Reduce 81 signed Reduce 81 sizeof Reduce 81 static Reduce 81 StringLiteral Reduce 81 struct Reduce 81 switch Reduce 81 union Reduce 81 unsigned Reduce 81 void Reduce 81 volatile Reduce 81 while Reduce 81
state 162 block : LBRACE stm_list . RBRACE (101) RBRACE shift 243 . error	State 314 <Block> ::= '{' <Stm List> • '}' '}' Shift 315
state 163 param : CONST type ID . (18) . reduce 18	State 319 <Param> ::= const <Type> Id • ')' Reduce 17 ',' Reduce 17
state 164 id_list : ID COMMA id_list . (22) . reduce 22	State 322 <Id List> ::= Id ',' <Id List> • ')' Reduce 21
state 165 func_proto : func_id LPARAN types RPARAN SEMI . (10) . reduce 10	State 342 <Func Proto> ::= <Func ID> '(' <Types> ')' ';' • (EOF) Reduce 9 auto Reduce 9 char Reduce 9 const Reduce 9 double Reduce 9 enum Reduce 9 extern Reduce 9

	float Reduce 9 Id Reduce 9 int Reduce 9 long Reduce 9 register Reduce 9 short Reduce 9 signed Reduce 9 static Reduce 9 struct Reduce 9 typedef Reduce 9 union Reduce 9 unsigned Reduce 9 void Reduce 9 volatile Reduce 9
state 166 func_proto : func_id LPARAN params RPARAN SEMI . (11) . reduce 11	State 334 <Func Proto> ::= <Func ID> '(' <Params> ')' ';' • (EOF) Reduce 10 auto Reduce 10 char Reduce 10 const Reduce 10 double Reduce 10 enum Reduce 10 extern Reduce 10 float Reduce 10 Id Reduce 10 int Reduce 10 long Reduce 10 register Reduce 10 short Reduce 10 signed Reduce 10 static Reduce 10 struct Reduce 10 typedef Reduce 10 union Reduce 10 unsigned Reduce 10 void Reduce 10 volatile Reduce 10
state 167 func_decl : func_id LPARAN params RPARAN block . (13) . reduce 13	State 335 <Func Decl> ::= <Func ID> '(' <Params> ')' <Block> • (EOF) Reduce 12 auto Reduce 12 char Reduce 12 const Reduce 12 double Reduce 12 enum Reduce 12 extern Reduce 12 float Reduce 12 Id Reduce 12 int Reduce 12 long Reduce 12 register Reduce 12 short Reduce 12 signed Reduce 12 static Reduce 12 struct Reduce 12 typedef Reduce 12 union Reduce 12 unsigned Reduce 12 void Reduce 12 volatile Reduce 12

state 168 func_decl : func_id LPARAN id_list RPARAN struct_def . block (14) LBRACE shift 92 . error block goto 244	State 325 <Func Decl> ::= <Func ID> '(' <Id List> ')' <Struct Def> • <Block> '{' Shift 227 <Block> Goto 326
state 169 params : param COMMA params . (16) . reduce 16	State 329 <Params> ::= <Param> ',' <Params> • ')' Reduce 15
state 170 param : type . ID (19) ID shift 101 . error	State 330 <Param> ::= <Type> • Id Id Shift 331
state 171 types : type COMMA types . (20) . reduce 20	State 339 <Types> ::= <Type> ',' <Types> • ')' Reduce 19
state 172 types : type . COMMA types (20) types : type . (21) COMMA shift 100 RPARAN reduce 21	State 338 <Types> ::= <Type> • ',' <Types> <Types> ::= <Type> • ',' Shift 337 ')' Reduce 20
state 173 op_unary : MINUS op_unary . (149) . reduce 149	State 174 <Op Unary> ::= '-' <Op Unary> • '-' Reduce 148 '!=' Reduce 148 '%' Reduce 148 '&' Reduce 148 '&&' Reduce 148 '&=' Reduce 148 ')' Reduce 148 '*' Reduce 148 '*=' Reduce 148 ',' Reduce 148 '/' Reduce 148 '/=' Reduce 148 ':' Reduce 148 ';' Reduce 148 '?' Reduce 148 ']' Reduce 148 '^' Reduce 148 '^=' Reduce 148 ' ' Reduce 148 ' ' Reduce 148 ' =' Reduce 148 '+' Reduce 148 '+=' Reduce 148 '<' Reduce 148 '<<' Reduce 148 '<=' Reduce 148 '<=' Reduce 148 '=' Reduce 148 '==' Reduce 148 '>' Reduce 148 '>=' Reduce 148

	'>>' Reduce 148 '>>=' Reduce 148
state 174 op_unary : MINUSMINUS op_unary . (153) . reduce 153	State 173 <Op Unary> ::= '--' <Op Unary> • '-' Reduce 152 '!=' Reduce 152 '%' Reduce 152 '&' Reduce 152 '&&' Reduce 152 '&=' Reduce 152 ')' Reduce 152 '*' Reduce 152 '*=' Reduce 152 ',' Reduce 152 '/' Reduce 152 '/=' Reduce 152 ':' Reduce 152 ';' Reduce 152 '?' Reduce 152 ']' Reduce 152 '^' Reduce 152 '^=' Reduce 152 ' ' Reduce 152 ' ' Reduce 152 ' =' Reduce 152 '+' Reduce 152 '+=' Reduce 152 '<' Reduce 152 '<<' Reduce 152 '<=' Reduce 152 '=' Reduce 152 '-=' Reduce 152 '==' Reduce 152 '>' Reduce 152 '>=' Reduce 152 '>>' Reduce 152 '>>=' Reduce 152
state 175 op_unary : EXCLAM op_unary . (147) . reduce 147	State 172 <Op Unary> ::= '!' <Op Unary> • '-' Reduce 146 '!=' Reduce 146 '%' Reduce 146 '&' Reduce 146 '&&' Reduce 146 '&=' Reduce 146 ')' Reduce 146 '*' Reduce 146 '*=' Reduce 146 ',' Reduce 146 '/' Reduce 146 '/=' Reduce 146 ':' Reduce 146 ';' Reduce 146 '?' Reduce 146 ']' Reduce 146 '^' Reduce 146 '^=' Reduce 146 ' ' Reduce 146 ' ' Reduce 146 ' =' Reduce 146 '+' Reduce 146

	'+=' Reduce 146 '<' Reduce 146 '<<' Reduce 146 '<=' Reduce 146 '<=' Reduce 146 '=' Reduce 146 '-' Reduce 146 '==' Reduce 146 '>' Reduce 146 '>=' Reduce 146 '>>' Reduce 146 '>=' Reduce 146
state 176 op_unary : AMP op_unary . (151) . reduce 151	State 171 <Op Unary> ::= '&' <Op Unary> • '-' Reduce 150 '!=' Reduce 150 '%' Reduce 150 '&' Reduce 150 '&&' Reduce 150 '&=' Reduce 150 ')' Reduce 150 '*' Reduce 150 '*=' Reduce 150 ',' Reduce 150 '/' Reduce 150 '/' Reduce 150 ':' Reduce 150 ';' Reduce 150 '?' Reduce 150 ']' Reduce 150 '^' Reduce 150 '^=' Reduce 150 ' ' Reduce 150 ' ' Reduce 150 ' =' Reduce 150 '+' Reduce 150 '+=' Reduce 150 '<' Reduce 150 '<<' Reduce 150 '<=' Reduce 150 '<=' Reduce 150 '=' Reduce 150 '-' Reduce 150 '==' Reduce 150 '>' Reduce 150 '>=' Reduce 150 '>>' Reduce 150 '>=' Reduce 150
state 177 op_unary : LPARAN type . RPARAN op_unary (156) RPARAN shift 245 . error	State 168 <Op Unary> ::= '(' <Type> • ')' <Op Unary> ')' Shift 169
state 178 expr : expr . COMMA op_assign (104) value : LPARAN expr . RPARAN (173) RPARAN shift 246 COMMA shift 194 . error	State 85 <Value> ::= '(' <Expr> • ')' <Op Assign> ')' Shift 86 ',' Shift 76
state 179	State 167

<pre> op_unary : TIMES op_unary . (150) . reduce 150 </pre>	<pre> <Op Unary> ::= '*' <Op Unary> • '-' Reduce 149 '!=' Reduce 149 '%' Reduce 149 '&' Reduce 149 '&&' Reduce 149 '&=' Reduce 149 ')' Reduce 149 '*' Reduce 149 '*=' Reduce 149 ',' Reduce 149 '/' Reduce 149 '/' Reduce 149 ':' Reduce 149 ';' Reduce 149 '?' Reduce 149 ']' Reduce 149 '^' Reduce 149 '^=' Reduce 149 ' ' Reduce 149 ' ' Reduce 149 ' =' Reduce 149 '+' Reduce 149 '+=' Reduce 149 '<' Reduce 149 '<<' Reduce 149 '<=' Reduce 149 '=' Reduce 149 '==' Reduce 149 '>' Reduce 149 '>=' Reduce 149 '>>' Reduce 149 '>=' Reduce 149 </pre>
<pre> state 180 op_unary : TILDE op_unary . (148) . reduce 148 </pre>	<pre> State 166 <Op Unary> ::= '~' <Op Unary> • '-' Reduce 147 '!=' Reduce 147 '%' Reduce 147 '&' Reduce 147 '&&' Reduce 147 '&=' Reduce 147 ')' Reduce 147 '*' Reduce 147 '*=' Reduce 147 ',' Reduce 147 '/' Reduce 147 '/' Reduce 147 ':' Reduce 147 ';' Reduce 147 '?' Reduce 147 ']' Reduce 147 '^' Reduce 147 '^=' Reduce 147 ' ' Reduce 147 ' ' Reduce 147 ' =' Reduce 147 '+' Reduce 147 '+=' Reduce 147 '<' Reduce 147 '<<' Reduce 147 </pre>

	'<=' Reduce 147 '<=' Reduce 147 '=' Reduce 147 '-' Reduce 147 '==' Reduce 147 '>' Reduce 147 '>=' Reduce 147 '>>' Reduce 147 '>>=' Reduce 147
state 181 op_unary : PLUSPLUS op_unary . (152) . reduce 152	State 165 <Op Unary> ::= '++' <Op Unary> • '-' Reduce 151 '!=' Reduce 151 '%' Reduce 151 '&' Reduce 151 '&&' Reduce 151 '&=' Reduce 151 ')' Reduce 151 '*' Reduce 151 '*=' Reduce 151 ',' Reduce 151 '/' Reduce 151 '/' Reduce 151 ':' Reduce 151 ';' Reduce 151 '?' Reduce 151 ']' Reduce 151 '^' Reduce 151 '^=' Reduce 151 ' ' Reduce 151 ' ' Reduce 151 ' =' Reduce 151 '+' Reduce 151 '+=' Reduce 151 '<' Reduce 151 '<<' Reduce 151 '<=' Reduce 151 '<=' Reduce 151 '=' Reduce 151 '-' Reduce 151 '==' Reduce 151 '>' Reduce 151 '>=' Reduce 151 '>>' Reduce 151 '>>=' Reduce 151
state 182 value : ID LPARAN . expr RPARAN (170) value : ID LPARAN . RPARAN (171) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 RPARAN shift 247 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115	State 52 <Value> ::= Id '(' • <Expr> ')' ' <Value> ::= Id '(' • ')' ' '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 ')' Shift 53 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51

OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 expr goto 248 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Expr> Goto 74 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 183 op_unary : SIZEOF LPARAN . type RPARAN (157) op_unary : SIZEOF LPARAN . ID pointers RPARAN (158) sign : . (64) ENUM shift 31 ID shift 249 SIGNED shift 7 STRUCT shift 32 UNION shift 33 UNSIGNED shift 12 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 type goto 250 base goto 26 sign goto 27	State 56 <Op Unary> ::= sizeof '(' • <Type> ')' ' <Op Unary> ::= sizeof '(' • Id • <Pointers> ')' ' <Sign> ::= • enum Shift 24 Id Shift 57 signed Shift 20 struct Shift 26 union Shift 28 unsigned Shift 30 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Sign> Goto 60 <Type> Goto 71
state 184 op_assign : op_if AMPEQ . op_assign (112) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117	State 129 <Op Assign> ::= <Op If> '&=' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 Decliteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55

<p>STRINGLITERAL shift 118 . error</p> <p>op_if goto 119 value goto 121 op_assign goto 251 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134</p>	<p>StringLiteral Shift 73</p> <p><Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 130 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96</p>
<p>state 185</p> <p>op_assign : op_if TIMESEQ . op_assign (109)</p> <p>MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error</p> <p>op_if goto 119 value goto 121 op_assign goto 252 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134</p>	<p>State 138</p> <p><Op Assign> ::= <Op If> '*' • <Op Assign></p> <p>'-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 Decliteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73</p> <p><Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 139 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96</p>
<p>state 186</p> <p>op_assign : op_if DIVEQ . op_assign (110)</p> <p>MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106</p>	<p>State 140</p> <p><Op Assign> ::= <Op If> '/=' • <Op Assign></p> <p>'-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43</p>

<p> TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error </p> <p> op_if goto 119 value goto 121 op_assign goto 253 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </p>	<p> '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 </p> <p> <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 141 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </p>
<p> state 187 op_assign : op_if CARETEQ . op_assign (111) </p> <p> MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error </p> <p> op_if goto 119 value goto 121 op_assign goto 254 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 </p>	<p> State 142 <Op Assign> ::= <Op If> '^=' • <Op Assign> </p> <p> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 </p> <p> <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 143 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </p>

<pre> op_pointer goto 134 state 188 op_assign : op_if PIPEEQ . op_assign (113) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 value goto 121 op_assign goto 255 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 144 <Op Assign> ::= <Op If> ' ' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 145 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 189 op_assign : op_if PLUSEQ . op_assign (107) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 value goto 121 op_assign goto 256 op_or goto 123 </pre>	<pre> State 146 <Op Assign> ::= <Op If> '+=' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 147 <Op BinAND> Goto 89 <Op BinOR> Goto 127 </pre>

op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	<Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 190 op_assign : op_if LTLTEQ . op_assign (115) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 value goto 121 op_assign goto 257 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 148 <Op Assign> ::= <Op If> '<=>' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 149 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 191 op_assign : op_if EQ . op_assign (106) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115	State 150 <Op Assign> ::= <Op If> '=' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50

OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 value goto 121 op_assign goto 258 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 151 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 192 op_assign : op_if MINUSEQ . op_assign (108) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 value goto 121 op_assign goto 259 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 152 <Op Assign> ::= <Op If> '-' = ' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 153 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 193 op_assign : op_if GTGTEQ . op_assign (114) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104	State 154 <Op Assign> ::= <Op If> '>=' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41

<pre> AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 value goto 121 op_assign goto 260 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 155 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 194 expr : expr COMMA . op_assign (104) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 value goto 121 op_assign goto 261 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 </pre>	<pre> State 76 <Expr> ::= <Expr> ',' • <Op Assign> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 164 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 </pre>

op_unary goto 133 op_pointer goto 134	<Value> Goto 96
state 195 array : LBRACKET expr RBRACKET . (36) . reduce 36	State 177 <Array> ::= '[' <Expr> ']' • ',' Reduce 35 ';' Reduce 35 '=' Reduce 35
state 196 op_if : op_or QUESTION . op_if COLON op_if (117) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 262 value goto 121 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 132 <Op If> ::= <Op Or> '?' • <Op If> ':' <Op If> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 133 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 197 op_or : op_or PIPEPIPE . op_and (119) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118	State 136 <Op Or> ::= <Op Or> ' ' • <Op And> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73

<pre> . error value goto 121 op_and goto 263 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Op Add> Goto 77 <Op And> Goto 137 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 198 op_and : op_and AMPAMP . op_binor (121) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_binor goto 264 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 88 <Op And> ::= <Op And> '&&' • <Op BinOR> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op BinAND> Goto 89 <Op BinOR> Goto 119 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 199 op_binor : op_binor PIPE . op_binxor (123) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 </pre>	<pre> State 120 <Op BinOR> ::= <Op BinOR> ' ' • <Op BinXOR> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 </pre>

OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_binxor goto 265 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op BinAND> Goto 89 <Op BinXOR> Goto 121 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 200 op_binxor : op_binxor CARET . op_binand (125) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_binand goto 266 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 122 <Op BinXOR> ::= <Op BinXOR> '^' • <Op BinAND> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op BinAND> Goto 123 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 201 op_binand : op_binand AMP . op_equate (127) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116	State 90 <Op BinAND> ::= <Op BinAND> '&' • <Op Equate> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54

<pre> SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_equate goto 267 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op Compare> Goto 91 <Op Equate> Goto 113 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 202 op_equate : op_equate EXCLAMEQ . op_compare (130) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_compare goto 268 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 114 <Op Equate> ::= <Op Equate> '!=' • <Op Compare> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op Compare> Goto 115 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 203 op_equate : op_equate EQEQ . op_compare (129) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 </pre>	<pre> State 117 <Op Equate> ::= <Op Equate> '==' • <Op Compare> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op Compare> Goto 118 </pre>

op_compare goto 269 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	<Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 204 op_compare : op_compare LT . op_shift (132) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_shift goto 270 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 92 <Op Compare> ::= <Op Compare> '<' • <Op Shift> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 99 <Op Unary> Goto 104 <Value> Goto 96
state 205 op_compare : op_compare LTEQ . op_shift (134) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_shift goto 271 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 107 <Op Compare> ::= <Op Compare> '<=' • <Op Shift> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 108 <Op Unary> Goto 104 <Value> Goto 96
state 206	State 109

<pre> op_compare : op_compare GT . op_shift (133) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_shift goto 272 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Op Compare> ::= <Op Compare> '>' • <Op Shift> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 110 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 207 op_compare : op_compare GTEQ . op_shift (135) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_shift goto 273 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 111 <Op Compare> ::= <Op Compare> '>=' • <Op Shift> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Shift> Goto 112 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 208 op_shift : op_shift LTLT . op_add (137) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 </pre>	<pre> State 100 <Op Shift> ::= <Op Shift> '<<' • <Op Add> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 </pre>

TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_add goto 274 op_mult goto 132 op_unary goto 133 op_pointer goto 134	'~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 101 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Unary> Goto 104 <Value> Goto 96
state 209 op_shift : op_shift GTGT . op_add (138) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_add goto 275 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 105 <Op Shift> ::= <Op Shift> '>>' • <Op Add> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 106 <Op Mult> Goto 93 <Op Pointer> Goto 81 <Op Unary> Goto 104 <Value> Goto 96
state 210 op_add : op_add MINUS . op_mult (141) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error	State 78 <Op Add> ::= <Op Add> '-' • <Op Mult> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Mult> Goto 79

<pre> value goto 121 op_mult goto 276 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Op Pointer> Goto 81 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 211 op_add : op_add PLUS . op_mult (140) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_mult goto 277 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 102 <Op Add> ::= <Op Add> '+' • <Op Mult> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DeclLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Mult> Goto 103 <Op Pointer> Goto 81 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 212 op_mult : op_mult PERCENT . op_unary (145) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 278 op_pointer goto 134 </pre>	<pre> State 80 <Op Mult> ::= <Op Mult> '%' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DeclLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 163 <Value> Goto 96 </pre>
<pre> state 213 op_mult : op_mult TIMES . op_unary (143) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 </pre>	<pre> State 94 <Op Mult> ::= <Op Mult> '*' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 </pre>

<p> TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 279 op_pointer goto 134 </p>	<p> '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 95 <Value> Goto 96 </p>
<p> state 214 op_mult : op_mult DIV . op_unary (144) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 280 op_pointer goto 134 </p>	<p> State 97 <Op Mult> ::= <Op Mult> '/' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 98 <Value> Goto 96 </p>
<p> state 215 op_unary : op_pointer MINUSMINUS . (155) . reduce 155 </p>	<p> State 82 <Op Unary> ::= <Op Pointer> '--' • '-' Reduce 154 '!=' Reduce 154 '%' Reduce 154 '&' Reduce 154 '&&' Reduce 154 '&=' Reduce 154 ')' Reduce 154 '*' Reduce 154 '*=' Reduce 154 ',' Reduce 154 '/' Reduce 154 '/' Reduce 154 ':' Reduce 154 ';' Reduce 154 '?' Reduce 154 ']' Reduce 154 '^' Reduce 154 '^=' Reduce 154 ' ' Reduce 154 ' ' Reduce 154 ' =' Reduce 154 </p>

. reduce 39	';' Reduce 38
state 222 var_item : pointers var . (41) . reduce 41	State 185 <Var Item> ::= <Pointers> <Var> • , ' Reduce 40 , ' Reduce 40
state 223 var_decl : mod type var var_list SEMI . (31) . reduce 31	State 189 <Var Decl> ::= <Mod> <Type> <Var> <Var List> ';' • (EOF) Reduce 30 '-' Reduce 30 '--' Reduce 30 '!' Reduce 30 '&' Reduce 30 '(' Reduce 30 '*' Reduce 30 ';' Reduce 30 '{' Reduce 30 '}' Reduce 30 '~' Reduce 30 '++' Reduce 30 auto Reduce 30 break Reduce 30 case Reduce 30 char Reduce 30 CharLiteral Reduce 30 const Reduce 30 continue Reduce 30 Decliteral Reduce 30 default Reduce 30 do Reduce 30 double Reduce 30 enum Reduce 30 extern Reduce 30 float Reduce 30 FloatLiteral Reduce 30 for Reduce 30 goto Reduce 30 HexLiteral Reduce 30 Id Reduce 30 if Reduce 30 int Reduce 30 long Reduce 30 OctLiteral Reduce 30 register Reduce 30 return Reduce 30 short Reduce 30 signed Reduce 30 sizeof Reduce 30 static Reduce 30 StringLiteral Reduce 30 struct Reduce 30 switch Reduce 30 typedef Reduce 30 union Reduce 30 unsigned Reduce 30 void Reduce 30 volatile Reduce 30 while Reduce 30
state 224 enum_val : ID EQ DECLITERAL . (54)	State 8 <Enum Val> ::= Id '=' Decliteral •

. reduce 54	' ,' Reduce 53 '}' Reduce 53
state 225 enum_val : ID EQ HEXLITERAL . (53) . reduce 53	State 9 <Enum Val> ::= Id '=' HexLiteral • ' ,' Reduce 52 '}' Reduce 52
state 226 enum_val : ID EQ OCTLITERAL . (52) . reduce 52	State 10 <Enum Val> ::= Id '=' OctLiteral • ' ,' Reduce 51 '}' Reduce 51
state 227 enum_decl : ENUM ID LBRACE enum_def RBRACE SEMI . (48) . reduce 48	State 13 <Enum Decl> ::= enum Id '{' <Enum Def> '}' ';' • (EOF) Reduce 47 auto Reduce 47 char Reduce 47 const Reduce 47 double Reduce 47 enum Reduce 47 extern Reduce 47 float Reduce 47 Id Reduce 47 int Reduce 47 long Reduce 47 register Reduce 47 short Reduce 47 signed Reduce 47 static Reduce 47 struct Reduce 47 typedef Reduce 47 union Reduce 47 unsigned Reduce 47 void Reduce 47 volatile Reduce 47
state 228 enum_def : enum_val COMMA enum_def . (49) . reduce 49	State 16 <Enum Def> ::= <Enum Val> ',' <Enum Def> • '}' Reduce 48
state 229 struct_decl : STRUCT ID LBRACE struct_def RBRACE SEMI . (27) . reduce 27	State 208 <Struct Decl> ::= struct Id '{' <Struct Def> '} ' ';' • (EOF) Reduce 26 auto Reduce 26 char Reduce 26 const Reduce 26 double Reduce 26 enum Reduce 26 extern Reduce 26 float Reduce 26 Id Reduce 26 int Reduce 26 long Reduce 26 register Reduce 26 short Reduce 26 signed Reduce 26 static Reduce 26 struct Reduce 26 typedef Reduce 26

	union Reduce 26 unsigned Reduce 26 void Reduce 26 volatile Reduce 26
state 230 union_decl : UNION ID LBRACE struct_def RBRACE SEMI . (28) . reduce 28	State 218 <Union Decl> ::= union Id '{' <Struct Def> '}' ';' • (EOF) Reduce 27 auto Reduce 27 char Reduce 27 const Reduce 27 double Reduce 27 enum Reduce 27 extern Reduce 27 float Reduce 27 Id Reduce 27 int Reduce 27 long Reduce 27 register Reduce 27 short Reduce 27 signed Reduce 27 static Reduce 27 struct Reduce 27 typedef Reduce 27 union Reduce 27 unsigned Reduce 27 void Reduce 27 volatile Reduce 27
state 231 normal_stm : BREAK SEMI . (92) . reduce 92	State 230 <Normal Stm> ::= break ';' • '-' Reduce 91 '--' Reduce 91 '!' Reduce 91 '&' Reduce 91 '(' Reduce 91 '*' Reduce 91 ';' Reduce 91 '{' Reduce 91 '}' Reduce 91 '~' Reduce 91 '++' Reduce 91 auto Reduce 91 break Reduce 91 case Reduce 91 char Reduce 91 CharLiteral Reduce 91 const Reduce 91 continue Reduce 91 Decliteral Reduce 91 default Reduce 91 do Reduce 91 double Reduce 91 else Reduce 91 enum Reduce 91 extern Reduce 91 float Reduce 91 FloatLiteral Reduce 91 for Reduce 91 goto Reduce 91 HexLiteral Reduce 91 Id Reduce 91 if Reduce 91

	int Reduce 91 long Reduce 91 OctLiteral Reduce 91 register Reduce 91 return Reduce 91 short Reduce 91 signed Reduce 91 sizeof Reduce 91 static Reduce 91 StringLiteral Reduce 91 struct Reduce 91 switch Reduce 91 union Reduce 91 unsigned Reduce 91 void Reduce 91 volatile Reduce 91 while Reduce 91
state 232 normal_stm : CONTINUE SEMI . (93) . reduce 93	State 232 <Normal Stm> ::= continue ';' • '-' Reduce 92 '--' Reduce 92 '!' Reduce 92 '&' Reduce 92 '(' Reduce 92 '*' Reduce 92 ';' Reduce 92 '{' Reduce 92 '}' Reduce 92 '~' Reduce 92 '++' Reduce 92 auto Reduce 92 break Reduce 92 case Reduce 92 char Reduce 92 CharLiteral Reduce 92 const Reduce 92 continue Reduce 92 DecLiteral Reduce 92 default Reduce 92 do Reduce 92 double Reduce 92 else Reduce 92 enum Reduce 92 extern Reduce 92 float Reduce 92 FloatLiteral Reduce 92 for Reduce 92 goto Reduce 92 HexLiteral Reduce 92 Id Reduce 92 if Reduce 92 int Reduce 92 long Reduce 92 OctLiteral Reduce 92 register Reduce 92 return Reduce 92 short Reduce 92 signed Reduce 92 sizeof Reduce 92 static Reduce 92 StringLiteral Reduce 92 struct Reduce 92 switch Reduce 92

	union Reduce 92 unsigned Reduce 92 void Reduce 92 volatile Reduce 92 while Reduce 92
state 233 normal_stm : DO stm . WHILE LPARAN expr RPARAN (87) WHILE shift 285 . error	State 309 <Normal Stm> ::= do <Stm> • while '(' <Expr> ')'
state 234 stm : FOR LPARAN . arg SEMI arg SEMI arg RPARAN stm (81) arg : . (97) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 SEMI reduce 97 op_if goto 119 expr goto 286 arg goto 287 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 235 <Stm> ::= for '(' • <Arg> ';' <Arg> ';' <Arg> ')' <Stm> <Arg> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 Decliteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 ';' Reduce 96 <Arg> Goto 236 <Expr> Goto 306 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 235 normal_stm : GOTO ID . SEMI (91) SEMI shift 288 . error	State 243 <Normal Stm> ::= goto Id • ';'
state 236 stm : ID COLON . (77) . reduce 77	State 246 <Stm> ::= Id ':' • '-' Reduce 76 '--' Reduce 76 '!' Reduce 76 '&' Reduce 76

	'(' Reduce 76 '*' Reduce 76 ';' Reduce 76 {' Reduce 76 '}' Reduce 76 '~' Reduce 76 '++' Reduce 76 auto Reduce 76 break Reduce 76 case Reduce 76 char Reduce 76 CharLiteral Reduce 76 const Reduce 76 continue Reduce 76 DecLiteral Reduce 76 default Reduce 76 do Reduce 76 double Reduce 76 enum Reduce 76 extern Reduce 76 float Reduce 76 FloatLiteral Reduce 76 for Reduce 76 goto Reduce 76 HexLiteral Reduce 76 Id Reduce 76 if Reduce 76 int Reduce 76 long Reduce 76 OctLiteral Reduce 76 register Reduce 76 return Reduce 76 short Reduce 76 signed Reduce 76 sizeof Reduce 76 static Reduce 76 StringLiteral Reduce 76 struct Reduce 76 switch Reduce 76 union Reduce 76 unsigned Reduce 76 void Reduce 76 volatile Reduce 76 while Reduce 76
state 237 stm : IF LPARAN . expr RPARAN stm (78) stm : IF LPARAN . expr RPARAN then_stm ELSE stm (79) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117	State 248 <Stm> ::= if '(' • <Expr> ')' <Stm> <Stm> ::= if '(' • <Expr> ')' <Then Stm> else <Stm> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55

<p>STRINGLITERAL shift 118 . error</p> <p>op_if goto 119 expr goto 289 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134</p>	<p>StringLiteral Shift 73</p> <p><Expr> Goto 249 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96</p>
<p>state 238 normal_stm : RETURN expr . SEMI (94) expr : expr . COMMA op_assign (104)</p> <p>COMMA shift 194 SEMI shift 290 . error</p>	<p>State 264 <Normal Stm> ::= return <Expr> . ';' ' <Expr> ::= <Expr> . ',' <Op Assign></p> <p>',' Shift 76 ';' Shift 265</p>
<p>state 239 normal_stm : SWITCH LPARAN . expr RPARAN LBRACE case_stms RBRACE (88)</p> <p>MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error</p> <p>op_if goto 119 expr goto 291 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133</p>	<p>State 267 <Normal Stm> ::= switch '(' . <Expr> ')' '{' <Case Stms> '}'</p> <p>'-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 Decliteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73</p> <p><Expr> Goto 268 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96</p>

<pre> op_pointer goto 134 state 240 stm : WHILE LPARAN . expr RPARAN stm (80) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 expr goto 292 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 275 <Stm> ::= while '(' • <Expr> ')' <Stm> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Expr> Goto 276 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 241 normal_stm : expr SEMI . (90) . reduce 90 </pre>	<pre> State 280 <Normal Stm> ::= <Expr> ';' • '-' Reduce 89 '--' Reduce 89 '!' Reduce 89 '&' Reduce 89 '(' Reduce 89 '*' Reduce 89 ';' Reduce 89 '{' Reduce 89 '}' Reduce 89 '~' Reduce 89 '++' Reduce 89 auto Reduce 89 break Reduce 89 case Reduce 89 char Reduce 89 CharLiteral Reduce 89 const Reduce 89 continue Reduce 89 DecLiteral Reduce 89 default Reduce 89 do Reduce 89 double Reduce 89 </pre>

	else Reduce 89 enum Reduce 89 extern Reduce 89 float Reduce 89 FloatLiteral Reduce 89 for Reduce 89 goto Reduce 89 HexLiteral Reduce 89 Id Reduce 89 if Reduce 89 int Reduce 89 long Reduce 89 OctLiteral Reduce 89 register Reduce 89 return Reduce 89 short Reduce 89 signed Reduce 89 sizeof Reduce 89 static Reduce 89 StringLiteral Reduce 89 struct Reduce 89 switch Reduce 89 union Reduce 89 unsigned Reduce 89 void Reduce 89 volatile Reduce 89 while Reduce 89
state 242 stm_list : stm stm_list . (102) . reduce 102	State 285 <Stm List> ::= <Stm> <Stm List> • '}' Reduce 101 case Reduce 101 default Reduce 101
state 243 block : LBRACE stm_list RBRACE . (101) . reduce 101	State 315 <Block> ::= '{' <Stm List> '}' • (EOF) Reduce 100 '-' Reduce 100 '--' Reduce 100 '!' Reduce 100 '&' Reduce 100 '(' Reduce 100 '*' Reduce 100 ';' Reduce 100 '{' Reduce 100 '}' Reduce 100 '~' Reduce 100 '++' Reduce 100 auto Reduce 100 break Reduce 100 case Reduce 100 char Reduce 100 CharLiteral Reduce 100 const Reduce 100 continue Reduce 100 Decliteral Reduce 100 default Reduce 100 do Reduce 100 double Reduce 100 else Reduce 100 enum Reduce 100 extern Reduce 100 float Reduce 100 FloatLiteral Reduce 100

	for Reduce 100 goto Reduce 100 HexLiteral Reduce 100 Id Reduce 100 if Reduce 100 int Reduce 100 long Reduce 100 OctLiteral Reduce 100 register Reduce 100 return Reduce 100 short Reduce 100 signed Reduce 100 sizeof Reduce 100 static Reduce 100 StringLiteral Reduce 100 struct Reduce 100 switch Reduce 100 typedef Reduce 100 union Reduce 100 unsigned Reduce 100 void Reduce 100 volatile Reduce 100 while Reduce 100
state 244 func_decl : func_id LPARAN id_list RPARAN struct_def block . (14) . reduce 14	State 326 <Func Decl> ::= <Func ID> '(' <Id List> ')' <Struct Def> <Block> • (EOF) Reduce 13 auto Reduce 13 char Reduce 13 const Reduce 13 double Reduce 13 enum Reduce 13 extern Reduce 13 float Reduce 13 Id Reduce 13 int Reduce 13 long Reduce 13 register Reduce 13 short Reduce 13 signed Reduce 13 static Reduce 13 struct Reduce 13 typedef Reduce 13 union Reduce 13 unsigned Reduce 13 void Reduce 13 volatile Reduce 13
state 245 op_unary : LPARAN type RPARAN . op_unary (156) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113	State 169 <Op Unary> ::= '(' <Type> ')' • <Op Unary> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 Decliteral Shift 48 FloatLiteral Shift 49

<pre> HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error value goto 121 op_unary goto 293 op_pointer goto 134 </pre>	<pre> HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Pointer> Goto 81 <Op Unary> Goto 170 <Value> Goto 96 </pre>
<pre> state 246 value : LPARAN expr RPARAN . (173) . reduce 173 </pre>	<pre> State 86 <Value> ::= '(' <Expr> ')' • '-' Reduce 172 '--' Reduce 172 '!=' Reduce 172 '%' Reduce 172 '&' Reduce 172 '&&' Reduce 172 '&=' Reduce 172 ')' Reduce 172 '*' Reduce 172 '*=' Reduce 172 ',' Reduce 172 '.' Reduce 172 '/' Reduce 172 '/' Reduce 172 ':' Reduce 172 ';' Reduce 172 '?' Reduce 172 '[' Reduce 172 ']' Reduce 172 '^' Reduce 172 '^=' Reduce 172 ' ' Reduce 172 ' ' Reduce 172 ' =' Reduce 172 '+' Reduce 172 '++' Reduce 172 '+=' Reduce 172 '<' Reduce 172 '<<' Reduce 172 '<=' Reduce 172 '=' Reduce 172 '==' Reduce 172 '>' Reduce 172 '>' Reduce 172 '>=' Reduce 172 '>>' Reduce 172 '>=' Reduce 172 </pre>
<pre> state 247 value : ID LPARAN RPARAN . (171) . reduce 171 </pre>	<pre> State 53 <Value> ::= Id '(' ')' • '-' Reduce 170 '--' Reduce 170 '!=' Reduce 170 '%' Reduce 170 '&' Reduce 170 '&&' Reduce 170 '&=' Reduce 170 ')' Reduce 170 '*' Reduce 170 </pre>

	<pre> '*=' Reduce 170 ',' Reduce 170 '.' Reduce 170 '/' Reduce 170 '/=' Reduce 170 ':' Reduce 170 ';' Reduce 170 '? ' Reduce 170 '[' Reduce 170 ']' Reduce 170 '^' Reduce 170 '^=' Reduce 170 ' ' Reduce 170 ' ' Reduce 170 ' =' Reduce 170 '+' Reduce 170 '++' Reduce 170 '+=' Reduce 170 '<' Reduce 170 '<<' Reduce 170 '<=' Reduce 170 '<=' Reduce 170 '=' Reduce 170 '-=' Reduce 170 '==' Reduce 170 '>' Reduce 170 '->' Reduce 170 '>=' Reduce 170 '>>' Reduce 170 '>>=' Reduce 170 </pre>
<pre> state 248 expr : expr . COMMA op_assign (104) value : ID LPARAN expr . RPARAN (170) RPARAN shift 294 COMMA shift 194 . error </pre>	<pre> State 74 <Value> ::= Id '(' <Expr> . ')' <Expr> ::= <Expr> . ',' <Op Assign> ')' Shift 75 ',' Shift 76 </pre>
<pre> state 249 op_unary : SIZEOF LPARAN ID . pointers RPARAN (158) pointers : . (75) TIMES shift 44 RPARAN reduce 75 pointers goto 295 </pre>	<pre> State 57 <Op Unary> ::= sizeof '(' Id . <Pointers> ')' <Pointers> ::= . '*' Shift 33 ')' Reduce 74 <Pointers> Goto 58 </pre>
<pre> state 250 op_unary : SIZEOF LPARAN type . RPARAN (157) RPARAN shift 296 . error </pre>	<pre> State 71 <Op Unary> ::= sizeof '(' <Type> . ')' ')' Shift 72 </pre>
<pre> state 251 op_assign : op_if AMPEQ op_assign . (112) . reduce 112 </pre>	<pre> State 130 <Op Assign> ::= <Op If> '&=' <Op Assign> . ')' Reduce 111 ',' Reduce 111 ';' Reduce 111 ']' Reduce 111 </pre>
<pre> state 252 op_assign : op_if TIMESEQ op_assign . (109) </pre>	<pre> State 139 <Op Assign> ::= <Op If> '*=' <Op Assign> . </pre>

. reduce 109	')' Reduce 108 ' ,' Reduce 108 ' ;' Reduce 108 ']' Reduce 108
state 253 op_assign : op_if DIVEQ op_assign . (110) . reduce 110	State 141 <Op Assign> ::= <Op If> '/=' <Op Assign> • ')' Reduce 109 ' ,' Reduce 109 ' ;' Reduce 109 ']' Reduce 109
state 254 op_assign : op_if CARETEQ op_assign . (111) . reduce 111	State 143 <Op Assign> ::= <Op If> '^=' <Op Assign> • ')' Reduce 110 ' ,' Reduce 110 ' ;' Reduce 110 ']' Reduce 110
state 255 op_assign : op_if PIPEEQ op_assign . (113) . reduce 113	State 145 <Op Assign> ::= <Op If> ' =' <Op Assign> • ')' Reduce 112 ' ,' Reduce 112 ' ;' Reduce 112 ']' Reduce 112
state 256 op_assign : op_if PLUSEQ op_assign . (107) . reduce 107	State 147 <Op Assign> ::= <Op If> '+=' <Op Assign> • ')' Reduce 106 ' ,' Reduce 106 ' ;' Reduce 106 ']' Reduce 106
state 257 op_assign : op_if LTLTEQ op_assign . (115) . reduce 115	State 149 <Op Assign> ::= <Op If> '<=' <Op Assign> • ')' Reduce 114 ' ,' Reduce 114 ' ;' Reduce 114 ']' Reduce 114
state 258 op_assign : op_if EQ op_assign . (106) . reduce 106	State 151 <Op Assign> ::= <Op If> '=' <Op Assign> • ')' Reduce 105 ' ,' Reduce 105 ' ;' Reduce 105 ']' Reduce 105
state 259 op_assign : op_if MINUSEQ op_assign . (108) . reduce 108	State 153 <Op Assign> ::= <Op If> '-=' <Op Assign> • ')' Reduce 107 ' ,' Reduce 107 ' ;' Reduce 107 ']' Reduce 107
state 260	State 155

op_assign : op_if GTGTEQ op_assign . (114) . reduce 114	<Op Assign> ::= <Op If> '>=' <Op Assign> • ')' Reduce 113 ',' Reduce 113 ';' Reduce 113 ']' Reduce 113
state 261 expr : expr COMMA op_assign . (104) . reduce 104	State 164 <Expr> ::= <Expr> ',' <Op Assign> • ')' Reduce 103 ',' Reduce 103 ';' Reduce 103 ']' Reduce 103
state 262 op_if : op_or QUESTION op_if . COLON op_if (117) COLON shift 297 . error	State 133 <Op If> ::= <Op Or> '?' <Op If> • ':' <Op If> ':' Shift 134
state 263 op_or : op_or PIPEPIPE op_and . (119) op_and : op_and . AMPAMP op_binor (121) AMPAMP shift 198 AMPEQ reduce 119 RPARAN reduce 119 TIMESEQ reduce 119 COMMA reduce 119 DIVEQ reduce 119 COLON reduce 119 SEMI reduce 119 QUESTION reduce 119 RBRACKET reduce 119 CARETEQ reduce 119 PIPEPIPE reduce 119 PIPEEQ reduce 119 PLUSEQ reduce 119 LTLTEQ reduce 119 EQ reduce 119 MINUSEQ reduce 119 GTGTEQ reduce 119	State 137 <Op Or> ::= <Op Or> ' ' <Op And> • <Op And> ::= <Op And> • '&&' <Op BinOR> '&&' Shift 88 '&=' Reduce 118 ')' Reduce 118 '*=' Reduce 118 ',' Reduce 118 '/' Reduce 118 ':' Reduce 118 ';' Reduce 118 '?' Reduce 118 ']' Reduce 118 '^=' Reduce 118 ' ' Reduce 118 ' =' Reduce 118 '+=' Reduce 118 '<=' Reduce 118 '=' Reduce 118 '-' Reduce 118 '>=' Reduce 118
state 264 op_and : op_and AMPAMP op_binor . (121) op_binor : op_binor . PIPE op_binxor (123) PIPE shift 199 AMPAMP reduce 121 AMPEQ reduce 121 RPARAN reduce 121 TIMESEQ reduce 121 COMMA reduce 121 DIVEQ reduce 121 COLON reduce 121 SEMI reduce 121 QUESTION reduce 121 RBRACKET reduce 121 CARETEQ reduce 121 PIPEPIPE reduce 121 PIPEEQ reduce 121	State 119 <Op And> ::= <Op And> '&&' <Op BinOR> • <Op BinOR> ::= <Op BinOR> • ' ' <Op BinXOR> ' ' Shift 120 '&&' Reduce 120 '&=' Reduce 120 ')' Reduce 120 '*=' Reduce 120 ',' Reduce 120 '/' Reduce 120 ':' Reduce 120 ';' Reduce 120 '?' Reduce 120 ']' Reduce 120 '^=' Reduce 120 ' ' Reduce 120 ' =' Reduce 120 '+=' Reduce 120

PLUSEQ reduce 121 LTLTEQ reduce 121 EQ reduce 121 MINUSEQ reduce 121 GTGTEQ reduce 121	'<=' Reduce 120 '=' Reduce 120 '-=' Reduce 120 '>=' Reduce 120
state 265 op_binor : op_binor PIPE op_binxor . (123) op_binxor : op_binxor . CARET op_binand (125) CARET shift 200 AMPAMP reduce 123 AMPEQ reduce 123 RPARAN reduce 123 TIMESEQ reduce 123 COMMA reduce 123 DIVEQ reduce 123 COLON reduce 123 SEMI reduce 123 QUESTION reduce 123 RBRACKET reduce 123 CARETEQ reduce 123 PIPE reduce 123 PIPEPIPE reduce 123 PIPEEQ reduce 123 PLUSEQ reduce 123 LTLTEQ reduce 123 EQ reduce 123 MINUSEQ reduce 123 GTGTEQ reduce 123	State 121 <Op BinOR> ::= <Op BinOR> ' ' <Op BinXOR> • <Op BinXOR> ::= <Op BinXOR> • '^' <Op BinAND> '^' Shift 122 '&&' Reduce 122 '&=' Reduce 122 ')' Reduce 122 '*' Reduce 122 ',' Reduce 122 '/' Reduce 122 ':' Reduce 122 ';' Reduce 122 '?' Reduce 122 ']' Reduce 122 '^=' Reduce 122 ' ' Reduce 122 ' ' Reduce 122 ' =' Reduce 122 '+=' Reduce 122 '<=' Reduce 122 '=' Reduce 122 '-=' Reduce 122 '>=' Reduce 122
state 266 op_binxor : op_binxor CARET op_binand . (125) op_binand : op_binand . AMP op_equate (127) AMP shift 201 AMPAMP reduce 125 AMPEQ reduce 125 RPARAN reduce 125 TIMESEQ reduce 125 COMMA reduce 125 DIVEQ reduce 125 COLON reduce 125 SEMI reduce 125 QUESTION reduce 125 RBRACKET reduce 125 CARET reduce 125 CARETEQ reduce 125 PIPE reduce 125 PIPEPIPE reduce 125 PIPEEQ reduce 125 PLUSEQ reduce 125 LTLTEQ reduce 125 EQ reduce 125 MINUSEQ reduce 125 GTGTEQ reduce 125	State 123 <Op BinXOR> ::= <Op BinXOR> '^' <Op BinAND> • <Op BinAND> ::= <Op BinAND> • '&' <Op Equate> '&' Shift 90 '&&' Reduce 124 '&=' Reduce 124 ')' Reduce 124 '*' Reduce 124 ',' Reduce 124 '/' Reduce 124 ':' Reduce 124 ';' Reduce 124 '?' Reduce 124 ']' Reduce 124 '^' Reduce 124 '^=' Reduce 124 ' ' Reduce 124 ' ' Reduce 124 ' =' Reduce 124 '+=' Reduce 124 '<=' Reduce 124 '=' Reduce 124 '-=' Reduce 124 '>=' Reduce 124
state 267 op_binand : op_binand AMP op_equate . (127) op_equate : op_equate . EQEQ op_compare (129)	State 113 <Op BinAND> ::= <Op BinAND> '&' <Op Equate> • <Op Equate> ::= <Op Equate> • '==' <Op Compare>

<pre> op_equate : op_equate . EXCLAMEQ op_compare (130) EXCLAMEQ shift 202 EQEQ shift 203 AMP reduce 127 AMPAMP reduce 127 AMPEQ reduce 127 RPARAN reduce 127 TIMESEQ reduce 127 COMMA reduce 127 DIVEQ reduce 127 COLON reduce 127 SEMI reduce 127 QUESTION reduce 127 RBRACKET reduce 127 CARET reduce 127 CARETEQ reduce 127 PIPE reduce 127 PIPEPIPE reduce 127 PIPEEQ reduce 127 PLUSEQ reduce 127 LTLTEQ reduce 127 EQ reduce 127 MINUSEQ reduce 127 GTGTEQ reduce 127 </pre>	<pre> <Op Equate> ::= <Op Equate> • '!=' <Op Compare> '!=' Shift 114 '==' Shift 117 '&' Reduce 126 '&&' Reduce 126 '&=' Reduce 126 ')' Reduce 126 '*=' Reduce 126 ',' Reduce 126 '/' Reduce 126 ':' Reduce 126 ';' Reduce 126 '?' Reduce 126 ']' Reduce 126 '^' Reduce 126 '^=' Reduce 126 ' ' Reduce 126 ' ' Reduce 126 ' =' Reduce 126 '+=' Reduce 126 '<=' Reduce 126 '=' Reduce 126 '-' Reduce 126 '>=' Reduce 126 </pre>
<pre> state 268 op_equate : op_equate EXCLAMEQ op_compare . (130) op_compare : op_compare . LT op_shift (132) op_compare : op_compare . GT op_shift (133) op_compare : op_compare . LTEQ op_shift (134) op_compare : op_compare . GTEQ op_shift (135) LT shift 204 LTEQ shift 205 GT shift 206 GTEQ shift 207 EXCLAMEQ reduce 130 AMP reduce 130 AMPAMP reduce 130 AMPEQ reduce 130 RPARAN reduce 130 TIMESEQ reduce 130 COMMA reduce 130 DIVEQ reduce 130 COLON reduce 130 SEMI reduce 130 QUESTION reduce 130 RBRACKET reduce 130 CARET reduce 130 CARETEQ reduce 130 PIPE reduce 130 PIPEPIPE reduce 130 PIPEEQ reduce 130 PLUSEQ reduce 130 LTLTEQ reduce 130 EQ reduce 130 MINUSEQ reduce 130 EQEQ reduce 130 </pre>	<pre> State 115 <Op Equate> ::= <Op Equate> '!=' <Op Compare> • <Op Compare> ::= <Op Compare> • '<' <Op Shift> <Op Compare> ::= <Op Compare> • '>' <Op Shift> <Op Compare> ::= <Op Compare> • '<=' <Op Shift> <Op Compare> ::= <Op Compare> • '>=' <Op Shift> '<' Shift 92 '<=' Shift 107 '>' Shift 109 '>=' Shift 111 '!=' Reduce 129 '&' Reduce 129 '&&' Reduce 129 '&=' Reduce 129 ')' Reduce 129 '*=' Reduce 129 ',' Reduce 129 '/' Reduce 129 ':' Reduce 129 ';' Reduce 129 '?' Reduce 129 ']' Reduce 129 '^' Reduce 129 '^=' Reduce 129 ' ' Reduce 129 ' ' Reduce 129 ' =' Reduce 129 '+=' Reduce 129 '<=' Reduce 129 '=' Reduce 129 '-' Reduce 129 '>=' Reduce 129 </pre>

GTGTEQ reduce 130 state 269 op_equate : op_equate EQEQ op_compare . (129) op_compare : op_compare . LT op_shift (132) op_compare : op_compare . GT op_shift (133) op_compare : op_compare . LTEQ op_shift (134) op_compare : op_compare . GTEQ op_shift (135) LT shift 204 LTEQ shift 205 GT shift 206 GTEQ shift 207 EXCLAMEQ reduce 129 AMP reduce 129 AMPAMP reduce 129 AMPEQ reduce 129 RPARAN reduce 129 TIMESEQ reduce 129 COMMA reduce 129 DIVEQ reduce 129 COLON reduce 129 SEMI reduce 129 QUESTION reduce 129 RBRACKET reduce 129 CARET reduce 129 CARETEQ reduce 129 PIPE reduce 129 PIPEPIPE reduce 129 PIPEEQ reduce 129 PLUSEQ reduce 129 LTLTEQ reduce 129 EQ reduce 129 MINUSEQ reduce 129 EQEQ reduce 129 GTGTEQ reduce 129	'>=' Reduce 129 State 118 <Op Equate> ::= <Op Equate> '==' <Op Compare> • <Op Compare> ::= <Op Compare> • '<' <Op Shift> <Op Compare> ::= <Op Compare> • '>' <Op Shift> <Op Compare> ::= <Op Compare> • '<=' <Op Shift> <Op Compare> ::= <Op Compare> • '>=' <Op Shift> '<' Shift 92 '<=' Shift 107 '>' Shift 109 '>=' Shift 111 '!=' Reduce 128 '&' Reduce 128 '&&' Reduce 128 '&=' Reduce 128 ')' Reduce 128 '*' Reduce 128 ',' Reduce 128 '/' Reduce 128 ':' Reduce 128 ';' Reduce 128 '?' Reduce 128 ']' Reduce 128 '^' Reduce 128 '^=' Reduce 128 ' ' Reduce 128 ' ' Reduce 128 ' =' Reduce 128 '+=' Reduce 128 '<<=' Reduce 128 '=' Reduce 128 '-=' Reduce 128 '==' Reduce 128 '>>=' Reduce 128
state 270 op_compare : op_compare LT op_shift . (132) op_shift : op_shift . LTLT op_add (137) op_shift : op_shift . GTGT op_add (138) LTLT shift 208 GTGT shift 209 EXCLAMEQ reduce 132 AMP reduce 132 AMPAMP reduce 132 AMPEQ reduce 132 RPARAN reduce 132 TIMESEQ reduce 132 COMMA reduce 132 DIVEQ reduce 132 COLON reduce 132 SEMI reduce 132 QUESTION reduce 132 RBRACKET reduce 132 CARET reduce 132 CARETEQ reduce 132 PIPE reduce 132 PIPEPIPE reduce 132	State 99 <Op Compare> ::= <Op Compare> '<' <Op Shift> • <Op Shift> ::= <Op Shift> • '<<' <Op Add> <Op Shift> ::= <Op Shift> • '>>' <Op Add> '<<' Shift 100 '>>' Shift 105 '!=' Reduce 131 '&' Reduce 131 '&&' Reduce 131 '&=' Reduce 131 ')' Reduce 131 '*' Reduce 131 ',' Reduce 131 '/' Reduce 131 ':' Reduce 131 ';' Reduce 131 '?' Reduce 131 ']' Reduce 131 '^' Reduce 131 '^=' Reduce 131 ' ' Reduce 131 ' ' Reduce 131

<p>QUESTION reduce 133 RBRACKET reduce 133 CARET reduce 133 CARETEQ reduce 133 PIPE reduce 133 PIPEPIPE reduce 133 PIPEEQ reduce 133 PLUSEQ reduce 133 LT reduce 133 LTLTEQ reduce 133 LTEQ reduce 133 EQ reduce 133 MINUSEQ reduce 133 EQEQ reduce 133 GT reduce 133 GTEQ reduce 133 GTGTEQ reduce 133</p>	<p>'?' Reduce 132 ']' Reduce 132 '^' Reduce 132 '^=' Reduce 132 ' ' Reduce 132 ' ' Reduce 132 ' =' Reduce 132 '+=' Reduce 132 '<' Reduce 132 '<=' Reduce 132 '<=' Reduce 132 '=' Reduce 132 '-' Reduce 132 '==' Reduce 132 '>' Reduce 132 '>=' Reduce 132 '>=' Reduce 132</p>
<p>state 273 op_compare : op_compare GTEQ op_shift . (135) op_shift : op_shift . LTLT op_add (137) op_shift : op_shift . GTGT op_add (138)</p> <p>LTLT shift 208 GTGT shift 209 EXCLAMEQ reduce 135 AMP reduce 135 AMPAMP reduce 135 AMPEQ reduce 135 RPARAN reduce 135 TIMESEQ reduce 135 COMMA reduce 135 DIVEQ reduce 135 COLON reduce 135 SEMI reduce 135 QUESTION reduce 135 RBRACKET reduce 135 CARET reduce 135 CARETEQ reduce 135 PIPE reduce 135 PIPEPIPE reduce 135 PIPEEQ reduce 135 PLUSEQ reduce 135 LT reduce 135 LTLTEQ reduce 135 LTEQ reduce 135 EQ reduce 135 MINUSEQ reduce 135 EQEQ reduce 135 GT reduce 135 GTEQ reduce 135 GTGTEQ reduce 135</p>	<p>State 112 <Op Compare> ::= <Op Compare> '>=' <Op Shift> • <Op Shift> ::= <Op Shift> • '<<' <Op Add> <Op Shift> ::= <Op Shift> • '>>' <Op Add></p> <p>'<<' Shift 100 '>>' Shift 105 '!=' Reduce 134 '&' Reduce 134 '&&' Reduce 134 '&=' Reduce 134 ')' Reduce 134 '*=' Reduce 134 ',' Reduce 134 '/=' Reduce 134 ':' Reduce 134 ';' Reduce 134 '?' Reduce 134 ']' Reduce 134 '^' Reduce 134 '^=' Reduce 134 ' ' Reduce 134 ' ' Reduce 134 ' =' Reduce 134 '+=' Reduce 134 '<' Reduce 134 '<=' Reduce 134 '<=' Reduce 134 '=' Reduce 134 '-' Reduce 134 '==' Reduce 134 '>' Reduce 134 '>=' Reduce 134 '>=' Reduce 134</p>
<p>state 274 op_shift : op_shift LTLT op_add . (137) op_add : op_add . PLUS op_mult (140) op_add : op_add . MINUS op_mult (141)</p> <p>MINUS shift 210 PLUS shift 211 EXCLAMEQ reduce 137 AMP reduce 137 AMPAMP reduce 137 AMPEQ reduce 137 RPARAN reduce 137</p>	<p>State 101 <Op Shift> ::= <Op Shift> '<<' <Op Add> • <Op Add> ::= <Op Add> • '+' <Op Mult> <Op Add> ::= <Op Add> • '-' <Op Mult></p> <p>'-' Shift 78 '+' Shift 102 '!=' Reduce 136 '&' Reduce 136 '&&' Reduce 136 '&=' Reduce 136 ')' Reduce 136</p>

TIMESEQ reduce 137 COMMA reduce 137 DIVEQ reduce 137 COLON reduce 137 SEMI reduce 137 QUESTION reduce 137 RBRACKET reduce 137 CARET reduce 137 CARETEQ reduce 137 PIPE reduce 137 PIPEPIPE reduce 137 PIPEEQ reduce 137 PLUSEQ reduce 137 LT reduce 137 LTLT reduce 137 LTLTEQ reduce 137 LTEQ reduce 137 EQ reduce 137 MINUSEQ reduce 137 EQEQ reduce 137 GT reduce 137 GTEQ reduce 137 GTGT reduce 137 GTGTEQ reduce 137	'*=' Reduce 136 ',' Reduce 136 '/' Reduce 136 ':' Reduce 136 ';' Reduce 136 '? ' Reduce 136 ']' Reduce 136 '^' Reduce 136 '^=' Reduce 136 ' ' Reduce 136 ' ' Reduce 136 ' =' Reduce 136 '+' Reduce 136 '<' Reduce 136 '<<' Reduce 136 '<=' Reduce 136 '<=' Reduce 136 '=' Reduce 136 '==' Reduce 136 '>' Reduce 136 '>=' Reduce 136 '>>' Reduce 136 '>>=' Reduce 136
state 275 op_shift : op_shift GTGT op_add . (138) op_add : op_add . PLUS op_mult (140) op_add : op_add . MINUS op_mult (141) MINUS shift 210 PLUS shift 211 EXCLAMEQ reduce 138 AMP reduce 138 AMPAMP reduce 138 AMPEQ reduce 138 RPARAN reduce 138 TIMESEQ reduce 138 COMMA reduce 138 DIVEQ reduce 138 COLON reduce 138 SEMI reduce 138 QUESTION reduce 138 RBRACKET reduce 138 CARET reduce 138 CARETEQ reduce 138 PIPE reduce 138 PIPEPIPE reduce 138 PIPEEQ reduce 138 PLUSEQ reduce 138 LT reduce 138 LTLT reduce 138 LTLTEQ reduce 138 LTEQ reduce 138 EQ reduce 138 MINUSEQ reduce 138 EQEQ reduce 138 GT reduce 138 GTEQ reduce 138 GTGT reduce 138 GTGTEQ reduce 138	State 106 <Op Shift> ::= <Op Shift> '>>' <Op Add> • <Op Add> ::= <Op Add> • '+' <Op Mult> <Op Add> ::= <Op Add> • '-' <Op Mult> '-' Shift 78 '+' Shift 102 '!=' Reduce 137 '&' Reduce 137 '&&' Reduce 137 '&=' Reduce 137 ')' Reduce 137 '*=' Reduce 137 ',' Reduce 137 '/' Reduce 137 ':' Reduce 137 ';' Reduce 137 '? ' Reduce 137 ']' Reduce 137 '^' Reduce 137 '^=' Reduce 137 ' ' Reduce 137 ' ' Reduce 137 ' =' Reduce 137 '+' Reduce 137 '<' Reduce 137 '<<' Reduce 137 '<=' Reduce 137 '<=' Reduce 137 '=' Reduce 137 '==' Reduce 137 '=' Reduce 137 '>' Reduce 137 '>=' Reduce 137 '>>' Reduce 137 '>>=' Reduce 137
state 276 op_add : op_add MINUS op_mult . (141) op_mult : op_mult . TIMES op_unary (143)	State 79 <Op Add> ::= <Op Add> '-' <Op Mult> • <Op Mult> ::= <Op Mult> • '*' <Op Unary>

<pre> op_mult : op_mult . DIV op_unary (144) op_mult : op_mult . PERCENT op_unary (145) PERCENT shift 212 TIMES shift 213 DIV shift 214 MINUS reduce 141 EXCLAMEQ reduce 141 AMP reduce 141 AMPAMP reduce 141 AMPEQ reduce 141 RPARAN reduce 141 TIMESEQ reduce 141 COMMA reduce 141 DIVEQ reduce 141 COLON reduce 141 SEMI reduce 141 QUESTION reduce 141 RBRACKET reduce 141 CARET reduce 141 CARETEQ reduce 141 PIPE reduce 141 PIPEPIPE reduce 141 PIPEEQ reduce 141 PLUS reduce 141 PLUSEQ reduce 141 LT reduce 141 LTLT reduce 141 LTLTEQ reduce 141 LTEQ reduce 141 EQ reduce 141 MINUSEQ reduce 141 EQEQ reduce 141 GT reduce 141 GTEQ reduce 141 GTGT reduce 141 GTGTEQ reduce 141 </pre>	<pre> <Op Mult> ::= <Op Mult> • '/' <Op Unary> <Op Mult> ::= <Op Mult> • '%' <Op Unary> '%' Shift 80 '*' Shift 94 '/' Shift 97 '-' Reduce 140 '!=' Reduce 140 '&' Reduce 140 '&&' Reduce 140 '&=' Reduce 140 ')' Reduce 140 '*=' Reduce 140 ',' Reduce 140 '/' Reduce 140 ':' Reduce 140 ';' Reduce 140 '?' Reduce 140 ']' Reduce 140 '^' Reduce 140 '^=' Reduce 140 ' ' Reduce 140 ' ' Reduce 140 ' =' Reduce 140 '+' Reduce 140 '+=' Reduce 140 '<' Reduce 140 '<<' Reduce 140 '<=' Reduce 140 '<=' Reduce 140 '=' Reduce 140 '-' Reduce 140 '==' Reduce 140 '>' Reduce 140 '>=' Reduce 140 '>>' Reduce 140 '>>=' Reduce 140 </pre>
<pre> state 277 op_add : op_add PLUS op_mult . (140) op_mult : op_mult . TIMES op_unary (143) op_mult : op_mult . DIV op_unary (144) op_mult : op_mult . PERCENT op_unary (145) PERCENT shift 212 TIMES shift 213 DIV shift 214 MINUS reduce 140 EXCLAMEQ reduce 140 AMP reduce 140 AMPAMP reduce 140 AMPEQ reduce 140 RPARAN reduce 140 TIMESEQ reduce 140 COMMA reduce 140 DIVEQ reduce 140 COLON reduce 140 SEMI reduce 140 QUESTION reduce 140 RBRACKET reduce 140 CARET reduce 140 </pre>	<pre> State 103 <Op Add> ::= <Op Add> '+' <Op Mult> • <Op Mult> ::= <Op Mult> • '*' <Op Unary> <Op Mult> ::= <Op Mult> • '/' <Op Unary> <Op Mult> ::= <Op Mult> • '%' <Op Unary> '%' Shift 80 '*' Shift 94 '/' Shift 97 '-' Reduce 139 '!=' Reduce 139 '&' Reduce 139 '&&' Reduce 139 '&=' Reduce 139 ')' Reduce 139 '*=' Reduce 139 ',' Reduce 139 '/' Reduce 139 ':' Reduce 139 ';' Reduce 139 '?' Reduce 139 ']' Reduce 139 </pre>

<p>CARETEQ reduce 140 PIPE reduce 140 PIPEPIPE reduce 140 PIPEEQ reduce 140 PLUS reduce 140 PLUSEQ reduce 140 LT reduce 140 LTLT reduce 140 LTLTEQ reduce 140 LTEQ reduce 140 EQ reduce 140 MINUSEQ reduce 140 EQEQ reduce 140 GT reduce 140 GTEQ reduce 140 GTGT reduce 140 GTGTEQ reduce 140</p>	<p>'^' Reduce 139 '^=' Reduce 139 ' ' Reduce 139 ' ' Reduce 139 ' =' Reduce 139 '+' Reduce 139 '+=' Reduce 139 '<' Reduce 139 '<<' Reduce 139 '<=' Reduce 139 '<=' Reduce 139 '=' Reduce 139 '-' Reduce 139 '==' Reduce 139 '>' Reduce 139 '>=' Reduce 139 '>>' Reduce 139 '>>=' Reduce 139</p>
<p>state 278 op_mult : op_mult PERCENT op_unary . (145) . . reduce 145</p>	<p>State 163 <Op Mult> ::= <Op Mult> '%' <Op Unary> . '- ' Reduce 144 '!=' Reduce 144 '%' Reduce 144 '&' Reduce 144 '&&' Reduce 144 '&=' Reduce 144 ')' Reduce 144 '*' Reduce 144 '*=' Reduce 144 ',' Reduce 144 '/' Reduce 144 '/' Reduce 144 ':' Reduce 144 ';' Reduce 144 '?' Reduce 144 ']' Reduce 144 '^' Reduce 144 '^=' Reduce 144 ' ' Reduce 144 ' ' Reduce 144 ' =' Reduce 144 '+' Reduce 144 '+=' Reduce 144 '<' Reduce 144 '<<' Reduce 144 '<=' Reduce 144 '<=' Reduce 144 '=' Reduce 144 '-' Reduce 144 '==' Reduce 144 '>' Reduce 144 '>=' Reduce 144 '>>' Reduce 144 '>>=' Reduce 144</p>
<p>state 279 op_mult : op_mult TIMES op_unary . (143) . . reduce 143</p>	<p>State 95 <Op Mult> ::= <Op Mult> '*' <Op Unary> . '- ' Reduce 142 '!=' Reduce 142 '%' Reduce 142 '&' Reduce 142</p>

	'&&' Reduce 142 '&=' Reduce 142 ')' Reduce 142 '*' Reduce 142 '*=' Reduce 142 ',' Reduce 142 '/' Reduce 142 '/' Reduce 142 ':' Reduce 142 ';' Reduce 142 '?' Reduce 142 ']' Reduce 142 '^' Reduce 142 '^=' Reduce 142 ' ' Reduce 142 ' ' Reduce 142 ' =' Reduce 142 '+' Reduce 142 '+=' Reduce 142 '<' Reduce 142 '<<' Reduce 142 '<=' Reduce 142 '<=' Reduce 142 '=' Reduce 142 '-' Reduce 142 '==' Reduce 142 '>' Reduce 142 '>=' Reduce 142 '>>' Reduce 142 '>>=' Reduce 142
state 280 op_mult : op_mult DIV op_unary . (144) . reduce 144	State 98 <Op Mult> ::= <Op Mult> '/' <Op Unary> . '-' Reduce 143 '!=' Reduce 143 '%' Reduce 143 '&' Reduce 143 '&&' Reduce 143 '&=' Reduce 143 ')' Reduce 143 '*' Reduce 143 '*=' Reduce 143 ',' Reduce 143 '/' Reduce 143 '/' Reduce 143 ':' Reduce 143 ';' Reduce 143 '?' Reduce 143 ']' Reduce 143 '^' Reduce 143 '^=' Reduce 143 ' ' Reduce 143 ' ' Reduce 143 ' =' Reduce 143 '+' Reduce 143 '+=' Reduce 143 '<' Reduce 143 '<<' Reduce 143 '<=' Reduce 143 '<=' Reduce 143 '=' Reduce 143 '-' Reduce 143 '==' Reduce 143

	'^' Reduce 159 '^=' Reduce 159 ' ' Reduce 159 ' ' Reduce 159 ' =' Reduce 159 '+' Reduce 159 '++' Reduce 159 '+=' Reduce 159 '<' Reduce 159 '<<' Reduce 159 '<=' Reduce 159 '<=' Reduce 159 '=' Reduce 159 '-' Reduce 159 '==' Reduce 159 '>' Reduce 159 '>-' Reduce 159 '>=' Reduce 159 '>>' Reduce 159 '>=' Reduce 159
state 283 expr : expr . COMMA op_assign (104) op_pointer : op_pointer LBRACKET expr . RBRACKET (162) COMMA shift 194 RBRACKET shift 298 . error	State 158 <Op Pointer> ::= <Op Pointer> '[' <Expr> • ']' <Expr> ::= <Expr> • ',' <Op Assign> ',' Shift 76 ']' Shift 159
state 284 op_pointer : op_pointer MINUSGT value . (161) . reduce 161	State 162 <Op Pointer> ::= <Op Pointer> '-> <Value> • '-' Reduce 160 '--' Reduce 160 '!=' Reduce 160 '%' Reduce 160 '&' Reduce 160 '&&' Reduce 160 '&=' Reduce 160 ')' Reduce 160 '*' Reduce 160 '*=' Reduce 160 ',' Reduce 160 '.' Reduce 160 '/' Reduce 160 '/' Reduce 160 ':' Reduce 160 ';' Reduce 160 '?' Reduce 160 '[' Reduce 160 ']' Reduce 160 '^' Reduce 160 '^=' Reduce 160 ' ' Reduce 160 ' ' Reduce 160 ' =' Reduce 160 '+' Reduce 160 '++' Reduce 160 '+=' Reduce 160 '<' Reduce 160 '<<' Reduce 160 '<=' Reduce 160 '<=' Reduce 160 '=' Reduce 160

	'-' Reduce 160 '==' Reduce 160 '>' Reduce 160 '-'>' Reduce 160 '>=' Reduce 160 '>>' Reduce 160 '>=' Reduce 160
state 285 normal_stm : DO stm WHILE . LPARAN expr RPARAN (87) LPARAN shift 299 . error	State 310 <Normal Stm> ::= do <Stm> while • '(' <Expr> ')' '(' Shift 311
state 286 arg : expr . (96) expr : expr . COMMA op_assign (104) COMMA shift 194 RPARAN reduce 96 SEMI reduce 96	State 306 <Arg> ::= <Expr> • <Expr> ::= <Expr> • ',' <Op Assign> ',' Shift 76 ')' Reduce 95 ';' Reduce 95
state 287 stm : FOR LPARAN arg . SEMI arg SEMI arg RPARAN stm (81) SEMI shift 300 . error	State 236 <Stm> ::= for '(' <Arg> • ';' <Arg> ';' <Arg> ')' <Stm> ';' Shift 237
state 288 normal_stm : GOTO ID SEMI . (91) . reduce 91	State 244 <Normal Stm> ::= goto Id ';' • '-' Reduce 90 '--' Reduce 90 '!' Reduce 90 '&' Reduce 90 '(' Reduce 90 '*' Reduce 90 ';' Reduce 90 '{' Reduce 90 '}' Reduce 90 '~' Reduce 90 '++' Reduce 90 auto Reduce 90 break Reduce 90 case Reduce 90 char Reduce 90 CharLiteral Reduce 90 const Reduce 90 continue Reduce 90 DecLiteral Reduce 90 default Reduce 90 do Reduce 90 double Reduce 90 else Reduce 90 enum Reduce 90 extern Reduce 90 float Reduce 90 FloatLiteral Reduce 90 for Reduce 90 goto Reduce 90 HexLiteral Reduce 90 Id Reduce 90 if Reduce 90 int Reduce 90 long Reduce 90

	OctLiteral Reduce 90 register Reduce 90 return Reduce 90 short Reduce 90 signed Reduce 90 sizeof Reduce 90 static Reduce 90 StringLiteral Reduce 90 struct Reduce 90 switch Reduce 90 union Reduce 90 unsigned Reduce 90 void Reduce 90 volatile Reduce 90 while Reduce 90
state 289 stm : IF LPARAN expr . RPARAN stm (78) stm : IF LPARAN expr . RPARAN then_stm ELSE stm (79) expr : expr . COMMA op_assign (104) RPARAN shift 301 COMMA shift 194 . error	State 249 <Stm> ::= if '(' <Expr> • ')' <Stm> <Stm> ::= if '(' <Expr> • ')' <Then Stm> else <Stm> <Expr> ::= <Expr> • ',' <Op Assign> ')' Shift 250 ',' Shift 76
state 290 normal_stm : RETURN expr SEMI . (94) . reduce 94	State 265 <Normal Stm> ::= return <Expr> ';' • '-' Reduce 93 '--' Reduce 93 '!' Reduce 93 '&' Reduce 93 '(' Reduce 93 '*' Reduce 93 ';' Reduce 93 '{' Reduce 93 '}' Reduce 93 '~' Reduce 93 '++' Reduce 93 auto Reduce 93 break Reduce 93 case Reduce 93 char Reduce 93 CharLiteral Reduce 93 const Reduce 93 continue Reduce 93 DecLiteral Reduce 93 default Reduce 93 do Reduce 93 double Reduce 93 else Reduce 93 enum Reduce 93 extern Reduce 93 float Reduce 93 FloatLiteral Reduce 93 for Reduce 93 goto Reduce 93 HexLiteral Reduce 93 Id Reduce 93 if Reduce 93 int Reduce 93 long Reduce 93 OctLiteral Reduce 93 register Reduce 93 return Reduce 93

	short Reduce 93 signed Reduce 93 sizeof Reduce 93 static Reduce 93 StringLiteral Reduce 93 struct Reduce 93 switch Reduce 93 union Reduce 93 unsigned Reduce 93 void Reduce 93 volatile Reduce 93 while Reduce 93
state 291 normal_stm : SWITCH LPARAN expr . RPARAN LBRACE case_stms RBRACE (88) expr : expr . COMMA op_assign (104) RPARAN shift 302 COMMA shift 194 . error	State 268 <Normal Stm> ::= switch '(' <Expr> • ')' '{' <Case Stms> '}' <Expr> ::= <Expr> • ',' <Op Assign> ')' Shift 269 ',' Shift 76
state 292 stm : WHILE LPARAN expr . RPARAN stm (80) expr : expr . COMMA op_assign (104) RPARAN shift 303 COMMA shift 194 . error	State 276 <Stm> ::= while '(' <Expr> • ')' <Stm> <Expr> ::= <Expr> • ',' <Op Assign> ')' Shift 277 ',' Shift 76
state 293 op_unary : LPARAN type RPARAN op_unary . (156) . reduce 156	State 170 <Op Unary> ::= '(' <Type> ')' <Op Unary> • '-' Reduce 155 '!=' Reduce 155 '%' Reduce 155 '&' Reduce 155 '&&' Reduce 155 '&=' Reduce 155 ')' Reduce 155 '*' Reduce 155 '*=' Reduce 155 ',' Reduce 155 '/' Reduce 155 '/=' Reduce 155 ':' Reduce 155 ';' Reduce 155 '?' Reduce 155 ']' Reduce 155 '^' Reduce 155 '^=' Reduce 155 ' ' Reduce 155 ' ' Reduce 155 ' =' Reduce 155 '+' Reduce 155 '+=' Reduce 155 '<' Reduce 155 '<<' Reduce 155 '<=' Reduce 155 '<=' Reduce 155 '=' Reduce 155 '==> Reduce 155 '==> Reduce 155

	'>' Reduce 155 '>=' Reduce 155 '>>' Reduce 155 '>>=' Reduce 155
state 294 value : ID LPARAN expr RPARAN . (170) . reduce 170	State 75 <Value> ::= Id '(' <Expr> ')' • '-' Reduce 169 '--' Reduce 169 '!=' Reduce 169 '%' Reduce 169 '&' Reduce 169 '&&' Reduce 169 '&=' Reduce 169 ')' Reduce 169 '*' Reduce 169 '*=' Reduce 169 ',' Reduce 169 '.' Reduce 169 '/' Reduce 169 '/=' Reduce 169 ':' Reduce 169 ';' Reduce 169 '?' Reduce 169 '[' Reduce 169 ']' Reduce 169 '^' Reduce 169 '^=' Reduce 169 ' ' Reduce 169 ' ' Reduce 169 ' =' Reduce 169 '+' Reduce 169 '++' Reduce 169 '+=' Reduce 169 '<' Reduce 169 '<<' Reduce 169 '<=' Reduce 169 '<=' Reduce 169 '=' Reduce 169 '-=' Reduce 169 '==' Reduce 169 '>' Reduce 169 '->' Reduce 169 '>=' Reduce 169 '>>' Reduce 169 '>>=' Reduce 169
state 295 op_unary : SIZEOF LPARAN ID pointers . RPARAN (158) RPARAN shift 304 . error	State 58 <Op Unary> ::= sizeof '(' Id <Pointers> • ')' • ')' Shift 59
state 296 op_unary : SIZEOF LPARAN type RPARAN . (157) . reduce 157	State 72 <Op Unary> ::= sizeof '(' <Type> ')' • '-' Reduce 156 '!=' Reduce 156 '%' Reduce 156 '&' Reduce 156 '&&' Reduce 156 '&=' Reduce 156 ')' Reduce 156 '*' Reduce 156

	'*' Reduce 156 ',' Reduce 156 '/' Reduce 156 '/=' Reduce 156 ':' Reduce 156 ';' Reduce 156 '?' Reduce 156 ']' Reduce 156 '^' Reduce 156 '^=' Reduce 156 ' ' Reduce 156 ' ' Reduce 156 ' =' Reduce 156 '+' Reduce 156 '+=' Reduce 156 '<' Reduce 156 '<<' Reduce 156 '<=' Reduce 156 '<=' Reduce 156 '=' Reduce 156 '-=' Reduce 156 '==' Reduce 156 '>' Reduce 156 '>=' Reduce 156 '>>' Reduce 156 '>>=' Reduce 156
state 297 op_if : op_or QUESTION op_if COLON . op_if (117) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 305 value goto 121 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 134 <Op If> ::= <Op Or> '?' <Op If> ':' • <Op If> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Op Add> Goto 77 <Op And> Goto 87 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 135 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 298 op_pointer : op_pointer LBRACKET expr	State 159 <Op Pointer> ::= <Op Pointer> '['

<p>RBRACKET . (162)</p> <p>. reduce 162</p>	<p><Expr> ']' •</p> <p>'-' Reduce 161 '--' Reduce 161 '!=' Reduce 161 '%' Reduce 161 '&' Reduce 161 '&&' Reduce 161 '&=' Reduce 161 ')' Reduce 161 '*' Reduce 161 '*=' Reduce 161 ',' Reduce 161 '.' Reduce 161 '/' Reduce 161 '/' Reduce 161 ':' Reduce 161 ';' Reduce 161 '?' Reduce 161 '[' Reduce 161 ']' Reduce 161 '^' Reduce 161 '^=' Reduce 161 ' ' Reduce 161 ' ' Reduce 161 ' =' Reduce 161 '+' Reduce 161 '++' Reduce 161 '+=' Reduce 161 '<' Reduce 161 '<<' Reduce 161 '<=' Reduce 161 '<=' Reduce 161 '=' Reduce 161 '==' Reduce 161 '==' Reduce 161 '>' Reduce 161 '>' Reduce 161 '>=' Reduce 161 '>=' Reduce 161</p>
<p>state 299</p> <p>normal_stm : DO stm WHILE LPARAN . expr RPARAN (87)</p> <p>MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error</p> <p>op_if goto 119</p>	<p>State 311</p> <p><Normal Stm> ::= do <Stm> while '(' • <Expr> ')'</p> <p>'-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73</p> <p><Expr> Goto 312 <Op Add> Goto 77</p>

<pre> expr goto 306 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 300 stm : FOR LPARAN arg SEMI . arg SEMI arg RPARAN stm (81) arg : . (97) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 SEMI reduce 97 op_if goto 119 expr goto 286 arg goto 307 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 237 <Stm> ::= for '(' <Arg> ';' • <Arg> ';' <Arg> ')' <Stm> <Arg> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 ';' Reduce 96 <Arg> Goto 238 <Expr> Goto 306 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 301 stm : IF LPARAN expr RPARAN . stm (78) stm : IF LPARAN expr RPARAN . then_stm ELSE stm (79) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 </pre>	<pre> State 250 <Stm> ::= if '(' <Expr> ')' • <Stm> <Stm> ::= if '(' <Expr> ')' • <Then Stm> else <Stm> <Sign> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 </pre>

AMP shift 105	'&' Shift 42
LPARAN shift 106	'(' Shift 43
TIMES shift 107	'*' Shift 44
SEMI shift 146	';' Shift 228
LBRACE shift 92	'{' Shift 227
TILDE shift 109	'~' Shift 45
PLUSPLUS shift 110	'++' Shift 46
AUTO shift 1	auto Shift 1
BREAK shift 147	break Shift 229
CHARLITERAL shift 111	CharLiteral Shift 47
CONST shift 2	const Shift 2
CONTINUE shift 148	continue Shift 231
DECLITERAL shift 112	DecLiteral Shift 48
DO shift 149	do Shift 233
ENUM shift 31	enum Shift 24
EXTERN shift 4	extern Shift 17
FLOATLITERAL shift 113	FloatLiteral Shift 49
FOR shift 308	for Shift 251
GOTO shift 151	goto Shift 242
HEXLITERAL shift 114	HexLiteral Shift 50
ID shift 152	Id Shift 245
IF shift 309	if Shift 259
OCTLITERAL shift 116	OctLiteral Shift 54
REGISTER shift 6	register Shift 19
RETURN shift 154	return Shift 263
SIGNED shift 7	signed Shift 20
SIZEOF shift 117	sizeof Shift 55
STATIC shift 8	static Shift 21
STRINGLITERAL shift 118	StringLiteral Shift 73
STRUCT shift 32	struct Shift 26
SWITCH shift 155	switch Shift 266
UNION shift 33	union Shift 28
UNSIGNED shift 12	unsigned Shift 30
VOLATILE shift 13	volatile Shift 31
WHILE shift 310	while Shift 293
CHAR reduce 64	char Reduce 63
DOUBLE reduce 64	double Reduce 63
FLOAT reduce 64	float Reduce 63
INT reduce 64	int Reduce 63
LONG reduce 64	long Reduce 63
SHORT reduce 64	short Reduce 63
VOID reduce 64	void Reduce 63
var_decl goto 157	<Base> Goto 32
block goto 158	<Block> Goto 278
type goto 57	<Expr> Goto 279
mod goto 25	<Mod> Goto 36
op_if goto 119	<Normal Stm> Goto 297
expr goto 159	<Op Add> Goto 77
base goto 26	<Op And> Goto 87
sign goto 27	<Op Assign> Goto 126
stm goto 311	<Op BinAND> Goto 89
then_stm goto 312	<Op BinOR> Goto 127
normal_stm goto 313	<Op BinXOR> Goto 125
value goto 121	<Op Compare> Goto 91
op_assign goto 122	<Op Equate> Goto 124
op_or goto 123	<Op If> Goto 128
op_and goto 124	<Op Mult> Goto 93
op_binor goto 125	<Op Or> Goto 131
op_binxor goto 126	<Op Pointer> Goto 81
op_binand goto 127	<Op Shift> Goto 116
op_equate goto 128	<Op Unary> Goto 104
op_compare goto 129	<Sign> Goto 60
op_shift goto 130	<Stm> Goto 299

op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	<Then Stm> Goto 307 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283
state 302 normal_stm : SWITCH LPARAN expr RPARAN . LBRACE case_stms RBRACE (88) LBRACE shift 314 . error	State 269 <Normal Stm> ::= switch '(' <Expr> ')' • '{' <Case Stms> '}' '{' Shift 270
state 303 stm : WHILE LPARAN expr RPARAN . stm (80) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 150 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 153 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 156 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158	State 277 <Stm> ::= while '(' <Expr> ')' • <Stm> <Sign> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 Decliteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 234 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 247 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 274 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279

<pre> type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 315 normal_stm goto 161 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Mod> Goto 36 <Normal Stm> Goto 281 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 282 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283 </pre>
<pre> state 304 op_unary : SIZEOF LPARAN ID pointers RPARAN . (158) . reduce 158 </pre>	<pre> State 59 <Op Unary> ::= sizeof '(' Id <Pointers> ')' • '-' Reduce 157 '!' Reduce 157 '%' Reduce 157 '&' Reduce 157 '&&' Reduce 157 '&=' Reduce 157 ')' Reduce 157 '*' Reduce 157 '*=' Reduce 157 ',' Reduce 157 '/' Reduce 157 '/=' Reduce 157 ':' Reduce 157 ';' Reduce 157 '?' Reduce 157 ']' Reduce 157 '^' Reduce 157 '^=' Reduce 157 ' ' Reduce 157 ' ' Reduce 157 ' =' Reduce 157 '+' Reduce 157 '+=' Reduce 157 '<' Reduce 157 '<<' Reduce 157 '<=' Reduce 157 '=' Reduce 157 '-=' Reduce 157 '==' Reduce 157 '>' Reduce 157 '>=' Reduce 157 '>>' Reduce 157 '>>=' Reduce 157 </pre>
<pre> state 305 op_if : op_or QUESTION op_if COLON op_if . (117) </pre>	<pre> State 135 <Op If> ::= <Op Or> '?' <Op If> ':' <Op If> • </pre>

. reduce 117	'&=' Reduce 116 ')' Reduce 116 '*=' Reduce 116 ',' Reduce 116 '/=' Reduce 116 ':' Reduce 116 ';' Reduce 116 ']' Reduce 116 '^=' Reduce 116 ' =' Reduce 116 '+=' Reduce 116 '<=<' Reduce 116 '=' Reduce 116 '-=' Reduce 116 '>=' Reduce 116
state 306 normal_stm : DO stm WHILE LPARAN expr . RPARAN (87) expr : expr . COMMA op_assign (104) RPARAN shift 316 COMMA shift 194 . error	State 312 <Normal Stm> ::= do <Stm> while '(' <Expr> • ')' <Expr> ::= <Expr> • ',' <Op Assign> ')' Shift 313 ',' Shift 76
state 307 stm : FOR LPARAN arg SEMI arg . SEMI arg RPARAN stm (81) SEMI shift 317 . error	State 238 <Stm> ::= for '(' <Arg> ';' <Arg> • ';' <Arg> ')' <Stm> ';' Shift 239
state 308 stm : FOR . LPARAN arg SEMI arg SEMI arg RPARAN stm (81) then_stm : FOR . LPARAN arg SEMI arg SEMI arg RPARAN then_stm (85) LPARAN shift 318 . error	State 251 <Stm> ::= for • '(' <Arg> ';' <Arg> ';' <Arg> ')' <Stm> <Then Stm> ::= for • '(' <Arg> ';' <Arg> ';' <Arg> ')' <Then Stm> '(' Shift 252
state 309 stm : IF . LPARAN expr RPARAN stm (78) stm : IF . LPARAN expr RPARAN then_stm ELSE stm (79) then_stm : IF . LPARAN expr RPARAN then_stm ELSE then_stm (83) LPARAN shift 319 . error	State 259 <Stm> ::= if • '(' <Expr> ')' <Stm> <Stm> ::= if • '(' <Expr> ')' <Then Stm> else <Stm> <Then Stm> ::= if • '(' <Expr> ')' <Then Stm> else <Then Stm> '(' Shift 260
state 310 stm : WHILE . LPARAN expr RPARAN stm (80) then_stm : WHILE . LPARAN expr RPARAN then_stm (84) LPARAN shift 320 . error	State 293 <Stm> ::= while • '(' <Expr> ')' <Stm> <Then Stm> ::= while • '(' <Expr> ')' <Then Stm> '(' Shift 294
state 311 stm : IF LPARAN expr RPARAN stm . (78) . reduce 78	State 299 <Stm> ::= if '(' <Expr> ')' <Stm> • '-' Reduce 77 '--' Reduce 77 '!' Reduce 77 '&' Reduce 77 '(' Reduce 77

	'*' Reduce 77 ';' Reduce 77 '{' Reduce 77 '}' Reduce 77 '~' Reduce 77 '++' Reduce 77 auto Reduce 77 break Reduce 77 case Reduce 77 char Reduce 77 CharLiteral Reduce 77 const Reduce 77 continue Reduce 77 DecLiteral Reduce 77 default Reduce 77 do Reduce 77 double Reduce 77 enum Reduce 77 extern Reduce 77 float Reduce 77 FloatLiteral Reduce 77 for Reduce 77 goto Reduce 77 HexLiteral Reduce 77 Id Reduce 77 if Reduce 77 int Reduce 77 long Reduce 77 OctLiteral Reduce 77 register Reduce 77 return Reduce 77 short Reduce 77 signed Reduce 77 sizeof Reduce 77 static Reduce 77 StringLiteral Reduce 77 struct Reduce 77 switch Reduce 77 union Reduce 77 unsigned Reduce 77 void Reduce 77 volatile Reduce 77 while Reduce 77
state 312 stm : IF LPARAN expr RPARAN then_stm . ELSE stm (79) ELSE shift 321 . error	State 307 <Stm> ::= if '(' <Expr> ')' <Then Stm> • else <Stm> else Shift 308
state 313 stm : normal_stm . (82) then_stm : normal_stm . (86) MINUS reduce 82 MINUSMINUS reduce 82 EXCLAM reduce 82 AMP reduce 82 LPARAN reduce 82 TIMES reduce 82 SEMI reduce 82 LBRACE reduce 82 RBRACE reduce 82 TILDE reduce 82 PLUSPLUS reduce 82	State 297 <Stm> ::= <Normal Stm> • <Then Stm> ::= <Normal Stm> • '-' Reduce 81 '--' Reduce 81 '!' Reduce 81 '&' Reduce 81 '(' Reduce 81 '*' Reduce 81 ';' Reduce 81 '{' Reduce 81 '}' Reduce 81 '~' Reduce 81 '++' Reduce 81

<pre> AUTO reduce 82 BREAK reduce 82 CASE reduce 82 CHAR reduce 82 CHARLITERAL reduce 82 CONST reduce 82 CONTINUE reduce 82 DECLITERAL reduce 82 DEFAULT reduce 82 DO reduce 82 DOUBLE reduce 82 ELSE reduce 86 ENUM reduce 82 EXTERN reduce 82 FLOAT reduce 82 FLOATLITERAL reduce 82 FOR reduce 82 GOTO reduce 82 HEXLITERAL reduce 82 ID reduce 82 IF reduce 82 INT reduce 82 LONG reduce 82 OCTLITERAL reduce 82 REGISTER reduce 82 RETURN reduce 82 SHORT reduce 82 SIGNED reduce 82 SIZEOF reduce 82 STATIC reduce 82 STRINGLITERAL reduce 82 STRUCT reduce 82 SWITCH reduce 82 UNION reduce 82 UNSIGNED reduce 82 VOID reduce 82 VOLATILE reduce 82 WHILE reduce 82 </pre>	<pre> auto Reduce 81 break Reduce 81 case Reduce 81 char Reduce 81 CharLiteral Reduce 81 const Reduce 81 continue Reduce 81 DeclLiteral Reduce 81 default Reduce 81 do Reduce 81 double Reduce 81 enum Reduce 81 extern Reduce 81 float Reduce 81 FloatLiteral Reduce 81 for Reduce 81 goto Reduce 81 HexLiteral Reduce 81 Id Reduce 81 if Reduce 81 int Reduce 81 long Reduce 81 OctLiteral Reduce 81 register Reduce 81 return Reduce 81 short Reduce 81 signed Reduce 81 sizeof Reduce 81 static Reduce 81 StringLiteral Reduce 81 struct Reduce 81 switch Reduce 81 union Reduce 81 unsigned Reduce 81 void Reduce 81 volatile Reduce 81 while Reduce 81 else Reduce 85 </pre>
<pre> state 314 normal_stm : SWITCH LPARAN expr RPARAN LBRACE . case_stms RBRACE (88) case_stms : . (100) CASE shift 322 DEFAULT shift 323 RBRACE reduce 100 case_stms goto 324 </pre>	<pre> State 270 <Normal Stm> ::= switch '(' <Expr> ')' '{' • <Case Stms> '}' <Case Stms> ::= • case Shift 271 default Shift 287 '}' Reduce 99 <Case Stms> Goto 291 </pre>
<pre> state 315 stm : WHILE LPARAN expr RPARAN stm . (80) . reduce 80 </pre>	<pre> State 282 <Stm> ::= while '(' <Expr> ')' <Stm> • '-' Reduce 79 '--' Reduce 79 '!' Reduce 79 '&' Reduce 79 '(' Reduce 79 '*' Reduce 79 ';' Reduce 79 '{' Reduce 79 '}' Reduce 79 '~' Reduce 79 '++' Reduce 79 auto Reduce 79 </pre>

	break Reduce 79 case Reduce 79 char Reduce 79 CharLiteral Reduce 79 const Reduce 79 continue Reduce 79 DecLiteral Reduce 79 default Reduce 79 do Reduce 79 double Reduce 79 enum Reduce 79 extern Reduce 79 float Reduce 79 FloatLiteral Reduce 79 for Reduce 79 goto Reduce 79 HexLiteral Reduce 79 Id Reduce 79 if Reduce 79 int Reduce 79 long Reduce 79 OctLiteral Reduce 79 register Reduce 79 return Reduce 79 short Reduce 79 signed Reduce 79 sizeof Reduce 79 static Reduce 79 StringLiteral Reduce 79 struct Reduce 79 switch Reduce 79 union Reduce 79 unsigned Reduce 79 void Reduce 79 volatile Reduce 79 while Reduce 79
state 316 normal_stm : DO stm WHILE LPARAN expr RPARAN . (87) . reduce 87	State 313 <Normal Stm> ::= do <Stm> while '(' <Expr> ')' • '-' Reduce 86 '--' Reduce 86 '!' Reduce 86 '&' Reduce 86 '(' Reduce 86 '*' Reduce 86 ';' Reduce 86 '{' Reduce 86 '}' Reduce 86 '~' Reduce 86 '++' Reduce 86 auto Reduce 86 break Reduce 86 case Reduce 86 char Reduce 86 CharLiteral Reduce 86 const Reduce 86 continue Reduce 86 DecLiteral Reduce 86 default Reduce 86 do Reduce 86 double Reduce 86 else Reduce 86 enum Reduce 86

	extern Reduce 86 float Reduce 86 FloatLiteral Reduce 86 for Reduce 86 goto Reduce 86 HexLiteral Reduce 86 Id Reduce 86 if Reduce 86 int Reduce 86 long Reduce 86 OctLiteral Reduce 86 register Reduce 86 return Reduce 86 short Reduce 86 signed Reduce 86 sizeof Reduce 86 static Reduce 86 StringLiteral Reduce 86 struct Reduce 86 switch Reduce 86 union Reduce 86 unsigned Reduce 86 void Reduce 86 volatile Reduce 86 while Reduce 86
state 317 stm : FOR LPARAN arg SEMI arg SEMI . arg RPARAN stm (81) arg : . (97) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 RPARAN reduce 97 op_if goto 119 expr goto 286 arg goto 325 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133	State 239 <Stm> ::= for '(' <Arg> ';' <Arg> ';' • <Arg> ')' <Stm> <Arg> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 ')' Reduce 96 <Arg> Goto 240 <Expr> Goto 306 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104

op_pointer goto 134	<Value> Goto 96
state 318 stm : FOR LPARAN . arg SEMI arg SEMI arg RPARAN stm (81) then_stm : FOR LPARAN . arg SEMI arg SEMI arg RPARAN then_stm (85) arg : . (97) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 SEMI reduce 97 op_if goto 119 expr goto 286 arg goto 326 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 252 <Stm> ::= for '(' • <Arg> ';' <Arg> ';' <Arg> ')' <Stm> <Then Stm> ::= for '(' • <Arg> ';' <Arg> ';' <Arg> ')' <Arg> ';' <Arg> ')' <Then Stm> <Arg> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 ';' Reduce 96 <Arg> Goto 253 <Expr> Goto 306 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 319 stm : IF LPARAN . expr RPARAN stm (78) stm : IF LPARAN . expr RPARAN then_stm ELSE stm (79) then_stm : IF LPARAN . expr RPARAN then_stm ELSE then_stm (83) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115	State 260 <Stm> ::= if '(' • <Expr> ')' <Stm> <Stm> ::= if '(' • <Expr> ')' <Then Stm> else <Stm> <Then Stm> ::= if '(' • <Expr> ')' <Then Stm> else <Then Stm> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50

OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 expr goto 327 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Expr> Goto 261 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 320 stm : WHILE LPARAN . expr RPARAN stm (80) then_stm : WHILE LPARAN . expr RPARAN then_stm (84) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 . error op_if goto 119 expr goto 328 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 294 <Stm> ::= while '(' • <Expr> ')' <Stm> <Then Stm> ::= while '(' • <Expr> ')' <Then Stm> '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 <Expr> Goto 295 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 321 stm : IF LPARAN expr RPARAN then_stm ELSE . stm (79)	State 308 <Stm> ::= if '(' <Expr> ')' <Then Stm> else • <Stm>

sign : . (64)	<Sign> ::= •
MINUS shift 102	'-' Shift 39
MINUSMINUS shift 103	'--' Shift 40
EXCLAM shift 104	'!' Shift 41
AMP shift 105	'&' Shift 42
LPARAN shift 106	'(' Shift 43
TIMES shift 107	'*' Shift 44
SEMI shift 146	';' Shift 228
LBRACE shift 92	'{' Shift 227
TILDE shift 109	'~' Shift 45
PLUSPLUS shift 110	'++' Shift 46
AUTO shift 1	auto Shift 1
BREAK shift 147	break Shift 229
CHARLITERAL shift 111	CharLiteral Shift 47
CONST shift 2	const Shift 2
CONTINUE shift 148	continue Shift 231
DECLITERAL shift 112	DecLiteral Shift 48
DO shift 149	do Shift 233
ENUM shift 31	enum Shift 24
EXTERN shift 4	extern Shift 17
FLOATLITERAL shift 113	FloatLiteral Shift 49
FOR shift 150	for Shift 234
GOTO shift 151	goto Shift 242
HEXLITERAL shift 114	HexLiteral Shift 50
ID shift 152	Id Shift 245
IF shift 153	if Shift 247
OCTLITERAL shift 116	OctLiteral Shift 54
REGISTER shift 6	register Shift 19
RETURN shift 154	return Shift 263
SIGNED shift 7	signed Shift 20
SIZEOF shift 117	sizeof Shift 55
STATIC shift 8	static Shift 21
STRINGLITERAL shift 118	StringLiteral Shift 73
STRUCT shift 32	struct Shift 26
SWITCH shift 155	switch Shift 266
UNION shift 33	union Shift 28
UNSIGNED shift 12	unsigned Shift 30
VOLATILE shift 13	volatile Shift 31
WHILE shift 156	while Shift 274
CHAR reduce 64	char Reduce 63
DOUBLE reduce 64	double Reduce 63
FLOAT reduce 64	float Reduce 63
INT reduce 64	int Reduce 63
LONG reduce 64	long Reduce 63
SHORT reduce 64	short Reduce 63
VOID reduce 64	void Reduce 63
var_decl goto 157	<Base> Goto 32
block goto 158	<Block> Goto 278
type goto 57	<Expr> Goto 279
mod goto 25	<Mod> Goto 36
op_if goto 119	<Normal Stm> Goto 281
expr goto 159	<Op Add> Goto 77
base goto 26	<Op And> Goto 87
sign goto 27	<Op Assign> Goto 126
stm goto 329	<Op BinAND> Goto 89
normal_stm goto 161	<Op BinOR> Goto 127
value goto 121	<Op BinXOR> Goto 125
op_assign goto 122	<Op Compare> Goto 91
op_or goto 123	<Op Equate> Goto 124
op_and goto 124	<Op If> Goto 128
op_binor goto 125	<Op Mult> Goto 93
op_binxor goto 126	<Op Or> Goto 131

op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	<Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 302 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283
state 322 case_stms : CASE . value COLON stm_list case_stms (98) LPARAN shift 281 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 STRINGLITERAL shift 118 . error value goto 330	State 271 <Case Stms> ::= case • <Value> ':' <Stm List> <Case Stms> '(' Shift 84 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 StringLiteral Shift 73 <Value> Goto 272
state 323 case_stms : DEFAULT . COLON stm_list (99) COLON shift 331 . error	State 287 <Case Stms> ::= default • ':' <Stm List> ':' Shift 288
state 324 normal_stm : SWITCH LPARAN expr RPARAN LBRACE case_stms . RBRACE (88) RBRACE shift 332 . error	State 291 <Normal Stm> ::= switch '(' <Expr> ')' ' {' <Case Stms> • '}' '}' Shift 292
state 325 stm : FOR LPARAN arg SEMI arg SEMI arg . RPARAN stm (81) RPARAN shift 333 . error	State 240 <Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> • ')' <Stm> ')' Shift 241
state 326 stm : FOR LPARAN arg . SEMI arg SEMI arg RPARAN stm (81) then_stm : FOR LPARAN arg . SEMI arg SEMI arg RPARAN then_stm (85) SEMI shift 334 . error	State 253 <Stm> ::= for '(' <Arg> • ';' <Arg> ';' <Arg> ')' <Stm> <Then Stm> ::= for '(' <Arg> • ';' <Arg> ';' <Arg> ')' <Then Stm> ';' Shift 254
state 327 stm : IF LPARAN expr . RPARAN stm (78) stm : IF LPARAN expr . RPARAN then_stm ELSE stm (79) then_stm : IF LPARAN expr . RPARAN then_stm ELSE then_stm (83) expr : expr . COMMA op_assign (104) RPARAN shift 335 COMMA shift 194 . error	State 261 <Stm> ::= if '(' <Expr> • ')' <Stm> <Stm> ::= if '(' <Expr> • ')' <Then Stm> else <Stm> <Then Stm> ::= if '(' <Expr> • ')' <Then Stm> else <Then Stm> <Expr> ::= <Expr> • ',' <Op Assign> ')' Shift 262 ',' Shift 76

<pre> state 328 stm : WHILE LPARAN expr . RPARAN stm (80) then_stm : WHILE LPARAN expr . RPARAN then_stm (84) expr : expr . COMMA op_assign (104) RPARAN shift 336 COMMA shift 194 . error </pre>	<pre> State 295 <Stm> ::= while '(' <Expr> • ')' <Stm> <Then Stm> ::= while '(' <Expr> • ')' <Then Stm> <Expr> ::= <Expr> • ',' <Op Assign> ')' Shift 296 ',' Shift 76 </pre>
<pre> state 329 stm : IF LPARAN expr RPARAN then_stm ELSE stm . (79) . reduce 79 </pre>	<pre> State 302 <Stm> ::= if '(' <Expr> ')' <Then Stm> else <Stm> • '-' Reduce 78 '--' Reduce 78 '!' Reduce 78 '&' Reduce 78 '(' Reduce 78 '*' Reduce 78 ';' Reduce 78 {' Reduce 78 '}' Reduce 78 '~' Reduce 78 '++' Reduce 78 auto Reduce 78 break Reduce 78 case Reduce 78 char Reduce 78 CharLiteral Reduce 78 const Reduce 78 continue Reduce 78 DecLiteral Reduce 78 default Reduce 78 do Reduce 78 double Reduce 78 enum Reduce 78 extern Reduce 78 float Reduce 78 FloatLiteral Reduce 78 for Reduce 78 goto Reduce 78 HexLiteral Reduce 78 Id Reduce 78 if Reduce 78 int Reduce 78 long Reduce 78 OctLiteral Reduce 78 register Reduce 78 return Reduce 78 short Reduce 78 signed Reduce 78 sizeof Reduce 78 static Reduce 78 StringLiteral Reduce 78 struct Reduce 78 switch Reduce 78 union Reduce 78 unsigned Reduce 78 void Reduce 78 volatile Reduce 78 while Reduce 78 </pre>
<pre> state 330 case_stms : CASE value . COLON stm_list </pre>	<pre> State 272 <Case Stms> ::= case <Value> • ':' <Stm> </pre>

case_stms (98)	List> <Case Stms>
COLON shift 337 . error	':' Shift 273
state 331 case_stms : DEFAULT COLON . stm_list (99) sign : . (64) stm_list : . (103) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 150 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 153 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 156 RBRACE reduce 103 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26	State 288 <Case Stms> ::= default ':' • <Stm List> <Sign> ::= • <Stm List> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 DecLiteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 234 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 247 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 274 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 '}' Reduce 102 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 281 <Op Add> Goto 77 <Op And> Goto 87

<pre> sign goto 27 stm goto 160 normal_stm goto 161 value goto 121 stm_list goto 338 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 284 <Stm List> Goto 289 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283 </pre>
<pre> state 332 normal_stm : SWITCH LPARAN expr RPARAN LBRACE case_stms RBRACE . (88) . reduce 88 </pre>	<pre> State 292 <Normal Stm> ::= switch '(' <Expr> ')' '{' <Case Stms> '}' • '-' Reduce 87 '--' Reduce 87 '!' Reduce 87 '&' Reduce 87 '(' Reduce 87 '*' Reduce 87 ';' Reduce 87 '{' Reduce 87 '}' Reduce 87 '~' Reduce 87 '++' Reduce 87 auto Reduce 87 break Reduce 87 case Reduce 87 char Reduce 87 CharLiteral Reduce 87 const Reduce 87 continue Reduce 87 DeclLiteral Reduce 87 default Reduce 87 do Reduce 87 double Reduce 87 else Reduce 87 enum Reduce 87 extern Reduce 87 float Reduce 87 FloatLiteral Reduce 87 for Reduce 87 goto Reduce 87 HexLiteral Reduce 87 Id Reduce 87 if Reduce 87 int Reduce 87 long Reduce 87 OctLiteral Reduce 87 register Reduce 87 return Reduce 87 short Reduce 87 signed Reduce 87 sizeof Reduce 87 static Reduce 87 StringLiteral Reduce 87 </pre>

	struct Reduce 87 switch Reduce 87 union Reduce 87 unsigned Reduce 87 void Reduce 87 volatile Reduce 87 while Reduce 87
state 333 stm : FOR LPARAN arg SEMI arg SEMI arg RPARAN . stm (81) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 150 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 153 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 156 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159	State 241 <Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> ')' • <Stm> <Sign> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 Decliteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 234 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 247 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 274 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 281 <Op Add> Goto 77

<pre> base goto 26 sign goto 27 stm goto 339 normal_stm goto 161 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 304 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283 </pre>
<pre> state 334 stm : FOR LPARAN arg SEMI . arg SEMI arg RPARAN stm (81) then_stm : FOR LPARAN arg SEMI . arg SEMI arg RPARAN then_stm (85) arg : . (97) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 SEMI reduce 97 op_if goto 119 expr goto 286 arg goto 340 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> State 254 <Stm> ::= for '(' <Arg> ';' • <Arg> ';' <Arg> ')' <Stm> <Then Stm> ::= for '(' <Arg> ';' • <Arg> ';' <Arg> ')' <Then Stm> <Arg> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 ';' Reduce 96 <Arg> Goto 255 <Expr> Goto 306 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96 </pre>
<pre> state 335 stm : IF LPARAN expr RPARAN . stm (78) stm : IF LPARAN expr RPARAN . then_stm </pre>	<pre> State 262 <Stm> ::= if '(' <Expr> ')' • <Stm> <Stm> ::= if '(' <Expr> ')' • <Then </pre>

<pre> ELSE stm (79) then_stm : IF LPARAN expr RPARAN . then_stm ELSE then_stm (83) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 308 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 309 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 310 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 311 then_stm goto 341 normal_stm goto 313 value goto 121 op_assign goto 122 </pre>	<pre> Stm> else <Stm> <Then Stm> ::= if '(' <Expr> ')' • <Then Stm> else <Then Stm> <Sign> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 DecLiteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 251 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 259 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 293 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 297 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 </pre>
--	---

op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	<Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 299 <Then Stm> Goto 300 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283
state 336 stm : WHILE LPARAN expr RPARAN . stm (80) then_stm : WHILE LPARAN expr RPARAN . then_stm (84) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 308 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 309 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 310 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64	State 296 <Stm> ::= while '(' <Expr> ')' . <Stm> <Then Stm> ::= while '(' <Expr> ')' . <Then Stm> <Sign> ::= . '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 Decliteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 251 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 259 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 293 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63

<pre> VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 315 then_stm goto 342 normal_stm goto 313 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 297 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 282 <Then Stm> Goto 298 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283 </pre>
<pre> state 337 case_stms : CASE value COLON . stm_list case_stms (98) sign : . (64) stm_list : . (103) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 150 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 153 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 </pre>	<pre> State 273 <Case Stms> ::= case <Value> ':' • <Stm List> <Case Stms> <Sign> ::= • <Stm List> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 Decliteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 234 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 247 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 </pre>

<p> STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 156 RBRACE reduce 103 CASE reduce 103 CHAR reduce 64 DEFAULT reduce 103 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 160 normal_stm goto 161 value goto 121 stm_list goto 343 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </p>	<p> StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 274 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 '}' Reduce 102 case Reduce 102 default Reduce 102 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 281 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 284 <Stm List> Goto 286 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283 </p>
<p> state 338 case_stms : DEFAULT COLON stm_list . (99) . reduce 99 </p>	<p> State 289 <Case Stms> ::= default ':' <Stm List> . '}' Reduce 98 </p>
<p> state 339 stm : FOR LPARAN arg SEMI arg SEMI arg RPARAN stm . (81) . reduce 81 </p>	<p> State 304 <Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> ')' <Stm> . '-' Reduce 80 '--' Reduce 80 '!' Reduce 80 '&' Reduce 80 '(' Reduce 80 '*' Reduce 80 ';' Reduce 80 '{' Reduce 80 '}' Reduce 80 '~' Reduce 80 '++' Reduce 80 auto Reduce 80 </p>

	break Reduce 80 case Reduce 80 char Reduce 80 CharLiteral Reduce 80 const Reduce 80 continue Reduce 80 DecLiteral Reduce 80 default Reduce 80 do Reduce 80 double Reduce 80 enum Reduce 80 extern Reduce 80 float Reduce 80 FloatLiteral Reduce 80 for Reduce 80 goto Reduce 80 HexLiteral Reduce 80 Id Reduce 80 if Reduce 80 int Reduce 80 long Reduce 80 OctLiteral Reduce 80 register Reduce 80 return Reduce 80 short Reduce 80 signed Reduce 80 sizeof Reduce 80 static Reduce 80 StringLiteral Reduce 80 struct Reduce 80 switch Reduce 80 union Reduce 80 unsigned Reduce 80 void Reduce 80 volatile Reduce 80 while Reduce 80
state 340 stm : FOR LPARAN arg SEMI arg . SEMI arg RPARAN stm (81) then_stm : FOR LPARAN arg SEMI arg . SEMI arg RPARAN then_stm (85) SEMI shift 344 . error	State 255 <Stm> ::= for '(' <Arg> ';' <Arg> • ';' <Arg> ')' <Stm> <Then Stm> ::= for '(' <Arg> ';' <Arg> • ';' <Arg> ')' <Then Stm> ';' Shift 256
state 341 stm : IF LPARAN expr RPARAN then_stm . ELSE stm (79) then_stm : IF LPARAN expr RPARAN then_stm . ELSE then_stm (83) ELSE shift 345 . error	State 300 <Stm> ::= if '(' <Expr> ')' <Then Stm> • else <Stm> <Then Stm> ::= if '(' <Expr> ')' <Then Stm> • else <Then Stm> else Shift 301
state 342 then_stm : WHILE LPARAN expr RPARAN then_stm . (84) . reduce 84	State 298 <Then Stm> ::= while '(' <Expr> ')' <Then Stm> • else Reduce 83
state 343 case_stms : CASE value COLON stm_list . case_stms (98) case_stms : . (100)	State 286 <Case Stms> ::= case <Value> ':' <Stm List> • <Case Stms> <Case Stms> ::= •

CASE shift 322 DEFAULT shift 323 RBRACE reduce 100 case_stms goto 346	case Shift 271 default Shift 287 '}' Reduce 99 <Case Stms> Goto 290
state 344 stm : FOR LPARAN arg SEMI arg SEMI . arg RPARAN stm (81) then_stm : FOR LPARAN arg SEMI arg SEMI . arg RPARAN then_stm (85) arg : . (97) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 TILDE shift 109 PLUSPLUS shift 110 CHARLITERAL shift 111 DECLITERAL shift 112 FLOATLITERAL shift 113 HEXLITERAL shift 114 ID shift 115 OCTLITERAL shift 116 SIZEOF shift 117 STRINGLITERAL shift 118 RPARAN reduce 97 op_if goto 119 expr goto 286 arg goto 347 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	State 256 <Stm> ::= for '(' <Arg> ';' <Arg> ';' • <Arg> ')' <Stm> <Then Stm> ::= for '(' <Arg> ';' <Arg> ';' • <Arg> ')' <Then Stm> <Arg> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 '~' Shift 45 '++' Shift 46 CharLiteral Shift 47 DecLiteral Shift 48 FloatLiteral Shift 49 HexLiteral Shift 50 Id Shift 51 OctLiteral Shift 54 sizeof Shift 55 StringLiteral Shift 73 ')' Reduce 96 <Arg> Goto 257 <Expr> Goto 306 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Value> Goto 96
state 345 stm : IF LPARAN expr RPARAN then_stm ELSE . stm (79) then_stm : IF LPARAN expr RPARAN then_stm ELSE . then_stm (83) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146	State 301 <Stm> ::= if '(' <Expr> ')' <Then Stm> else • <Stm> <Then Stm> ::= if '(' <Expr> ')' <Then Stm> else • <Then Stm> <Sign> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227

LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 308 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 309 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13 WHILE shift 310 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 329 then_stm goto 348 normal_stm goto 313 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134	'~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 DeclLiteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 251 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 259 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30 volatile Shift 31 while Shift 293 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 297 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 302 <Then Stm> Goto 303 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283
---	--

state 346 case_stms : CASE value COLON stm_list case_stms . (98) . reduce 98	State 290 <Case Stms> ::= case <Value> ':' <Stm List> <Case Stms> • '}' Reduce 97
state 347 stm : FOR LPARAN arg SEMI arg SEMI arg . RPARAN stm (81) then_stm : FOR LPARAN arg SEMI arg SEMI arg . RPARAN then_stm (85) RPARAN shift 349 . error	State 257 <Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> • ')' <Stm> <Then Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> • ')' <Then Stm> ')' Shift 258
state 348 then_stm : IF LPARAN expr RPARAN then_stm ELSE then_stm . (83) . reduce 83	State 303 <Then Stm> ::= if '(' <Expr> ')' <Then Stm> else <Then Stm> • else Reduce 82
state 349 stm : FOR LPARAN arg SEMI arg SEMI arg RPARAN . stm (81) then_stm : FOR LPARAN arg SEMI arg SEMI arg RPARAN . then_stm (85) sign : . (64) MINUS shift 102 MINUSMINUS shift 103 EXCLAM shift 104 AMP shift 105 LPARAN shift 106 TIMES shift 107 SEMI shift 146 LBRACE shift 92 TILDE shift 109 PLUSPLUS shift 110 AUTO shift 1 BREAK shift 147 CHARLITERAL shift 111 CONST shift 2 CONTINUE shift 148 DECLITERAL shift 112 DO shift 149 ENUM shift 31 EXTERN shift 4 FLOATLITERAL shift 113 FOR shift 308 GOTO shift 151 HEXLITERAL shift 114 ID shift 152 IF shift 309 OCTLITERAL shift 116 REGISTER shift 6 RETURN shift 154 SIGNED shift 7 SIZEOF shift 117 STATIC shift 8 STRINGLITERAL shift 118 STRUCT shift 32 SWITCH shift 155 UNION shift 33 UNSIGNED shift 12 VOLATILE shift 13	State 258 <Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> • ')' <Stm> <Then Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> • ')' <Then Stm> <Sign> ::= • '-' Shift 39 '--' Shift 40 '!' Shift 41 '&' Shift 42 '(' Shift 43 '*' Shift 44 ';' Shift 228 '{' Shift 227 '~' Shift 45 '++' Shift 46 auto Shift 1 break Shift 229 CharLiteral Shift 47 const Shift 2 continue Shift 231 Decliteral Shift 48 do Shift 233 enum Shift 24 extern Shift 17 FloatLiteral Shift 49 for Shift 251 goto Shift 242 HexLiteral Shift 50 Id Shift 245 if Shift 259 OctLiteral Shift 54 register Shift 19 return Shift 263 signed Shift 20 sizeof Shift 55 static Shift 21 StringLiteral Shift 73 struct Shift 26 switch Shift 266 union Shift 28 unsigned Shift 30

<pre> WHILE shift 310 CHAR reduce 64 DOUBLE reduce 64 FLOAT reduce 64 INT reduce 64 LONG reduce 64 SHORT reduce 64 VOID reduce 64 var_decl goto 157 block goto 158 type goto 57 mod goto 25 op_if goto 119 expr goto 159 base goto 26 sign goto 27 stm goto 339 then_stm goto 350 normal_stm goto 313 value goto 121 op_assign goto 122 op_or goto 123 op_and goto 124 op_binor goto 125 op_binxor goto 126 op_binand goto 127 op_equate goto 128 op_compare goto 129 op_shift goto 130 op_add goto 131 op_mult goto 132 op_unary goto 133 op_pointer goto 134 </pre>	<pre> volatile Shift 31 while Shift 293 char Reduce 63 double Reduce 63 float Reduce 63 int Reduce 63 long Reduce 63 short Reduce 63 void Reduce 63 <Base> Goto 32 <Block> Goto 278 <Expr> Goto 279 <Mod> Goto 36 <Normal Stm> Goto 297 <Op Add> Goto 77 <Op And> Goto 87 <Op Assign> Goto 126 <Op BinAND> Goto 89 <Op BinOR> Goto 127 <Op BinXOR> Goto 125 <Op Compare> Goto 91 <Op Equate> Goto 124 <Op If> Goto 128 <Op Mult> Goto 93 <Op Or> Goto 131 <Op Pointer> Goto 81 <Op Shift> Goto 116 <Op Unary> Goto 104 <Sign> Goto 60 <Stm> Goto 304 <Then Stm> Goto 305 <Type> Goto 195 <Value> Goto 96 <Var Decl> Goto 283 </pre>
<pre> state 350 then_stm : FOR LPARAN arg SEMI arg SEMI arg RPARAN then_stm . (85) . reduce 85 </pre>	<pre> State 305 <Then Stm> ::= for '(' <Arg> ';' <Arg> ';' <Arg> ')' <Then Stm> . else Reduce 84 </pre>

Appendix E. Test Grammars

E.1. BASIC Programming Language

BASIC is one of the oldest programming language and one of the most popular. It was developed in 1964 by John G. Kemeny and Thomas E. Kurtz to teach students the basics of programming concepts. At the advent of the microcomputer, BASIC was implemented on numerous platforms such as the Commodore, Atari, Apple II, Altair, IBM PC computers. Over time, BASIC evolved into GW-BASIC, QBasic, Visual Basic, and recently Visual Basic .NET.

In practically all programming languages, the reserved word/symbol that denotes a comment is treated as a form of whitespace - having no effect in the manner in which the program runs. Once such type of comment is used to indicate the remainder of the line is to be ignored. These can be added to the end of any line without changing the meaning of the program. In C++, it is the `'/'` symbol; in BASIC '64 it is `'REM'`.

However, in the BASIC programming language, the line comment is treated like a statement. For instance, if `'REM'` was a normal line comment:

```
10 PRINT "Hello World" REM Common first program
```

would be a valid statement. However, in BASIC, this is illegal. In the example above, the comment must be added as an additional statement.

```
10 PRINT "Hello World" : REM Common first program
```

As a result, the Line Comment terminal that is used in the GOLD Parser cannot be used here. In the grammar below, a 'Remark' terminal was created that accepts all series of printable characters starting with the characters "REM ". In some implementations of BASIC, any string starting with "REM" is a comment statement. Examples include "REMARK", "REMARKABLE" and "REMODEL". This grammar requires the space.

```
! *
  BASIC '64

  Beginner's All-purpose Symbolic Instruction Code

  "It is practically impossible to teach good programming
  style to students that have had prior exposure to BASIC;
  as potential programmers they are mentally mutilated beyond
  hope of regeneration."

  - Edsger W. Dijkstra
* !

"Name"      = 'BASIC (Beginners All-purpose Symbolic Instruction Code)'
"Author"    = 'John G. Kemeny and Thomas E. Kurtz'
"Version"   = '1964 - Original - before Microsoft enhanced the'
              | 'language for the IBM PC.'

"About"     = 'BASIC is one of most common and popular teaching'
              | 'languages ever created.'

"Case Sensitive" = False
"Start Symbol"  = <Lines>

{String Chars} = {Printable} - ["]
{WS}           = {Whitespace} - {CR} - {LF}

NewLine       = {CR}{LF} | {CR}
Whitespace    = {WS}+

Remark        = REM{Space}{Printable}*
ID            = {Letter}[$%]?
String        = '''{String Chars}*'''
Integer       = {digit}+
Real          = {digit}+.{digit}+
```

```

<Lines>      ::= Integer <Statements> NewLine <Lines>
               | Integer <Statements> NewLine

<Statements> ::= <Statement> ':' <Statements>
               | <Statement>

<Statement>  ::= CLOSE '#' Integer
               | DATA <Constant List>
               | DIM ID '(' <Integer List> ')'
               | END
               | FOR ID '=' <Expression> TO <Expression>
               | FOR ID '=' <Expression> TO <Expression> STEP Integer
               | GOTO <Expression>
               | GOSUB <Expression>
               | IF <Expression> THEN <Statement>
               | INPUT <ID List>
               | INPUT '#' Integer ',' <ID List>
               | LET Id '=' <Expression>
               | NEXT <ID List>
               | OPEN <Value> FOR <Access> AS '#' Integer
               | POKE <Value List>
               | PRINT <Print list>
               | PRINT '#' Integer ',' <Print List>
               | READ <ID List>
               | RETURN
               | RESTORE
               | RUN
               | STOP
               | SYS <Value>
               | WAIT <Value List>
               | Remark

<Access>     ::= INPUT
               | OUPUT

<ID List>    ::= ID ',' <ID List>
               | ID

<Value List> ::= <Value> ',' <Value List>
               | <Value>

<Constant List> ::= <Constant> ',' <Constant List>
                  | <Constant>

<Integer List>  ::= Integer ',' <Integer List>
                  | Integer

<Expression List> ::= <Expression> ',' <Expression List>
                    | <Expression>

```


E.2. ANSI C Programming Language

The C programming language evolved at Bell Labs from a series of programming languages: 'CPL', 'BCPL', and then 'B'. As a result, C's development was a combined effort between Dennis Ritchie, Ken Thompson, and Martin Richards.

C was designed for the creation and implementation of low-level systems such as operating systems, device drivers, firmware, etc... To realize this goal, the language contains the ability to perform operations directly on memory and has direct access to system pointers. While this gives an enormous amount of control and flexibility, it also makes C a professional programming language - not to be used by an inexperienced programmer.

C (and later C++) quickly became the de facto standard for developing operating systems, applications and most other large projects. UNIX as well as Windows, Linux, and Mac-OS X were developed using this language (and its successors).

C is not a line-based grammar with the notable exception of compiler directives (which are preceded by a '#' character). These are usually not handled directly by the actual parser, but, rather, the pre-processor. Before the program is analyzed by the parser, C compilers scan the code and act on these commands. The final C program is then passed to the parser. This grammar does not contain the compiler directives.

```

"Name"      = 'ANSI C'
"Version"   = '1973'
"Author"    = 'Dennis Ritchie, Ken Thompson, Martin Richards'
"About"     = 'C is one of the most common, and complex,
              | 'programming languages in use today.'

"Case Sensitive" = True
"Start Symbol"  = <Decls>

{Hex Digit}    = {Digit} + [abcdefABCDEF]
{Oct Digit}    = [01234567]

{Id Head}      = {Letter} + [_]
{Id Tail}      = {Id Head} + {Digit}

{String Ch}    = {Printable} - ["]
{Char Ch}      = {Printable} - ['']

DeclLiteral    = [123456789]{digit}*
OctLiteral     = 0{Oct Digit}*
HexLiteral     = 0x{Hex Digit}+
FloatLiteral   = {Digit}*'.'{Digit}+

StringLiteral  = '"' ( {String Ch} | '\\' {Printable} ) * '"'
CharLiteral    = ' ' ( {Char Ch} | '\\' {Printable} ) ' '

Id             = {Id Head}{Id Tail}*

Comment Start  = '/*'
Comment End    = '*/'
Comment Line   = '// '

!=====

<Decls> ::= <Decl> <Decls>
        |

<Decl>  ::= <Func Decl>
        | <Func Proto>
        | <Struct Decl>
        | <Union Decl>
        | <Enum Decl>
        | <Var Decl>
        | <Typedef Decl>

!===== Function Declaration

<Func Proto> ::= <Func ID> '(' <Types> ')' ';'

```

```

        | <Func ID> '(' <Params> ')' ';'
        | <Func ID> '(' ' ' ')' ';'

<Func Decl> ::= <Func ID> '(' <Params> ')' <Block>
              | <Func ID> '(' <Id List> ')' <Struct Def> <Block>
              | <Func ID> '(' ' ' ')' <Block>

<Params>    ::= <Param> ',' <Params>
              | <Param>

<Param>     ::= const <Type> ID
              |      <Type> ID

<Types>     ::= <Type> ',' <Types>
              | <Type>

<Id List>   ::= Id ',' <Id List>
              | Id

<Func ID>   ::= <Type> ID
              | ID

!===== Type Declaration
<Typedef Decl> ::= typedef <Type> ID ';'

<Struct Decl>  ::= struct Id '{' <Struct Def> '}' ';'

<Union Decl>   ::= union Id '{' <Struct Def> '}' ';'

<Struct Def>   ::= <Var Decl> <Struct Def>
              | <Var Decl>

!===== Variable Declaration

<Var Decl>     ::= <Mod> <Type> <Var> <Var List> ';'
                  |      <Type> <Var> <Var List> ';'
                  | <Mod>      <Var> <Var List> ';'

<Var>          ::= ID <Array>
                  | ID <Array> '=' <Op If>

<Array>        ::= '[' <Expr> ']'
                  | '[' ' ' ']'

<Var List>     ::= ',' <Var Item> <Var List>
                  |

```

```

<Var Item> ::= <Pointers> <Var>

<Mod>      ::= extern
              | static
              | register
              | auto
              | volatile
              | const

!===== Enumerations

<Enum Decl> ::= enum Id '{' <Enum Def> '}' ';'

<Enum Def>  ::= <Enum Val> ',' <Enum Def>
              | <Enum Val>

<Enum Val>  ::= Id
              | Id '=' OctLiteral
              | Id '=' HexLiteral
              | Id '=' DecLiteral

!===== Types

<Type>      ::= <Base> <Pointers>

<Base>      ::= <Sign> <Scalar>
              | struct Id
              | struct '{' <Struct Def> '}'
              | union Id
              | union '{' <Struct Def> '}'
              | enum Id

<Sign>      ::= signed
              | unsigned
              |

<Scalar>    ::= char
              | int
              | short
              | long
              | short int
              | long int
              | float
              | double
              | void

```

```

<Pointers> ::= '*' <Pointers>
            |

!===== Statements

<Stm>      ::= <Var Decl>
            | Id ':' !Label
            | if '(' <Expr> ')' <Stm>
            | if '(' <Expr> ')' <Then Stm> else <Stm>
            | while '(' <Expr> ')' <Stm>
            | for '(' <Arg> ';' <Arg> ';' <Arg> ')' <Stm>
            | <Normal Stm>

<Then Stm> ::= if '(' <Expr> ')' <Then Stm> else <Then Stm>
            | while '(' <Expr> ')' <Then Stm>
            | for '(' <Arg> ';' <Arg> ';' <Arg> ')' <Then Stm>
            | <Normal Stm>

<Normal Stm> ::= do <Stm> while '(' <Expr> ')'
            | switch '(' <Expr> ')' '{' <Case Stms> '}'
            | <Block>
            | <Expr> ';'
            | goto Id ';'
            | break ';'
            | continue ';'
            | return <Expr> ';'
            | ';' !Null statement

<Arg>      ::= <Expr>
            |

<Case Stms> ::= case <Value> ':' <Stm List> <Case Stms>
            | default ':' <Stm List>
            |

<Block>    ::= '{' <Stm List> '}'

<Stm List> ::= <Stm> <Stm List>
            |

! =====
! Here begins the C's 15 levels of operator precedence.
! =====

<Expr>     ::= <Expr> ',' <Op Assign>
            | <Op Assign>

<Op Assign> ::= <Op If> '=' <Op Assign>

```

		<Op If> '+' <Op Assign>
		<Op If> '-=' <Op Assign>
		<Op If> '*=' <Op Assign>
		<Op If> '/=' <Op Assign>
		<Op If> '^=' <Op Assign>
		<Op If> '&=' <Op Assign>
		<Op If> ' =' <Op Assign>
		<Op If> '>>=' <Op Assign>
		<Op If> '<<=' <Op Assign>
		<Op If>
<Op If>	::=	<Op Or> '?' <Op If> ':' <Op If>
		<Op Or>
<Op Or>	::=	<Op Or> ' ' <Op And>
		<Op And>
<Op And>	::=	<Op And> '&&' <Op BinOR>
		<Op BinOR>
<Op BinOR>	::=	<Op BinOr> ' ' <Op BinXOR>
		<Op BinXOR>
<Op BinXOR>	::=	<Op BinXOR> '^' <Op BinAND>
		<Op BinAND>
<Op BinAND>	::=	<Op BinAND> '&' <Op Equate>
		<Op Equate>
<Op Equate>	::=	<Op Equate> '==' <Op Compare>
		<Op Equate> '!=' <Op Compare>
		<Op Compare>
<Op Compare>	::=	<Op Compare> '<' <Op Shift>
		<Op Compare> '>' <Op Shift>
		<Op Compare> '<=' <Op Shift>
		<Op Compare> '>=' <Op Shift>
		<Op Shift>
<Op Shift>	::=	<Op Shift> '<<' <Op Add>
		<Op Shift> '>>' <Op Add>
		<Op Add>
<Op Add>	::=	<Op Add> '+' <Op Mult>
		<Op Add> '-' <Op Mult>
		<Op Mult>
<Op Mult>	::=	<Op Mult> '*' <Op Unary>
		<Op Mult> '/' <Op Unary>
		<Op Mult> '%' <Op Unary>

```

| <Op Unary>

<Op Unary> ::= '!' <Op Unary>
| '~' <Op Unary>
| '-' <Op Unary>
| '*' <Op Unary>
| '&' <Op Unary>
| '++' <Op Unary>
| '--' <Op Unary>
| <Op Pointer> '++'
| <Op Pointer> '--'
| '(' <Type> ')' <Op Unary> !CAST
| sizeof '(' <Type> ')'
| sizeof '(' ID <Pointers> ')'
| <Op Pointer>

<Op Pointer> ::= <Op Pointer> '.' <Value>
| <Op Pointer> '->' <Value>
| <Op Pointer> '[' <Expr> ']'
| <Value>

<Value> ::= OctLiteral
| HexLiteral
| DecLiteral
| StringLiteral
| CharLiteral
| FloatLiteral
| Id '(' <Expr> ')'
| Id '(' ' ' ')'

| Id
| '(' <Expr> ')'

```

E.3. COBOL Programming Language

The COBOL (Common Business Oriented Language) programming language is one of the oldest still in use today. It was originally designed by the United States Department of defense under the supervision of the Conference on Data Systems Languages (CODASYL) Committee. Most of the groundwork and design of COBOL was done by General Grace Hopper of the United States Navy.

The COBOL language was designed to be a self-documenting language where programs would read as close to English as possible. All interaction between the user, the terminal and files are performed through language statements rather than third-party libraries. The metaphor used for data types is abstract and completely platform independent. As a result of these factors, COBOL is a very portable. The COBOL grammar is complex - containing, on average, over 200 reserved words and over 500 rules.

```
"Name"      = 'COBOL'
"Version"   = 'Current De-Facto Format'
"Author"    = 'General Grace Hopper (United States Navy) and the CODASYL Committee'
"About"     = 'Originally created in 1960, COBOL is one of the top 5 most popular'
              | 'languages in use today.'

"Case Sensitive" = False
"Start Symbol" = <Program>

{String Chars 1} = {Printable} - ['']
{String Chars 2} = {Printable} - [""]
{Id Tail Chars}  = {Alphanumeric} + [-]

{Pic Ch} = [ax9bpz0s,*+-$()]

StringLiteral = ''{String Chars 1}*'' | '''{String Chars 1}*'''
Integer       = {Digit}+
FloatLiteral  = {Digit}+.{Digit}+
Identifier    = {Letter}{Id Tail Chars}*

PictureSequence = pic(ture)? {Whitespace}+ (is)? {Whitespace}+ {Pic Ch}+ ([.v] {Pic Ch})?

<Program> ::= <Identification Division> <Environment Division> <Data Division>
```


<Procedure Division>

!-----
! Optional Keywords - Used for readability only
!-----

<Is Opt> ::= IS
| ARE
|

<To Opt> ::= TO
|

<By Opt> ::= BY
|

<On Opt> ::= ON
|

<Comma Opt> ::= ','
|

<Then Opt> ::= THEN
|

<Record Opt> ::= RECORD
|

<With Opt> ::= WITH
|

<In Opt> ::= IN
|

<Mode Opt> ::= MODE
|

<Than Opt> ::= THAN
|

<Key Opt> ::= KEY
|

!-----
! Shared By Multiple Rules
!-----

<At Opt> ::= AT '(' <Numeric> ',' <Numeric> ')'
| AT <Numeric>
|

<Giving Opt> ::= GIVING Identifier
|

<Using Opt> ::= USING <Using Id List>
|

<Using Id List> ::= <Using Id Item> ',' <Using Id List>
| <Using Id Item>

<Using Id Item> ::= Identifier
| Identifier BY REFERENCE
| Identifier BY CONTENT

```

<Paragraphs> ::= Identifier SECTION '.' <Paragraphs>
               | Identifier '.' <Sentences> <Paragraphs>
               |

<Not Opt>    ::= NOT
               |

<From Opt>   ::= FROM Identifier
               |

<Pointer Clause> ::= <With Opt> POINTER <Variable>
                   |

!-----
! Values, Literals, etc...
!-----

<Symbolic Value> ::= <Literal>
                   | <Variable>
                   | <Figurative>

<Value>        ::= <Literal>
                   | <Variable>

<Numeric>      ::= Integer
                   | Identifier

<Literal>      ::= Integer
                   | FloatLiteral
                   | StringLiteral
                   | QUOTE
                   | QUOTES

<Figurative>   ::= ZERO
                   | ZEROS
                   | ZEROES
                   | SPACE
                   | SPACES
                   | 'HIGH-VALUE'
                   | 'HIGH-VALUES'
                   | 'LOW-VALUE'
                   | 'LOW-VALUES'
                   | ALL StringLiteral
                   | NULL
                   | NULLS

<Variables>   ::= Identifier ',' <Variables>
                   | Identifier

<Variable>    ::= Identifier
                   | Identifier '(' <Subsets> ')'

<Subsets>     ::= <Numeric> ':' <Subsets>
                   | <Numeric>

!-----
!----- IDENTIFICATION DIVISION -----
!-----

<Identification Division> ::= IDENTIFICATION DIVISION '.' <Program Info List>

<Program Info List> ::= <Program Info Clause> <Program Info List>

```

```

|
<Program Info Clause> ::= AUTHOR '.' <Word List> '.'
                        | INSTALLATION '.' <Word List> '.'
                        | 'DATE-WRITTEN' '.' <Word List> '.'
                        | 'DATE-COMPILED' '.' <Word List> '.'
                        | SECURITY '.' <Word List> '.'
                        | 'PROGRAM-ID' '.' <Word List> <Prg Name Opt> '.'

<Word List> ::= <Word List Item> <Word List>
              | <Word List Item>

<Word List Item> ::= Identifier
                  | Integer
                  | FloatLiteral
                  | StringLiteral
                  | '/'

<Prg Name Opt> ::= <Is Opt> <Common Initial> <Optional Program>
                 |

<Common Initial> ::= COMMON
                  | INITIAL

<Optional Program> ::= PROGRAM
                   |

!-----
!----- ENVIRONMENT DIVISION -----
!-----

<Environment Division> ::= ENVIRONMENT DIVISION '.' <Config Section> <Input-Output
Section>

!-----
! CONFIGURATION SECTION
!-----

<Config Section>      ::= CONFIGURATION SECTION '.' <Source Computer> <Object Computer>
<Special Names>
|
<Source Computer>    ::= 'SOURCE-COMPUTER' '.' Identifier <Source Debug Opt> '.'
|
<Source Debug Opt>   ::= <With Opt> DEBUGGING MODE
|
<Object Computer>    ::= 'OBJECT-COMPUTER' '.' Identifier <Memory Opt> '.'
|
<Memory Opt>         ::= MEMORY <Size Opt> Integer <Memsize Args>
|
<Size Opt>           ::= SIZE
|
<Memsize Args>      ::= WORDS
                      | CHARACTERS
                      | MODULES

```

```

<Special Names> ::= 'SPECIAL-NAMES' '.' <Special Name List>
|

<Special Name List> ::= Identifier IS Identifier '.' <Special Name List>
|

!-----
! INPUT-OUTPUT SECTION
!-----

<Input-Output Section> ::= 'INPUT-OUTPUT' SECTION '.' <File Control>
|

<File Control> ::= 'FILE-CONTROL' '.' <Select Block>
|

<Select Block> ::= <Select Statement> <Select Block>
|

<Select Statement> ::= SELECT <Optional Opt> Identifier ASSIGN <To Opt> Identifier
<Select Opt List> '.'

<Optional Opt> ::= OPTIONAL
|

<Select Opt List> ::= <Select Option> <Select Opt List>
|

<Select Option> ::= <File Opt> STATUS <Is Opt> Identifier
| ACCESS <Mode Opt> <Is Opt> <Access Mode>
| ORGANIZATION <Is Opt> <Organization Kind>

<File Opt> ::= FILE
|

<Access Mode> ::= SEQUENTIAL
| RANDOM
| DYNAMIC

<Organization Kind> ::= SEQUENTIAL
| LINE SEQUENTIAL
| RELATIVE <Relative Key Opt>
| INDEXED <Record Key Opt>

<Relative Key Opt> ::= <Key Opt> <Is Opt> Identifier
|

<Record Key Opt> ::= RECORD KEY <Is Opt> Identifier
|

!-----
!----- DATA DIVISION -----
!-----

<Data Division> ::= DATA DIVISION '.' <Data Section List>

<Data Section List> ::= <Data Section Entry> <Data Section List>
|

<Data Section Entry> ::= <File Section>
| <Working-Storage Section>
| <Linkage Section>

```

```

| <Screen Section>

!-----
! Record Definition
!-----

<Record Entry Block> ::= Integer <Level Name> <Record Type> '.' <Record Entry Block>
| Integer <Level Name> <Record Type> '.'

<Level Name>      ::= Identifier
| FILLER

<Record Type>     ::= <Array Options> PictureSequence <Record Value Opt> <Record Pic
Options>
| REDEFINES Identifier

<Array Options>   ::= OCCURS Integer <Times Opt> INDEXED <By Opt> Identifier
|

<Times Opt>      ::= TIMES
|

<Record Pic Options> ::= USAGE <Is Opt> <Pic Usage Args>
| SIGN <Is Opt> <Sign Args> <Sep Char Option>
| SYNC <Left Right Opt>
| JUSTIFIED <Left Right Opt>
|

<Pic Usage Args> ::= BINARY
| COMPUTATIONAL
| COMP
| DISPLAY
| INDEX
| 'PACKED-DECIMAL'

<Sign Args>      ::= LEADING
| TRAILING

<Sep Char Option> ::= SEPARATE <Character Opt>
|

<Character Opt>  ::= CHARACTER
|

<Left Right Opt> ::= LEFT
| RIGHT
|

<Record Value Opt> ::= VALUE <Is Opt> <Symbolic Value>
|

!-----
! FILE SECTION
!-----

<File Section> ::= FILE SECTION '.' <File Desc Block>

<File Desc Block> ::= <File Desc Entry> <File Desc Block>
|

<File Desc Entry> ::= <File Desc Type> Identifier <File Label Entry> <File Name Entry>
'.' <Record Entry Block>

```

```

<File Desc Type> ::= FD
                  | SD

<File Label Entry> ::= LABEL RECORD <Is Opt> <File Label Type>
                  |

<File Label Type> ::= STANDARD
                  | OMITTED

<File Name Entry> ::= VALUE <Prep Opt> <Is Opt> <File Name>
                  |

<File Name> ::= Identifier
              | StringLiteral

<Prep Opt> ::= OF 'FILE-ID'
            |

!-----
! WORKING-STORAGE SECTION
!-----

<Working-Storage Section> ::= 'WORKING-STORAGE' SECTION '.' <Record Entry Block>

!-----
! LINKAGE SECTION
!-----

<Linkage Section> ::= LINKAGE SECTION '.' <Record Entry Block>

!-----
! SCREEN SECTION
!-----

<Screen Section> ::= SCREEN SECTION '.' <Screen Field List>

<Screen Field List> ::= <Screen Field> <Screen Field List>
                    | <Screen Field>

<Screen Field> ::= Integer <Field Name opt> <Field Def List> '.'

<Field Name opt> ::= Identifier
                 |

<Field Def List> ::= <Field Def Clause> <Field Def List>
                 |

<Field Def Clause> ::= LINE Integer
                   | COLUMN Integer
                   | 'FOREGROUND-COLOR' Integer
                   | 'BACKGROUND-COLOR' Integer
                   | VALUE <Is Opt> <Symbolic Value>
                   | PictureSequence
                   | FROM Identifier
                   | USING Identifier
                   | HIGHLIGHT
                   | 'REVERSE-VIDEO'
                   | BLINK

```

	UNDERLINE
	BLANK SCREEN
!-----	
!----- PROCEDURE DIVISION -----	
!-----	
<Procedure Division> ::= PROCEDURE DIVISION <Using Opt> '.' <Paragraphs>	
!-----	
! Sentences	
!-----	
! If you add statements to the grammar, make sure to update the corresponding	
! <Statement> rules below	
<Sentences> ::= <Sentence> '.' <Sentences>	
<Sentence> ::= <Accept Stm>	
<Add Stm>	
<Add Stm Ex> <End-Add Opt>	
<Call Stm>	
<Call Stm Ex> <End-Call Opt>	
<Close Stm>	
<Compute Stm>	
<Compute Stm Ex> <End-Compute Opt>	
<Display Stm>	
<Divide Stm>	
<Divide Stm Ex> <End-Divide Opt>	
<Evaluate Stm> <End-Evaluate Opt>	
<If Stm> <End-If Opt>	
<Move Stm>	
<Move Stm Ex> <End-Move Opt>	
<Multiply Stm>	
<Multiply Stm Ex> <End-Multiply Opt>	
<Open Stm>	
<Perform Stm>	
<Perform Stm Ex> <End Perform Opt>	
<Read Stm>	
<Read Stm Ex> <End-Read Opt>	
<Release Stm>	
<Rewrite Stm>	
<Rewrite Stm Ex> <End-Rewrite Opt>	
<Set Stm>	
<Start Stm>	
<Start Stm Ex> <End-Start Opt>	
<String Stm>	
<String Stm Ex> <End-String Opt>	
<Subtract Stm>	
<Subtract Stm Ex> <End-Subtract Opt>	
<Write Stm>	
<Write Stm Ex> <End-Write Opt>	
<Unstring Stm>	
<Unstring Stm Ex> <End-Unstring Opt>	
<Misc Stm>	
<End-Add Opt> ::= 'END-ADD'	
<End-Call Opt> ::= 'END-CALL'	

```

<End-Compute Opt> ::= 'END-COMPUTE'
                    |
<End-Divide Opt>   ::= 'END-DIVIDE'
                    |
<End-Evaluate Opt> ::= 'END-EVALUATE'
                    |
<End-If Opt>       ::= 'END-IF'
                    |
<End-Move Opt>     ::= 'END-MOVE'
                    |
<End-Multiply Opt> ::= 'END-MULTIPLY'
                    |
<End Perform Opt>  ::= 'END-PERFORM'
                    |
<End-Read Opt>     ::= 'END-READ'
                    |
<End-Rewrite Opt>  ::= 'END-REWRITE'
                    |
<End-Start Opt>    ::= 'END-START'
                    |
<End-String Opt>   ::= 'END-STRING'
                    |
<End-Subtract Opt> ::= 'END-SUBTRACT'
                    |
<End-Write Opt>    ::= 'END-WRITE'
                    |
<End-Unstring Opt> ::= 'END-UNSTRING'
                    |

!-----
! Statements - parts of a sentence
!-----
! If you add statements to the grammar, make sure to update the corresponding
! <Sentence> rules above

<Statements> ::= <Statement> <Statements>
               | <Statement>

<Statement> ::= <Accept Stm>
               | <Add Stm>
               | <Add Stm Ex>      'END-ADD'
               | <Call Stm>
               | <Call Stm Ex>     'END-CALL'
               | <Close Stm>
               | <Compute Stm>
               | <Compute Stm Ex>  'END-COMPUTE'
               | <Display Stm>
               | <Divide Stm>
               | <Divide Stm Ex>   'END-DIVIDE'
               | <Evaluate Stm>   'END-EVALUATE'

```


<If Stm>	'END-IF'
<Move Stm>	
<Move Stm Ex>	'END-MOVE'
<Multiply Stm>	
<Multiply Stm Ex>	'END-MULTIPLY'
<Open Stm>	
<Perform Stm>	
<Perform Stm Ex>	'END-PERFORM'
<Read Stm>	
<Read Stm Ex>	'END-READ'
<Release Stm>	
<Rewrite Stm>	
<Rewrite Stm Ex>	'END-REWRITE'
<Set Stm>	
<Start Stm>	
<Start Stm Ex>	'END-START'
<String Stm>	
<String Stm Ex>	'END-STRING'
<Subtract Stm>	
<Subtract Stm Ex>	'END-SUBTRACT'
<Write Stm>	
<Write Stm Ex>	'END-WRITE'
<Unstring Stm>	
<Unstring Stm Ex>	'END-UNSTRING'
<Misc Stm>	
!-----	
! Boolean Expressions	
!-----	
! Note: <Not Opt> is very significant in the operators below	
<Boolean Exp>	::= <And Exp> OR <Boolean Exp> <And Exp>
<And Exp>	::= <Negation Exp> And <And Exp> <Negation Exp>
<Negation Exp>	::= <Compare Exp> NOT <Compare Exp>
<Compare Exp>	::= <Symbolic Value> <Compare Op> <Symbolic Value> <Symbolic Value> IS ALPHABETIC <Symbolic Value> IS 'ALPHABETIC-UPPER' <Symbolic Value> IS 'ALPHABETIC-LOWER' '(' <Boolean Exp> ')' <Symbolic Value>
<Compare Op>	::= <Is Opt> <Greater Op> <Is Opt> NOT <Greater Op> <Is Opt> <Greater Eq Op> <Is Opt> <Less Op> <Is Opt> NOT <Less Op> <Is Opt> <Less Eq Op> <Is Opt> <Equal Op> <Is Opt> NOT <Equal Op>
<Greater Op>	::= GREATER <Than Opt> '>'
<Greater Eq Op>	::= GREATER <Than Opt> OR EQUAL <To Opt> '>='
<Less Op>	::= LESS <Than Opt>

```

| '<'
<Less Eq Op> ::= LESS <Than Opt> OR EQUAL <To Opt>
| '<='
<Equal Op> ::= EQUAL <To Opt>
| '='

!-----
! Clauses - Used by Numerous Statements
!-----

<Size Clause> ::= <Not Opt> <On Opt> SIZE ERROR <Statements>
<Invalid Clause> ::= <Not Opt> INVALID <Key Opt> <Statements>
<Exception Clause> ::= <Not Opt> <On Opt> EXCEPTION <Statements>
<Overflow Clause> ::= <Not Opt> <On Opt> OVERFLOW <Statements>
<At End Clause> ::= <Not Opt> <At Opt> END <Statements>

!-----
! ACCEPT Statement
!-----

<Accept Stm> ::= ACCEPT <At Opt> <Variables> <Accept Option>
<Accept Option> ::= FROM <From Arg>
|
<From Arg> ::= DATE
| DAY
| 'DAY-OF-WEEK'
| TIME
| CONSOLE

!-----
! ADD Statement
!-----

<Add Stm> ::= ADD <Value> TO <Variables> <Giving Opt>
<Add Stm Ex> ::= <Add Stm> <Size Clause>

!-----
! CALL Statement
!-----

<Call Stm> ::= CALL <Call List> <Using Opt>
<Call Stm Ex> ::= <Call Stm> <Exception Clause>
<Call List> ::= StringLiteral
| Identifier

!-----
! CLOSE Statement
!-----

<Close Stm> ::= CLOSE <Variables> <Close Options>
<Close Options> ::= UNIT <Close Method>
| REEL <Close Method>

```

```

| WITH LOCK
|
<Close Method> ::= FOR REMOVAL
| WITH NO REWIND

!-----
! COMPUTE Statement
!-----

<Compute Stm> ::= COMPUTE Identifier <Round Opt> <Equal Op> <Math Exp>

<Compute Stm Ex> ::= <Compute Stm> <Size Clause>

<Math Exp>      ::= <Mult Exp> '+' <Math Exp>
| <Mult Exp> '-' <Math Exp>
| <Mult Exp>

<Mult Exp>      ::= <Negate Exp> '*' <Mult Exp>
| <Negate Exp> '/' <Mult Exp>
| <Negate Exp>

<Negate Exp>    ::= '-' <Value>
| <Value>

!-----
! DISPLAY Statement
!-----

<Display Stm> ::= DISPLAY <Display Args> <At Opt> <Upon Opt> <Advancing Opt>

<Display Args> ::= <Value> <Comma Opt> <Display Args>
| <Value>

<Upon Opt> ::= UPON Identifier
|

<Advancing Opt> ::= <With Opt> NO ADVANCING
|

!-----
! DIVIDE Statement
!-----

<Divide Stm> ::= DIVIDE <Variables> <Divide Action> <Value> <Giving Opt> <Round Opt>

<Divide Stm Ex> ::= <Divide Stm> <Size Clause>

<Divide Action> ::= BY
| INTO

<Round Opt> ::= ROUNDED
|

!-----
! EVALUATE Statement
!-----

<Evaluate Stm> ::= EVALUATE <Subjects> <When Clauses>

<Subjects>      ::= <Subject> ALSO <Subjects>
| <Subject>

<Subject>       ::= TRUE

```

```

| <Boolean Exp>
<When Clauses> ::= <When Clause> <When Clauses>
| <When Clause>
<When Clause> ::= WHEN <Phrases> <Then Opt> <Statements>
| WHEN OTHER <Then Opt> <Statements>
<Phrases> ::= <Phrase> ALSO <Phrases>
| <Phrase>
<Phrase> ::= ANY
| <Symbolic Value> THROUGH <Symbolic Value>
| <Symbolic Value> THRU <Symbolic Value>
| <Symbolic Value>
!-----
! IF Statement
!-----
<If Stm> ::= IF <Boolean Exp> <Then Opt> <Statements> <If Args End>
<If Args End> ::= ELSIF <Boolean Exp> <Then Opt> <Statements> <If Args End>
| ELSE <Then Opt> <Statements>
|
!-----
! MOVE Statement
!-----
<Move Stm> ::= MOVE <Corr Opt> <Symbolic Value> TO <Variables>
<Move Stm Ex> ::= <Move Stm> <Exception Clause>
<Corr Opt> ::= CORRESPONDING
| CORR
|
!-----
! MULTIPLY Statement
!-----
<Multiply Stm> ::= MULTIPLY <Variables> BY <Value> <Giving Opt>
<Multiply Stm Ex> ::= <Multiply Stm> <Size Clause>
!-----
! OPEN Statement
!-----
<Open Stm> ::= OPEN <Open List>
<Open List> ::= <Open List> <Open Entry>
| <Open Entry>
<Open Entry> ::= INPUT <Variables> <Open Options>
| OUTPUT <Variables> <Open Options>
| EXTEND <Variables> <Open Options>
| 'I-O' <Variables> <Open Options>
<Open Options> ::= REVERSED
| WITH NO REWIND
|
!-----

```

```

! PERFORM Statement
!-----

<Perform Stm> ::= PERFORM Identifier <Thru Opt>
                | PERFORM Identifier <Thru Opt> <Numeric> TIMES

<Perform Stm Ex> ::= PERFORM VARYING <With Test> <Variable> FROM <Numeric> <Loop
Condition> <Statements>
                | PERFORM UNTIL <Boolean Exp> <Statements>

<Thru Opt> ::= THRU Identifier
              | THROUGH Identifier
              |

<With Test> ::= <With Opt> TEST AFTER
                | <With Opt> TEST BEFORE
                |

<Loop Condition> ::= TO <Numeric> BY <Numeric>
                    | BY <Numeric> <Numeric> UNTIL <Boolean Exp>

!-----
! READ Statement
!-----

<Read Stm> ::= READ Identifier <Next Opt> <Record Opt> <Into Clause>

<Read Stm Ex> ::= <Read Stm> <At End Clause>
                | <Read Stm> <Invalid Clause>

<Into Clause> ::= INTO <Variable>
                |

<Next Opt> ::= NEXT
              |

!-----
! RELEASE Statement
!-----

<Release Stm> ::= RELEASE Identifier <From Opt>

!-----
! REWRITE Statement
!-----

<Rewrite Stm> ::= REWRITE Identifier <From Opt>

<Rewrite Stm Ex> ::= <Rewrite Stm> <Invalid Clause>

!-----
! SET Statement
!-----

<Set Stm> ::= SET <Variable> TO <Set Value>

<Set Value> ::= ON
                | OFF
                | TRUE
                | <Symbolic Value>

!-----
! START Statement
!-----

```

```

<Start Stm>      ::= START Identifier <Key Clause>

<Start Stm Ex> ::= <Start Stm> <Invalid Clause>

<Key Clause>    ::= KEY <Compare Op> Identifier
                  |

!-----
! STRING Statement
!-----

<String Stm>     ::= STRING <String Items> INTO <Variable> <Pointer Clause>

<String Stm Ex> ::= <String Stm> <Overflow Clause>

<String Items>   ::= <String Item> <String Items>
                  | <String Item>

<String Item>    ::= <Value> <String Delimited>

<String Delimited> ::= DELIMITED <By Opt> <Value>
                  | DELIMITED <By Opt> SIZE

!-----
! SUBTRACT Statement
!-----

<Subtract Stm> ::= SUBTRACT <Value> FROM <Variables> <Giving Opt>

<Subtract Stm Ex> ::= <Subtract Stm> <Size Clause>

!-----
! WRITE Statement
!-----

<Write Stm> ::= WRITE Identifier <From Opt> <Write Options>

<Write Stm Ex> ::= <Write Stm> <Invalid Clause>

<Write Options> ::= AFTER ADVANCING <Numeric> LINES
                  |

!-----
! UNSTRING Statement
!-----

<Unstring Stm> ::= UNSTRING identifier <Unstring Delimiters> INTO Identifier <Unstring
Options> <Pointer Clause>

<Unstring Stm Ex> ::= <Unstring Stm> <Overflow Clause>

<Unstring Delimiters> ::= <Unstring Delimiter> <Unstring Delimiters>
                        | <Unstring Delimiter>

<Unstring Delimiter> ::= DELIMITED <By Opt> <All Opt> <Value>

<Unstring Options> ::= <Unstring Option> <Unstring Options>
                    |

<Unstring Option>  ::= DELIMITER <In Opt> Identifier
                    | COUNT <In Opt> Identifier

<Tallying Clause> ::= TALLYING <In Opt> Identifier

```

```
<All Opt> ::= ALL
           |

!-----
! Miscellaneous Statements - Simple and atomic
!-----

<Misc Stm> ::= CANCEL Identifier
              | EXIT PROGRAM
              | GO TO Identifier
              | STOP RUN
```

E.4. HTML

HTML (HyperText Markup Language) requires little explanation - being the dominant format used for websites. Like XML, HTML is a descendant of SGML (Standardized General Markup Language).

```

"Name"          = 'HTML'
"Version"       = '1.0'
"Author"        = 'W3C Recommendation'
"About"         = 'HyperText Markup Language'

"Start Symbol" = <HTML>

{ID Tail}       = {Alphanumeric} + [-/]
{Symbol Chars}  = {Printable} - {Letter} - [<>'\"=] - {Whitespace}
{String Chars}  = {Printable} - ["]
{Hex Digit}     = {Digit} + [abcdef]

StringLiteral = '"' {String Chars}* '"'

ID             = {Letter}{ID Tail}*
Symbol         = {Symbol Chars}
CharName       = '&' {Letter}+ ';'
CharNumber     = '&#' {Digit}+ ';'
Color          = '#' {Hex Digit}+

Comment Start  = '<!--'
Comment End    = '-->'

! ----- Webpage
<HTML> ::= <DocType Opt> '<html' <Parameters> '>' <Webpage Body> '</html>'

<DocType Opt> ::= '<!doctype' <Parameters> '>'
               |

<Webpage Body> ::= <Head> <Body>

<Head> ::= '<head' <Parameters> '>' <Head Items> '</head>'
        |

<Head Items> ::= <Head Item> <Head Items>
               |

<Head Item> ::= '<title'          <Parameters> '>' <Content> '</title>'

```



```

        | '<meta'          <Parameters> '>'
        | '<link'          <Parameters> '>'
        | '<base'          <Parameters> '>'
        | '<basefont'      <Parameters> '>'
        | '<script'        <Parameters> '>' <Content> '</script>'

<Body> ::= '<body' <Parameters> '>' <Content> '</body>'

! ----- Content
! The paragraph tag has special scope since it does not require an end tag

<Content> ::= '<p' <Parameters> '>' <P Content>
           | <Tag>          <Content>
           | <Word>         <Content>
           |

<P Content> ::= <Tag>          <P Content>
               | <Word>         <P Content>
               |

! -----

<Parameters> ::= <Parameter> <Parameters>
               |

<Parameter> ::= Id '=' StringLiteral
               | Id '=' Color
               | Id '=' Id
               | Id
               | StringLiteral

! ----- Tags

<Tag> ::= '<img'          <Parameters> '>'
        | '<br'           <Parameters> '>'
        | '<hr'           <Parameters> '>'
        | '<p'            <Parameters> '>' <P Content> '</p>'
        | '<h1'           <Parameters> '>' <Content> '</h1>'
        | '<h2'           <Parameters> '>' <Content> '</h2>'
        | '<h3'           <Parameters> '>' <Content> '</h3>'
        | '<h4'           <Parameters> '>' <Content> '</h4>'
        | '<h5'           <Parameters> '>' <Content> '</h5>'
        | '<h6'           <Parameters> '>' <Content> '</h6>'
        | '<a'            <Parameters> '>' <Content> '</a>'
        | '<map'          <Parameters> '>' <Map items> '</map>'
        | '<b'            <Parameters> '>' <Content> '</b>'
        | '<u'            <Parameters> '>' <Content> '</u>'
        | '<em>          <Parameters> '>' <Content> '</em>'
        | '<i'            <Parameters> '>' <Content> '</i>'
        | '<strike>       <Parameters> '>' <Content> '</strike>'
        | '<strong>       <Parameters> '>' <Content> '</strong>'
        | '<font>         <Parameters> '>' <Content> '</font>'

```

'<big'	<Parameters>	'>' <Content>	'</big>'
'<small'	<Parameters>	'>' <Content>	'</small>'
'<sup'	<Parameters>	'>' <Content>	'</sup>'
'<sub'	<Parameters>	'>' <Content>	'</sub>'
'<blockquote'	<Parameters>	'>' <Content>	'</blockquote>'
'<address'	<Parameters>	'>' <Content>	'</address>'
'<code'	<Parameters>	'>' <Content>	'</code>'
'<cite'	<Parameters>	'>' <Content>	'</cite>'
'<caption'	<Parameters>	'>' <Content>	'</caption>'
'<samp'	<Parameters>	'>' <Content>	'</samp>'
'<kbd'	<Parameters>	'>' <Content>	'</kbd>'
'<tt'	<Parameters>	'>' <Content>	'</tt>'
'<center'	<Parameters>	'>' <Content>	'</center>'
'<span'	<Parameters>	'>' <Content>	''
'<div'	<Parameters>	'>' <Content>	'</div>'
'<applet'	<Parameters>	'>' <Params>	'</applet>'
'<object'	<Parameters>	'>' <Params>	'</object>'
'<table'	<Parameters>	'>' <Table Rows>	'</table>'
'<ul'	<Parameters>	'>' <List Items>	''
'<ol'	<Parameters>	'>' <List Items>	''
'<dir'	<Parameters>	'>' <List Items>	'</dir>'
'<menu'	<Parameters>	'>' <List Items>	'</menu>'
'<dl'	<Parameters>	'>' <Def Items>	'</dl>'
'<form'	<Parameters>	'>' <Content>	'</form>'
'<input'	<Parameters>	'>'	
'<select'	<Parameters>	'>' <Options>	'</select>'
'<textarea'	<Parameters>	'>' <Content>	'</textarea>'

! ===== Table Body

<Table Rows> ::= '<tr' <Parameters> '>' <Table Cells> '</tr>' <Table Rows>
|

<Table Cells> ::= <Table Cell> <Table Cells>
|

<Table Cell> ::= '<td' <Parameters> '>' <Content> '</td>'
| '<th' <Parameters> '>' <Content> '</th>'

! ===== Definition List

<Def Items> ::= <Def Item> <Def Items>
|

<Def Item> ::= '<dt' <Parameters> '>' <Content> '</dt>'
| '<dd' <Parameters> '>' <Content> '</dd>'

```

! ===== List Body

<List Items> ::= <List Item> <List Items>
                |

<List Item> ::= '<li>' <Parameters> '>' <Content> '</li>'

! ===== Select Items

<Options> ::= <Option> <Options>
                |

<Option> ::= '<option>' <Parameters> '>'

! ===== Map Items

<Map Items> ::= <Area> <Map Items>
                |

<Area> ::= '<area>' <Parameters> '>'

! ===== Object Body

<Params> ::= '<param>' <Parameters> '>' <Params>
                |

! ===== Valid words include all the reserved symbols.

<Word> ::= Id
          | Symbol
          | '='
          | ''
          | CharName
          | CharNumber
          | Color

```

E.5. LISP Programming Language

The LISP (List Programming) language is one of the oldest still use today; second only to FORTRAN. It was invented by mathematician John McCarthy and organizes information into abstract "lists". This metaphor is used universally for both data and functions; in fact, the LISP runtime engine makes no differentiation between the two. This allows the programmer to perform actions that would be impossible in most programming languages such as, but not limited to, passing code as a parameter to a function, executing data, and modifying the program at runtime.

```

"Name"      = 'LISP'
"Author"    = 'John McCarthy'
"Version"   = 'Standard'
"About"     = 'LISP is an abstract language that organizes'
              | 'ALL data around "lists".'

"Start Symbol" = <s-Expression>

{Atom Char}   = {Printable} - {Whitespace} - [()"'']
{String Char} = {Printable} - ["\]

Atom    = ( {Atom Char} | '\'{Printable} )+
String = '"' ( {String Char} | '\'{Printable} )* '"'

<s-Expression> ::= <Quote> Atom
                  | <Quote> '(' <Series> ')'
                  | <Quote> '(' <s-Expression> '.' <s-Expression> ')'
                  | String

<Series> ::= <s-Expression> <Series>
          |

<Quote>  ::= ' '
          |

```

E.6. Smalltalk Programming Language

The Smalltalk programming language was one of the first languages to incorporate the concept of object-oriented programming. Developed in the Xerox labs at Palo Alto, Smalltalk was one of a series of ground-breaking ideas that defined modern computer science.

In the language's metaphor, everything is an object and every interaction between objects are known as messages. Due to a strict adherence to this concept, Smalltalk is recognized as a pure object-oriented language.

The Smalltalk grammar is amazingly simple with only three levels of operator precedence. In addition, the grammar contains no reserved words, and only a minimum of symbols. As a result, Smalltalk grammars are mainly (if not entirely) defined by programmer-defined objects and method names.

This is version IV of the grammar, which allows methods to be defined, but not yet the objects themselves.

It should also be noted that comments in Smalltalk are defined as a series of characters delimited by Double-Quotes ("). The 'Comment Start' and 'Comment End' terminals cannot be used since the start and end terminals are identical. However, since they are identical, Smalltalk comments cannot be nested and the normal 'Whitespace' terminal can be used.

```
!*  
SmallTalk IV  
  
The first object-oriented programming language.
```

"Smalltalk was developed in the Learning Research Group at Xerox's Palo Alto Research Center in the early 70s. The major ideas in Smalltalk are generally credited to Alan Kay with many roots in Simula, LISP and SketchPad. Dan Engalls wrote the first overlapping windows, opaque pop-up menus and BitBlt. Guess where Apple's OS and Microsoft Windows "found" their roots? Right, Smalltalk! Adele Goldberg and Dave Robson wrote the reference manuals for Smalltalk and were key development team members."

— Randy Best, STIC Director

"Why Smalltalk? Smalltalk uses a simplified version of English. Nouns and verbs. Object are the nouns. Verbs are messages sent from one object to the next. Easy as 1, 2, 3. It is no longer necessary to write cryptic programs. This means that almost anyone can learn to write Smalltalk programs."

— Peter William Lount

"I invented the term Object-Oriented, and I can tell you I did not have C++ in mind."

— Alan Kay (Inventor of Smalltalk)

*!

```
"Name"      = 'Smalltalk IV'
"Author"    = 'Alan Key'
"Version"   = 'IV'
"About"     = 'Smalltalk was the first object-oriented programming'
              | 'language.'
```

```
"Case Sensitive" = True
```

```
"Start Symbol" = <Program>
```

```
{Id Tail}      = {Alphanumeric} + [_]
{Integer Tail} = [ABCDEFGHJKLMNOPQRSTUVWXYZ] + {Digit}
{String Chars} = {Printable} - ['']
{Selector Char} = [!%&*+ /<=>?@\~,]
{Comment Char} = {Printable} + {Whitespace} - ["]
```

```
! The whitespace character has been modified to accept all
! double-quoted strings
```

```

Whitespace = {Whitespace}+ | ' "' {Comment Char}* ' "'

! The Keyword token is used in SmallTalk to represent a binary
! message. Essentially, this is an object defined operator

Keyword      = {Letter}{Id Tail}* ':'

Identifier    = {Letter}{Id Tail}*
BinarySelector = {Selector Char}+
IntegerLiteral = {Digit}+ | {Digit}+ 'r' {Integer Tail}+
FloatLiteral  = {Digit}+ '.' {Digit}+ ( [edq] '-'? {Digit}+ )?
ScaledLiteral = {Digit}+ ( '.' {Digit}+ )? 's' ( {Digit}+ )?
QuotedString  = ' ' ( {String Chars} | ' ' ' ' )* ' '

! ----- Grammar rules
<Program> ::= <Temporaries> <Sentences>

<Temporaries> ::= ' | ' ' | '
                | ' | ' <Identifier List> ' | '
                |

<Identifier List> ::= <Identifier List> Identifier
                    | Identifier

<Sentences> ::= <Sentence List>
               |

<Sentence List> ::= <Sentence List> <Expression> '.'
                  | <Expression> '.'

! ----- Expressions
<Expression> ::= Identifier ':' <Message Exp>
               | Identifier ':' <Primary>
               | <Message Exp>
               | <Primary>

<Primary> ::= Identifier
            | <Literal>
            | <Block>
            | '(' <Message Exp> ')'
            | '(' <Primary> ')'

<Message Exp> ::= <Unary Exp>
                | <Binary Exp>
                | <Keyword Exp>

<Unary Exp> ::= <Primary> <Unary Exp List>

```

```

<Unary Exp List> ::= <Unary Exp List> <Unary Message>
                  | <Unary Message>

<Binary Exp> ::= <Unary Exp> <Binary Exp List>
                | <Primary> <Binary Exp List>

<Binary Exp List> ::= <Binary Exp List> <Binary Message>
                    | <Binary Message>

<Keyword Exp> ::= <Binary Exp> <Keyword Message>
                 | <Primary> <Keyword Message>
                 | <Unary Exp> <Keyword Message>

! ----- Messages
<Unary Message> ::= <Unary Selector>

<Binary Message> ::= <Binary Selector> <Unary Exp>
                   | <Binary Selector> <Primary>

<Keyword Message> ::= Keyword <Binary Exp> <Keyword Message>
                    | Keyword <Primary> <Keyword Message>
                    | Keyword <Primary>
                    | Keyword <Binary Exp>
                    | Keyword <Unary Exp> <Keyword Message>
                    | Keyword <Unary Exp>

<Block> ::= '[' ':' Identifier '|' <Sentences> ']'
          | '[' <Sentences> ']'

<Binary Selector> ::= BinarySelector

<Unary Selector> ::= Identifier

! ----- Literals
<Literal> ::= <Literal Item>
           | '#' '(' ' ' ')'
           | '#' '(' <Literal List> ')'

<Literal List> ::= <Literal List> <Literal Item>

<Literal Item> ::= IntegerLiteral
                | QuotedString
                | FloatLiteral
                | ScaledLiteral

```


E.7. SQL

The SQL (Structured Query Language) programming language was developed as a uniform means of modifying and querying relational databases. By using a single abstract language to interact with the database, programs can be written that are independent of the vender and format of the database itself.

```
"Name"          = 'SQL 89'
"Version"       = '1989'
"About"         = 'This is the ANSI 89 version of SQL. Variations'
                  | 'are used by Oracle, Microsoft and most other'
                  | 'database developers'

"Start Symbol" = <Query>

{String Ch 1}   = {Printable} - ["]
{String Ch 2}   = {Printable} - ['']
{Id Ch Standard} = {Alphanumeric} + [_]
{Id Ch Extended} = {Printable} - ['['] - [']']

Comment Start = '/*'
Comment End   = '*/'
Comment Line  = '--'

StringLiteral  = '{String Ch 1}*{' | '{String Ch 2}*{'
IntegerLiteral = {Digit}+
RealLiteral    = {Digit}+'.'{Digit}+

!----- Identifiers in SQL are very complex.

Id              = ({Letter}{Id Ch Standard}* | '['{Id Ch Extended}+']')
                ('.')({Letter}{Id Ch Standard}* | '['{Id Ch Extended}+']')?

<Query>        ::= <Alter Stm>
                  | <Create Stm>
                  | <Delete Stm>
                  | <Drop Stm>
                  | <Insert Stm>
                  | <Select Stm>
                  | <Update Stm>

<Alter Stm>     ::= ALTER TABLE Id ADD COLUMN <Field Def> <Constraint
```

```

Opt>
    | ALTER TABLE Id ADD <Constraint>
    | ALTER TABLE Id DROP COLUMN Id
    | ALTER TABLE Id DROP CONSTRAINT Id

<Create Stm> ::= CREATE <Unique> INDEX IntegerLiteral ON Id '(' <Order
List> ')' <With Clause>
    | CREATE TABLE Id '(' <ID List> ')' <Constraint Opt>

<Unique> ::= UNIQUE
    |

<With Clause> ::= WITH PRIMARY
    | WITH DISALLOW NULL
    | WITH IGNORE NULL
    |

<Field Def> ::= Id <Type> NOT NULL
    | Id <Type>

<Field Def List> ::= <Field Def> ',' <Field Def List>
    | <Field Def>

<Type> ::= BIT
    | DATE
    | TIME
    | TIMESTAMP
    | DECIMAL
    | REAL
    | FLOAT
    | SMALLINT
    | INTEGER
    | INTERVAL
    | CHARACTER

<Constraint Opt> ::= <Constraint>
    |

<Constraint> ::= CONSTRAINT Id <Constraint Type>
    | CONSTRAINT Id

<Constraint Type> ::= PRIMARY KEY '(' <Id List> ')'
    | UNIQUE '(' <Id List> ')'
    | NOT NULL '(' <Id List> ')'
    | FOREIGN KEY '(' <Id List> ')' REFERENCES Id '('
<Id List> ')''

<Drop Stm> ::= DROP TABLE Id
    | DROP INDEX Id ON Id

```

```

<Insert Stm>      ::= INSERT INTO Id '(' <Id List> ')' <Select Stm>
                   | INSERT INTO Id '(' <Id List> ')' VALUES '(' <Expr
List> ')'

<Update Stm>      ::= UPDATE Id SET <Assign List> <Where Clause>

<Assign List>     ::= Id '=' <Expression> ',' <Assign List>
                   | Id '=' <Expression>

<Delete Stm>      ::= DELETE FROM Id <Where Clause>

<Select Stm>      ::= SELECT <Columns> <Into Clause> <From Clause> <Where
Clause> <Group Clause> <Having Clause> <Order Clause>

<Columns>         ::= <Restriction> '*'
                   | <Restriction> <Column List>

<Column List>     ::= <Column Source> ',' <Column List>
                   | <Column Source>

<Column Source>   ::= <Aggregate>
                   | Id

<Restriction>     ::= ALL
                   | DISTINCT
                   |

<Aggregate>       ::= Count '(' '*' ')'
                   | Count '(' <Expression> ')'
                   | Avg '(' <Expression> ')'
                   | Min '(' <Expression> ')'
                   | Max '(' <Expression> ')'
                   | StDev '(' <Expression> ')'
                   | StDevP '(' <Expression> ')'
                   | Sum '(' <Expression> ')'
                   | Var '(' <Expression> ')'
                   | VarP '(' <Expression> ')'

<Into Clause>     ::= INTO Id
                   |

<From Clause>     ::= FROM <Id List> <Join Chain>

<Join Chain>      ::= <Join> <Join Chain>
                   |

<Join>            ::= INNER JOIN <Id List> ON Id '=' Id
                   | LEFT JOIN <Id List> ON Id '=' Id
                   | RIGHT JOIN <Id List> ON Id '=' Id

```

	JOIN <Id List> ON Id '=' Id
<Where Clause>	::= WHERE <Expression>
<Group Clause>	::= GROUP BY <Id List>
<Order Clause>	::= ORDER BY <Order List>
<Order List>	::= ID <Order Type> ',' <Order List> ID <Order Type>
<Order Type>	::= ASC DESC
<Having Clause>	::= HAVING <Expression>
!-----	
Expressions	
<Expression>	::= <And Exp> OR <Expression> <And Exp>
<And Exp>	::= <Not Exp> AND <And Exp> <Not Exp>
<Not Exp>	::= NOT <Pred Exp> <Pred Exp>
<Pred Exp>	::= <Add Exp> BETWEEN <Add Exp> AND <Add Exp> <Add Exp> NOT BETWEEN <Add Exp> AND <Add Exp> <Value> IS NOT NULL <Value> IS NULL <Add Exp> LIKE StringLiteral <Add Exp> IN <Tuple> <Add Exp> '=' <Add Exp> <Add Exp> '< >' <Add Exp> <Add Exp> '!=' <Add Exp> <Add Exp> '>' <Add Exp> <Add Exp> '>=' <Add Exp> <Add Exp> '<' <Add Exp> <Add Exp> '<=' <Add Exp> <Add Exp>
<Add Exp>	::= <Add Exp> '+' <Mult Exp> <Add Exp> '-' <Mult Exp>

		<Mult Exp>
<Mult Exp>	::=	<Mult Exp> '*' <Negate Exp>
		<Mult Exp> '/' <Negate Exp>
		<Negate Exp>
<Negate Exp>	::=	'-' <Value>
		<Value>
<Value>	::=	<Tuple>
		ID
		IntegerLiteral
		RealLiteral
		StringLiteral
<Tuple>	::=	'(' <Select Stm> ')'
		'(' <Expr List> ')'
<Expr List>	::=	<Expression> ',' <Expr List>
		<Expression>
<Id List>	::=	Id ',' <Id List>
		Id

E.8. Visual Basic .NET

Visual Basic .NET is the latest version in the long evolution of the BASIC programming language. The Visual Basic .NET programming language is far more "clean" and orthogonal than its predecessors. Although some of the old constructs exist, the language is far easier to read and write.

Major syntax changes include, but are not limited to:

1. The primitive file access of VB6 was replaced by a class library. As a result, special statements such as `"Open "test" For Input As #1"` no longer exist.
2. Class module files were replaced by `'Class ... End Class'` declarations
3. Module files were replaced by `'Module ... End Module'` declarations
4. Structured error handling was added with C++ style `Try ... Catch` statements. The old unstructured approach is still, unfortunately, available.

Unfortunately, the designers of Visual Basic .NET did not remove the datatype postfix characters on identifiers. In the original BASIC, variables types were determined by `$` for strings and `%` for integers. QuickBasic expanded the postfix notation to include other symbols for long integers, singles, etc... Part of the ID terminal definition was commented out to prevent the old format. You can allow the postfix characters if you like.

This grammar also does not contain the compiler directives.

```

"Name"      = 'Visual Basic .NET'
"Author"    = 'John G. Kemeny and Thomas E. Kurtz'
"Version"   = '.NET'
"About"     = 'Visual Basic .NET is the latest version in the'
              | 'long evolution of the BASIC programming language.'

"Case Sensitive" = False
"Start Symbol"  = <Program>

! ----- Sets

{String Chars} = {Printable} - ["]
{Date Chars}  = {Printable} - [#]
{ID Name Chars} = {Printable} - ['[ '' ]']
{Hex Digit}    = {Digit} + [abcdef]
{Oct Digit}    = [01234567]

{WS}           = {Whitespace} - {CR} - {LF}
{Id Tail}      = {Alphanumeric} + [_]

! ----- Terminals

NewLine        = {CR}{LF} | {CR} | ':'
Whitespace     = {WS}+ | '_' {WS}* {CR} {LF}?

Comment Line   = ' ' | Rem                      !Fixed by Vladimir Morozov

LABEL          = {Letter}{ID Tail}*':'

!Fixed by Vladimir Morozov

ID              = {Letter}{ID Tail}*              ! [%&@!#$]?    !Archaic
postfix chars   | '[' {ID Name Chars}* ']'

QualifiedID     = (({Letter}{ID Tail}* | '[' {ID Name Chars}* '']) (
'.'({Letter}{ID Tail}* | '[' {ID Name Chars}* '']) )+

MemberID        = '.' {Letter}{ID Tail}*
                  | '[' {ID Name Chars}* ']'

!Fixed by Vladimir Morozov
StringLiteral   = '"' ( {String Chars} | '""' )* '"'

CharLiteral     = '"' {String Chars}* '"C'
IntLiteral      = [-]? {digit}+ [FRDSIL]?

!Fixed by Vladimir Morozov
RealLiteral     = [-]? {digit}* '.' {digit}+ ( 'E' [+]? {Digit}+ )?

```



```

[FR]?
    | [-]? {digit}+ 'E' [+]? {Digit}+ [FR]?

DateLiteral    = '#' {Date chars} '#'
HexLiteral     = '&H' {Hex Digit}+ [SIL]?
OctLiteral     = '&O' {Oct Digit}+ [SIL]?

! ----- Rules

<Program>      ::= <NameSpace Item> <Program>
                | <Imports> <Program>
                | <Option Decl> <Program>
                |

! ----- (Shared attributes)

<NL>           ::= NewLine <NL>
                | NewLine

<Modifiers>    ::= <Modifier> <Modifiers>
                |

<Modifier>    ::= Shadows
                | Shared
                | MustInherit
                | NotInheritable

                | Overridable
                | NotOverridable
                | MustOverride
                | Overrides
                | Overloads

                | Default
                | ReadOnly
                | WriteOnly

                | <Access>

<Access Opt>   ::= <Access>
                |

<Access>      ::= Public
                | Private
                | Friend
                | Protected

<Var Member>   ::= <Attributes> <Access> <Var Decl> <NL>
                | <Attributes> <Access Opt> Const <Var Decl> <NL>

```

```

| <Attributes> <Access Opt> Static <Var Decl> <NL>

<Implements> ::= Implements <ID List>

<ID List> ::= <Identifier> ',' <ID List>
| <Identifier>

<Option Decl> ::= Option <IDs> <NL>

<IDs> ::= ID <IDs>
| ID

<Type> ::= As <Attributes> <Identifier>
|

<Compare Op> ::= '=' | '<' | '<' | '>' | '>=' | '<='

! ----- Namespace

<Namespace> ::= Namespace ID <NL> <Namespace Items> End Namespace
<NL>

<Namespace Items> ::= <Namespace Item> <Namespace Items>
|

<Namespace Item> ::= <Class>
| <Declare>
| <Delegate>
| <Enumeration>
| <Interface>
| <Structure>
| <Module>
| <Namespace>

! ----- Attributes

<Attributes> ::= '<' <Attribute List> '>'
|

<Attribute List> ::= <Attribute> ',' <Attribute List>
| <Attribute>

<Attribute> ::= <Attribute Mod> ID <Argument List Opt>

<Attribute Mod> ::= Assembly
| Module
|

! ----- Delegates

<Delegate> ::= <Attributes> <Modifiers> Delegate <Method>
| <Attributes> <Modifiers> Delegate <Declare>

```

```

! ----- Imports
<Imports> ::= Imports <Identifier> <NL>
           | Imports ID '=' <Identifier> <NL>

! ----- Events
<Event Member> ::= <Attributes> <Modifiers> Event ID <Parameters Or
Type> <Implements Opt> <NL>

<Parameters Or Type> ::= <Param List>
                       | As <Identifier>

<Implements Opt> ::= <Implements>
                    |

! ----- Class
<Class>      ::= <Attributes> <Modifiers> Class ID <NL> <Class Items>
End Class <NL>

<Class Items> ::= <Class Item> <Class Items>
                |

<Class Item>  ::= <Declare>
                | <Method>
                | <Property>
                | <Var Member>
                | <Enumeration>
                | <Inherits>

<Inherits> ::= Inherits <Identifier> <NL>

! ----- Structures
<Structure>   ::= <Attributes> <Modifiers> Structure ID <NL>
<Structure List> End Structure <NL>

<Structure List> ::= <Structure Item> <Structure List>
                    |

<Structure Item> ::= <Implements>
                    | <Enumeration>
                    | <Structure>
                    | <Class>
                    | <Delegate>
                    | <Var Member>
                    | <Event Member>
                    | <Declare>
                    | <Method>
                    | <Property>

```

```

! ----- Module
<Module>      ::= <Attributes> <Modifiers> Module ID <NL> <Module
Items> End Module <NL>

<Module Items> ::= <Module Item> <Module Items>
                |

<Module Item>  ::= <Declare>
                | <Method>
                | <Property>
                | <Var Member>
                | <Enumeration>
                | <Option Decl>

! ----- Interface
<Interface> ::= <Attributes> <Modifiers> Interface ID <NL> <Interface
Items> End Interface <NL>

<Interface Items> ::= <Interface Item> <Interface Items>
                    |

<Interface Item>  ::= <Implements>
                    | <Event Member>
                    | <Enum Member>
                    | <Method Member>
                    | <Property Member>

<Enum Member>     ::= <Attributes> <Modifiers> Enum ID <NL>

<Method Member>   ::= <Attributes> <Modifiers> Sub <Sub ID> <Param
List> <Handles Or Implements> <NL>
                    | <Attributes> <Modifiers> Function ID <Param
List> <Type> <Handles Or Implements> <NL>

<Property Member> ::= <Attributes> <Modifiers> Property ID <Param
List> <Type> <Handles Or Implements> <NL>

! ----- Parameters
<Param List Opt> ::= <Param List>
                    |

<Param List>     ::= '(' <Param Items> ')'
                    | '(' ' ' ')'

<Param Items>    ::= <Param Item> ',' <Param Items>
                    | <Param Item>

<Param Item>     ::= <Param Passing> ID <Type>

```

```

<Param Passing> ::= ByVal
                  | ByRef
                  | Optional
                  | ParamArray
                  |

! ----- Arguments
<Argument List Opt> ::= <Argument List>
                    |

<Argument List> ::= '(' <Argument Items> ')'

<Argument Items> ::= <Argument> ',' <Argument Items>
                  | <Argument>

<Argument> ::= <Expression>
              | Id ':' <Expression>
              | !NULL

! ----- Declares
<Declare> ::= <Attributes> <Modifiers> Declare <Charset> Sub ID
Lib StringLiteral <Alias> <Param List Opt> <NL>
              | <Attributes> <Modifiers> Declare <Charset> Function ID
Lib StringLiteral <Alias> <Param List Opt> <Type> <NL>

<Charset> ::= Ansi | Unicode | Auto | !Null

<Alias> ::= Alias StringLiteral
        |

! ----- Methods
<Method> ::= <Attributes> <Modifiers> Sub <Sub ID> <Param List>
<Handles Or Implements> <NL> <Statements> End Sub <NL>
        | <Attributes> <Modifiers> Function ID <Param List> <Type>
<Handles Or Implements> <NL> <Statements> End Function <NL>

<Sub ID> ::= ID
          | New !Class creation

<Handles Or Implements> ::= <Implements>
                          | <Handles>
                          |

<Handles> ::= Handles <ID List>

! ----- Properties

```

```

<Property> ::= <Attributes> <Modifiers> Property ID <Param List>
<Type> <NL> <Property Items> End Property <NL>

<Property Items> ::= <Property Item> <Property Items>
|

<Property Item> ::= Get <NL> <Statements> End Get <NL>
| Set <Param List> <NL> <Statements> End Set <NL>

! ----- Enumerations
<Enumeration> ::= <Attributes> <Modifiers> Enum ID <NL> <Enum List>
End Enum <NL>

<Enum List> ::= <Enum Item> <Enum List>
|

<Enum Item> ::= Id '=' IntLiteral <NL>
| Id <NL>

! ----- Variables

<Var Decl> ::= <Var Decl Item> ',' <Var Decl>
| <Var Decl Item>

<Var Decl Item> ::= <Var Decl ID> As <Identifier> <Argument List Opt>
| <Var Decl ID> As <Identifier> '=' <Expression>
!Initialize
| <Var Decl ID> As New <Identifier> <Argument List
Opt>
| <Var Decl ID>
| <Var Decl ID> '=' <Expression>
!Initialize

<Var Decl ID> ::= ID <Argument List Opt>

! ----- Normal Statements

<Statements> ::= <Statement> <Statements>
|

<Statement> ::= <Loop Stm>
| <For Stm>
| <If Stm>
| <Select Stm>
| <SyncLock Stm>
| <Try Stm>
| <With Stm>
| <Option Decl>
| <Local Decl>

```

	<Non-Block Stm> <NL>	!Note the <NL>. A non-
block statement	can be a full statement	
	LABEL <NL>	
<Non-Block Stm>	::= Call <Variable>	
	ReDim <Var Decl>	
	ReDim Preserve <Var Decl>	
	Erase ID	
	Throw <Value>	
	RaiseEvent <Identifier> <Argument List Opt>	
	AddHandler <Expression> ',' <Expression>	
	RemoveHandler <Expression> ',' <Expression>	
	Exit Do	
	Exit For	
	Exit Function	
	Exit Property	
	Exit Select	
	Exit Sub	
	Exit Try	
	Exit While	
	GoTo ID	!Argh - they still have
this	Return <Value>	
	Error <Value>	!Raise an error
by number	On Error GoTo IntLiteral	!-1 or 0
	On Error GoTo Id	
	On Error Resume Next	
	Resume ID	
	Resume Next	
	<Variable> <Assign Op> <Expression>	
	<Variable>	
	<Method Call>	
<Assign Op>	::= '=' '^=' '*=' '/=' '\='	
	'+=' '-=' '&=' '<=>' '>=>'	
!	-----Local declarations	
<Local Decl>	::= Dim <Var Decl> <NL>	
	Const <Var Decl> <NL>	
	Static <Var Decl> <NL>	
!	----- Do Statement	
<Loop Stm>	::= Do <Test Type> <Expression> <NL> <Statements> Loop <NL>	
	Do <NL> <Statements> Loop <Test Type> <Expression> <NL>	

```

| While <Expression> <NL> <Statements> End While <NL>

<Test Type> ::= While
| Until

! ----- For Statement

<For Stm> ::= For <Identifier> '=' <Expression> To <Expression> <Step
Opt> <NL> <Statements> Next <NL>
| For Each <Variable> In <Variable> <NL> <Statements> Next
<NL>

<Step Opt> ::= Step <Expression>
|

! ----- If Statement

<If Stm> ::= If <Expression> <Then Opt> <NL> <Statements> <If
Blocks> End If <NL>
| If <Expression> Then <Non-Block Stm> <NL>
| If <Expression> Then <Non-Block Stm> Else <Non-Block
Stm> <NL>

<Then Opt> ::= Then !!The reserved word 'Then' is optional for
Block-If statements
|

<If Blocks> ::= ElseIf <Expression> <Then Opt> <NL> <Statements> <If
Blocks>
| Else <NL> <Statements>
|

! ----- Select Statement

<Select Stm> ::= Select <Case Opt> <Expression> <NL> <Select Blocks>
End Select <NL>

<Case Opt> ::= Case !!The "Case" after
Select is optional in VB.NET
|

<Select Blocks> ::= Case <Case Clauses> <NL> <Statements> <Select
Blocks>
| Case Else <NL> <Statements>
|

<Case Clauses> ::= <Case Clause> ',' <Case Clauses>
| <Case Clause>

```



```

<Case Clause> ::= <Is Opt> <Compare Op> <Expression>
                |
                | <Expression> To <Expression>

<Is Opt> ::= Is
            | !Null

! ----- SyncLock Statement

<SyncLock Stm> ::= SyncLock <NL> <Statements> End SyncLock <NL>

! ----- Try Statement

<Try Stm>      ::= Try <NL> <Statements> <Catch Blocks> End Try <NL>

<Catch Blocks> ::= <Catch Block> <Catch Blocks>
                | <Catch Block>

<Catch Block>  ::= Catch <Identifier> As ID <NL> <Statements>
                | Catch <NL> <Statements>

! ----- With Statement

<With Stm> ::= With <Value> <NL> <Statements> End With <NL>

! ----- Expressions

<Expression> ::= <And Exp> Or      <Expression>
                | <And Exp> OrElse <Expression>
                | <And Exp> XOr   <Expression>
                | <And Exp>

<And Exp>    ::= <Not Exp> And      <And Exp>
                | <Not Exp> AndAlso <And Exp>
                | <Not Exp>

<Not Exp>    ::= NOT <Compare Exp>
                | <Compare Exp>

<Compare Exp> ::= <Shift Exp> <Compare Op> <Compare Exp>           !e.g. x
< y
                | TypeOf <Add Exp> Is <Object>
                | <Shift Exp> Is <Object>
                | <Shift Exp> Like <Value>
                | <Shift Exp>

<Shift Exp>  ::= <Concat Exp> '<<' <Shift Exp>
                | <Concat Exp> '>>' <Shift Exp>
                | <Concat Exp>

<Concat Exp> ::= <Add Exp> '&' <Concat Exp>

```

		<Add Exp>	
<Add Exp>	::=	<Modulus Exp> '+' <Add Exp> <Modulus Exp> '-' <Add Exp> <Modulus Exp>	
<Modulus Exp>	::=	<Int Div Exp> Mod <Modulus Exp> <Int Div Exp>	
<Int Div Exp>	::=	<Mult Exp> '\' <Int Div Exp> <Mult Exp>	
<Mult Exp>	::=	<Negate Exp> '*' <Mult Exp> <Negate Exp> '/' <Mult Exp> <Negate Exp>	
<Negate Exp>	::=	'-' <Power Exp> <Power Exp>	
<Power Exp>	::=	<Power Exp> '^' <Value> <Value>	
<Value>	::=	'(' <Expression> ')' New <Identifier> <Argument List Opt> IntLiteral HexLiteral OctLiteral StringLiteral CharLiteral RealLiteral DateLiteral True False Me MyClass MyBase Nothing <Variable>	
<Object>	::=	<Identifier> !Object identifiers Me MyClass MyBase Nothing	
<Variable>	::=	<Identifier> <Argument List Opt> <Method Calls>	
<Method Calls>	::=	<Method Call> <Method Calls> 	
<Method Call>	::=	MemberID <Argument List Opt>	

<code><Identifier> ::= ID QualifiedID</code>	<code>!Any type of identifier</code>
--	--------------------------------------

E.9. XML

XML (eXtenable Markup Language), like the commonly used HTML format, is a descendant of SGML (Standardized General Markup Language). However, XML is not a descendant of HTML and structural differences exist between the two formats.

The XML syntax essentially allows a tree to be defined using normal ASCII characters. The format used to start and end logical objects is identical to the format used by HTML with three major exceptions:

1. All starting tags must have an ending tag
2. Single tags (objects without sub-objects) end in />
3. All attributes must be delimited by double quotes

```
"Name"           = 'XML'
"Version"        = '1.0'
"Author"         = 'W3C Recommendation 10-February-1998'
"About"          = 'eXtenable Markup Language'

"Start Symbol"   = <XML Doc>

{Symbol Chars}   = {Printable} - {Letter} - [<>'\"=] - {Whitespace}
{Attribute Chars} = {Printable} - ["]

Comment Start    = '<!--'
Comment End      = '-->'

AttributeValue   = '"' {Attribute Chars}* '"'

Name             = {Letter}{Alphanumeric}*
Symbol           = {Symbol Chars}

CharName         = '&' {Letter}+ ';'
CharNumber       = '&#' {Digit}+ ';'

```

```

! -----

<XML Doc> ::= '<?' Name <Attributes> '?>' <Object>

<Objects>  ::= <Object> <Objects>
              | <Object>

<Object>   ::= <Start Tag> <Content> <End Tag>
              | <Unary Tag>

<Start Tag> ::= '<' Name <Attributes> '>'
<End Tag>   ::= '</' Name '>'

<Unary Tag> ::= '<' Name <Attributes> '/>'

<Content>   ::= <Objects>
              | <Text>
              |

<Attributes> ::= <Attribute> <Attributes>
                |

<Attribute> ::= Name '=' AttributeValue

! -----

<Text> ::= <Word> <Text>
        | <Word>

<Word> ::= Name
        | Symbol
        | '='
        | CharName
        | CharNumber

```

Appendix F. Engine Test

F.1. Test Program

The following is a very simple ANSI C library that was used in Computer Science 251: Compiler Design. This source code was used to complete one of the course projects - the creation of an interpreter using the YACC Compiler-compiler. Far more complex programs were tested, but the parse trees are too large to put into this document.

```
/*
CSc 251
Compiler Design

This is a VERY simple library that contains a few functions
designed to help with common tasks in C.
*/

typedef int bool;
const int FALSE = 0;
const int TRUE  = ! 0;

char *newString(char *str) {

    //This function creates a new dynamic copy of str and
    //returns its pointer
    void *p;

    p = malloc(strlen(str)+1);

    strcpy(p, str);
    return p;
}

//=====
```

```

char *strcat(char *str, char ch) {

    char newstring[1];

    newstring[0]=ch;
    newstring[1]='\0';
    strcat(str, newstring);
    return str;
}

```

F.2. Parse Tree

The following is the parse tree generated by the Engine for the program listed above. The "Trim Reductions" attribute was set to true. As a result, the reductions containing a single nonterminal were "trimmed" from the parse tree. For more information, please see Chapter 5.

```

+<Decls> ::= <Decl> <Decls>
|
| +<Typedef Decl> ::= typedef <Type> Id ';'
| |
| | +<typedef>
| | |
| | | +<Type> ::= <Base> <Pointers>
| | | |
| | | | +<Base> ::= <Sign> <Scalar>
| | | | |
| | | | | +<Sign> ::=
| | | | | +<Scalar> ::= int
| | | | | |
| | | | | | +<int>
| | | | +<Pointers> ::=
| | | +<bool>
| | +<+>
|
| +<Decls> ::= <Decl> <Decls>
| |
| | +<Var Decl> ::= <Mod> <Type> <Var> <Var List> ';'
| | |
| | | +<Mod> ::= const
| | | |
| | | | +<const>
| | | +<Type> ::= <Base> <Pointers>
| | | |
| | | | +<Base> ::= <Sign> <Scalar>
| | | | |
| | | | | +<Sign> ::=
| | | | | +<Scalar> ::= int
| | | | | |
| | | | | | +<int>
| | | | +<Pointers> ::=
| | | +<Var> ::= Id <Array> '=' <Op If>
| | | |
| | | | +<FALSE>
| | | | +<Array> ::=

```

```

|--+--
|--+--<Value> ::= OctLiteral
|--| +-0
|--+--<Var List> ::=
|--+--;
+--<Decls> ::= <Decl> <Decls>
+--<Var Decl> ::= <Mod> <Type> <Var> <Var List> ';'
+--<Mod> ::= const
+--| +-const
+--<Type> ::= <Base> <Pointers>
+--<Base> ::= <Sign> <Scalar>
+--| +-<Sign> ::=
+--| +-<Scalar> ::= int
+--| +-int
+--| +-<Pointers> ::=
+--<Var> ::= Id <Array> '=' <Op If>
+--| +-TRUE
+--| +-<Array> ::=
+--| +-=
+--| +-<Op Unary> ::= '!' <Op Unary>
+--| +-!
+--| +-<Value> ::= OctLiteral
+--| +-0
+--<Var List> ::=
+--+--;
+--<Decls> ::= <Decl> <Decls>
+--<Func Decl> ::= <Func ID> '(' <Params> ')' <Block>
+--<Func ID> ::= <Type> Id
+--<Type> ::= <Base> <Pointers>
+--<Base> ::= <Sign> <Scalar>
+--<Sign> ::=
+--<Scalar> ::= char
+--| +-char
+--<Pointers> ::= '*' <Pointers>
+--| +-*
+--<Pointers> ::=
+--| +-newString
+--(
+--<Param> ::= <Type> Id
+--<Type> ::= <Base> <Pointers>
+--<Base> ::= <Sign> <Scalar>
+--<Sign> ::=
+--<Scalar> ::= char
+--| +-char
+--<Pointers> ::= '*' <Pointers>
+--| +-*
+--<Pointers> ::=
+--| +-str
+--)
+--<Block> ::= '{' <Stm List> '}'
+--| +-{

```



```

+-<Stm List> ::= <Stm> <Stm List>
+-<Var Decl> ::= <Type> <Var> <Var List> ';'
|
+-<Type> ::= <Base> <Pointers>
|
| +-<Base> ::= <Sign> <Scalar>
| | +-<Sign> ::=
| | +-<Scalar> ::= void
| | +-void
| +-<Pointers> ::= '*' <Pointers>
| +-*
| +-<Pointers> ::=
+-<Var> ::= Id <Array>
| +-p
| +-<Array> ::=
+-<Var List> ::=
+-;

+-<Stm List> ::= <Stm> <Stm List>
+-<Normal Stm> ::= <Expr> ';'
|
| +-<Op Assign> ::= <Op If> '=' <Op Assign>
| | +-<Value> ::= Id
| | | +-p
| | +-=
| | +-<Value> ::= Id '(' <Expr> ')'
| | +-malloc
| | +- (
| | +-<Op Add> ::= <Op Add> '+' <Op Mult>
| | | +-<Value> ::= Id '(' <Expr> ')'
| | | | +-strlen
| | | | +- (
| | | | +-<Value> ::= Id
| | | | | +-str
| | | +- )
| | +-+
| | +-<Value> ::= DecLiteral
| | +-1
| +- )
| +-;

+-<Stm List> ::= <Stm> <Stm List>
+-<Normal Stm> ::= <Expr> ';'
|
| +-<Value> ::= Id '(' <Expr> ')'
| | +-strcpy
| | +- (
| | +-<Expr> ::= <Expr> ',' <Op Assign>
| | | +-<Value> ::= Id
| | | | +-p
| | | +- ,
| | | +-<Value> ::= Id
| | | | +-str
| | +- )
| +-;

+-<Stm List> ::= <Stm> <Stm List>
+-<Normal Stm> ::= return <Expr> ';'

```

```

| | | | | +-return
| | | | | +-<Value> ::= Id
| | | | | | +-p
| | | | | +-;
| | | | | +-<Stm List> ::=
| | +-}
+-<Decls> ::= <Decl> <Decls>
| +-<Func Decl> ::= <Func ID> '(' <Params> ')' <Block>
| | +-<Func ID> ::= <Type> Id
| | | +-<Type> ::= <Base> <Pointers>
| | | | +-<Base> ::= <Sign> <Scalar>
| | | | | +-<Sign> ::=
| | | | | +-<Scalar> ::= char
| | | | | | +-char
| | | | +-<Pointers> ::= '*' <Pointers>
| | | | | +-*
| | | | | +-<Pointers> ::=
| | +-strcatch
+-(
+-<Params> ::= <Param> ',' <Params>
| +-<Param> ::= <Type> Id
| | +-<Type> ::= <Base> <Pointers>
| | | +-<Base> ::= <Sign> <Scalar>
| | | | +-<Sign> ::=
| | | | +-<Scalar> ::= char
| | | | | +-char
| | | +-<Pointers> ::= '*' <Pointers>
| | | | +-*
| | | | +-<Pointers> ::=
| +-str
+-,
+-<Param> ::= <Type> Id
| +-<Type> ::= <Base> <Pointers>
| | +-<Base> ::= <Sign> <Scalar>
| | | +-<Sign> ::=
| | | +-<Scalar> ::= char
| | | | +-char
| | +-<Pointers> ::=
| +-ch
+-)
+-<Block> ::= '{' <Stm List> '}'
| +-{
| +-<Stm List> ::= <Stm> <Stm List>
| | +-<Var Decl> ::= <Type> <Var> <Var List> ';'
| | | +-<Type> ::= <Base> <Pointers>
| | | | +-<Base> ::= <Sign> <Scalar>
| | | | | +-<Sign> ::=
| | | | | +-<Scalar> ::= char
| | | | | | +-char
| | | +-<Pointers> ::=
| +-<Var> ::= Id <Array>

```

```

+-newstring
+-<Array> ::= '[' <Expr> ']'
| +-[
|   +-<Value> ::= DecLiteral
|   | +-1
|   +-]
+-<Var List> ::=
+-;
+-<Stm List> ::= <Stm> <Stm List>
+-<Normal Stm> ::= <Expr> ';'
| +-<Op Assign> ::= <Op If> '=' <Op Assign>
|   +-<Op Pointer> ::= <Op Pointer> '[' <Expr> ']'
|   | +-<Value> ::= Id
|   | | +-newstring
|   +-[
|     +-<Value> ::= OctLiteral
|     | +-0
|     +-]
|   +=
|   +-<Value> ::= Id
|   | +-ch
+-;
+-<Stm List> ::= <Stm> <Stm List>
+-<Normal Stm> ::= <Expr> ';'
| +-<Op Assign> ::= <Op If> '=' <Op Assign>
|   +-<Op Pointer> ::= <Op Pointer> '[' <Expr> ']'
|   | +-<Value> ::= Id
|   | | +-newstring
|   +-[
|     +-<Value> ::= DecLiteral
|     | +-1
|     +-]
|   +=
|   +-<Value> ::= CharLiteral
|   | +-'\0'
+-;
+-<Stm List> ::= <Stm> <Stm List>
+-<Normal Stm> ::= <Expr> ';'
| +-<Value> ::= Id '(' <Expr> ')'
|   +-strcat
|   +- (
|     +-<Expr> ::= <Expr> ',' <Op Assign>
|     | +-<Value> ::= Id
|     | | +-str
|     +- ,
|     +-<Value> ::= Id
|     | +-newstring
|   +- )
+-;
+-<Stm List> ::= <Stm> <Stm List>
+-<Normal Stm> ::= return <Expr> ';'

```

```
| | | | | | | | | | | | | +-return  
| | | | | | | | | | | | | +-<Value> ::= Id  
| | | | | | | | | | | | | | +-str  
| | | | | | | | | | | | | +-;  
| | | | | | | | | | | | +-<Stm List> ::=  
| | | | | | | | | | | | +-}  
| | | | | | | | | | | +-<Decls> ::=
```


Appendix G. Case Mapping

The following chart contains the case mapping table used by the Builder. This information reflects the case mapping requirements found on the Unicode Consortium Website. When a grammar is defined as case insensitive (the "Case Sensitive" parameter is set to false), the Builder adds the lowercase and uppercase versions of each character used.

Table G-1. Case Mapping

Uppercase		Lowercase		Name
Dec	Hex	Dec	Hex	
#65	&41	#97	&61	Latin A
#66	&42	#98	&62	Latin B
#67	&43	#99	&63	Latin C
#68	&44	#100	&64	Latin D
#69	&45	#101	&65	Latin E
#70	&46	#102	&66	Latin F
#71	&47	#103	&67	Latin G
#72	&48	#104	&68	Latin H
#73	&49	#105	&69	Latin I
#74	&4A	#106	&6A	Latin J
#75	&4B	#107	&6B	Latin K
#76	&4C	#108	&6C	Latin L
#77	&4D	#109	&6D	Latin M
#78	&4E	#110	&6E	Latin N
#79	&4F	#111	&6F	Latin O
#80	&50	#112	&70	Latin P
#81	&51	#113	&71	Latin Q
#82	&52	#114	&72	Latin R
#83	&53	#115	&73	Latin S
#84	&54	#116	&74	Latin T
#85	&55	#117	&75	Latin U
#86	&56	#118	&76	Latin V
#87	&57	#119	&77	Latin W
#88	&58	#120	&78	Latin X
#89	&59	#121	&79	Latin Y

Uppercase	
#90	&5A
#181	&B5
#192	&C0
#193	&C1
#194	&C2
#195	&C3
#196	&C4
#197	&C5
#198	&C6
#199	&C7
#200	&C8
#201	&C9
#202	&CA
#203	&CB
#204	&CC
#205	&CD
#206	&CE
#207	&CF
#208	&D0
#209	&D1
#210	&D2
#211	&D3
#212	&D4
#213	&D5
#214	&D6
#216	&D8
#217	&D9
#218	&DA
#219	&DB
#220	&DC
#221	&DD
#222	&DE
#256	&100
#258	&102
#260	&104
#262	&106
#264	&108
#266	&10A
#268	&10C
#270	&10E
#272	&110
#274	&112
#276	&114
#278	&116
#280	&118
#282	&11A
#284	&11C
#286	&11E
#288	&120

Lowercase	
#122	&7A
#956	&3BC
#224	&E0
#225	&E1
#226	&E2
#227	&E3
#228	&E4
#229	&E5
#230	&E6
#231	&E7
#232	&E8
#233	&E9
#234	&EA
#235	&EB
#236	&EC
#237	&ED
#238	&EE
#239	&EF
#240	&F0
#241	&F1
#242	&F2
#243	&F3
#244	&F4
#245	&F5
#246	&F6
#248	&F8
#249	&F9
#250	&FA
#251	&FB
#252	&FC
#253	&FD
#254	&FE
#257	&101
#259	&103
#261	&105
#263	&107
#265	&109
#267	&10B
#269	&10D
#271	&10F
#273	&111
#275	&113
#277	&115
#279	&117
#281	&119
#283	&11B
#285	&11D
#287	&11F
#289	&121

Name
Latin Z
Micro Sign
Latin A With Grave
Latin A With Acute
Latin A With Circumflex
Latin A With Tilde
Latin A With Diaeresis
Latin A With Ring Above
Latin Ae
Latin C With Cedilla
Latin E With Grave
Latin E With Acute
Latin E With Circumflex
Latin E With Diaeresis
Latin I With Grave
Latin I With Acute
Latin I With Circumflex
Latin I With Diaeresis
Latin Eth
Latin N With Tilde
Latin O With Grave
Latin O With Acute
Latin O With Circumflex
Latin O With Tilde
Latin O With Diaeresis
Latin O With Stroke
Latin U With Grave
Latin U With Acute
Latin U With Circumflex
Latin U With Diaeresis
Latin Y With Acute
Latin Thorn
Latin A With Macron
Latin A With Breve
Latin A With Ogonek
Latin C With Acute
Latin C With Circumflex
Latin C With Dot Above
Latin C With Caron
Latin D With Caron
Latin D With Stroke
Latin E With Macron
Latin E With Breve
Latin E With Dot Above
Latin E With Ogonek
Latin E With Caron
Latin G With Circumflex
Latin G With Breve
Latin G With Dot Above

Uppercase	
#290	&122
#292	&124
#294	&126
#296	&128
#298	&12A
#300	&12C
#302	&12E
#306	&132
#308	&134
#310	&136
#313	&139
#315	&13B
#317	&13D
#319	&13F
#321	&141
#323	&143
#325	&145
#327	&147
#330	&14A
#332	&14C
#334	&14E
#336	&150
#338	&152
#340	&154
#342	&156
#344	&158
#346	&15A
#348	&15C
#350	&15E
#352	&160
#354	&162
#356	&164
#358	&166
#360	&168
#362	&16A
#364	&16C
#366	&16E
#368	&170
#370	&172
#372	&174
#374	&176
#376	&178
#377	&179
#379	&17B
#381	&17D
#383	&17F
#385	&181
#386	&182
#388	&184

Lowercase	
#291	&123
#293	&125
#295	&127
#297	&129
#299	&12B
#301	&12D
#303	&12F
#307	&133
#309	&135
#311	&137
#314	&13A
#316	&13C
#318	&13E
#320	&140
#322	&142
#324	&144
#326	&146
#328	&148
#331	&14B
#333	&14D
#335	&14F
#337	&151
#339	&153
#341	&155
#343	&157
#345	&159
#347	&15B
#349	&15D
#351	&15F
#353	&161
#355	&163
#357	&165
#359	&167
#361	&169
#363	&16B
#365	&16D
#367	&16F
#369	&171
#371	&173
#373	&175
#375	&177
#255	&FF
#378	&17A
#380	&17C
#382	&17E
#115	&73
#595	&253
#387	&183
#389	&185

Name
Latin G With Cedilla
Latin H With Circumflex
Latin H With Stroke
Latin I With Tilde
Latin I With Macron
Latin I With Breve
Latin I With Ogonek
Latin Capital Ligature Ij
Latin J With Circumflex
Latin K With Cedilla
Latin L With Acute
Latin L With Cedilla
Latin L With Caron
Latin L With Middle Dot
Latin L With Stroke
Latin N With Acute
Latin N With Cedilla
Latin N With Caron
Latin Eng
Latin O With Macron
Latin O With Breve
Latin O With Double Acute
Latin Capital Ligature Oe
Latin R With Acute
Latin R With Cedilla
Latin R With Caron
Latin S With Acute
Latin S With Circumflex
Latin S With Cedilla
Latin S With Caron
Latin T With Cedilla
Latin T With Caron
Latin T With Stroke
Latin U With Tilde
Latin U With Macron
Latin U With Breve
Latin U With Ring Above
Latin U With Double Acute
Latin U With Ogonek
Latin W With Circumflex
Latin Y With Circumflex
Latin Y With Diaeresis
Latin Z With Acute
Latin Z With Dot Above
Latin Z With Caron
Latin Small Letter Long S
Latin B With Hook
Latin B With Topbar
Latin Tone Six

Uppercase	
#390	&186
#391	&187
#393	&189
#394	&18A
#395	&18B
#398	&18E
#399	&18F
#400	&190
#401	&191
#403	&193
#404	&194
#406	&196
#407	&197
#408	&198
#412	&19C
#413	&19D
#415	&19F
#416	&1A0
#418	&1A2
#420	&1A4
#422	&1A6
#423	&1A7
#425	&1A9
#428	&1AC
#430	&1AE
#431	&1AF
#433	&1B1
#434	&1B2
#435	&1B3
#437	&1B5
#439	&1B7
#440	&1B8
#444	&1BC
#452	&1C4
#453	&1C5
#455	&1C7
#456	&1C8
#458	&1CA
#459	&1CB
#461	&1CD
#463	&1CF
#465	&1D1
#467	&1D3
#469	&1D5
#471	&1D7
#473	&1D9
#475	&1DB
#478	&1DE
#480	&1E0

Lowercase	
#596	&254
#392	&188
#598	&256
#599	&257
#396	&18C
#477	&1DD
#601	&259
#603	&25B
#402	&192
#608	&260
#611	&263
#617	&269
#616	&268
#409	&199
#623	&26F
#626	&272
#629	&275
#417	&1A1
#419	&1A3
#421	&1A5
#640	&280
#424	&1A8
#643	&283
#429	&1AD
#648	&288
#432	&1B0
#650	&28A
#651	&28B
#436	&1B4
#438	&1B6
#658	&292
#441	&1B9
#445	&1BD
#454	&1C6
#454	&1C6
#457	&1C9
#457	&1C9
#460	&1CC
#460	&1CC
#462	&1CE
#464	&1D0
#466	&1D2
#468	&1D4
#470	&1D6
#472	&1D8
#474	&1DA
#476	&1DC
#479	&1DF
#481	&1E1

Name
Latin Open O
Latin C With Hook
Latin African D
Latin D With Hook
Latin D With Topbar
Latin Reversed E
Latin Schwa
Latin Open E
Latin F With Hook
Latin G With Hook
Latin Gamma
Latin Iota
Latin I With Stroke
Latin K With Hook
Latin Turned M
Latin N With Left Hook
Latin O With Middle Tilde
Latin O With Horn
Latin Oi
Latin P With Hook
Latin Letter Yr
Latin Tone Two
Latin Esh
Latin T With Hook
Latin T With Retroflex Hook
Latin U With Horn
Latin Upsilon
Latin V With Hook
Latin Y With Hook
Latin Z With Stroke
Latin Ezh
Latin Ezh Reversed
Latin Tone Five
Latin Dz With Caron
Latin D With Small Letter Z With Caron
Latin Lj
Latin L With Small Letter J
Latin Nj
Latin N With Small Letter J
Latin A With Caron
Latin I With Caron
Latin O With Caron
Latin U With Caron
Latin U With Diaeresis And Macron
Latin U With Diaeresis And Acute
Latin U With Diaeresis And Caron
Latin U With Diaeresis And Grave
Latin A With Diaeresis And Macron
Latin A With Dot Above And Macron

Uppercase		Lowercase		Name
#482	&1E2	#483	&1E3	Latin Ae With Macron
#484	&1E4	#485	&1E5	Latin G With Stroke
#486	&1E6	#487	&1E7	Latin G With Caron
#488	&1E8	#489	&1E9	Latin K With Caron
#490	&1EA	#491	&1EB	Latin O With Ogonek
#492	&1EC	#493	&1ED	Latin O With Ogonek And Macron
#494	&1EE	#495	&1EF	Latin Ezh With Caron
#497	&1F1	#499	&1F3	Latin Dz
#498	&1F2	#499	&1F3	Latin D With Small Letter Z
#500	&1F4	#501	&1F5	Latin G With Acute
#502	&1F6	#405	&195	Latin Hwair
#503	&1F7	#447	&1BF	Latin Wynn
#504	&1F8	#505	&1F9	Latin N With Grave
#506	&1FA	#507	&1FB	Latin A With Ring Above And Acute
#508	&1FC	#509	&1FD	Latin Ae With Acute
#510	&1FE	#511	&1FF	Latin O With Stroke And Acute
#512	&200	#513	&201	Latin A With Double Grave
#514	&202	#515	&203	Latin A With Inverted Breve
#516	&204	#517	&205	Latin E With Double Grave
#518	&206	#519	&207	Latin E With Inverted Breve
#520	&208	#521	&209	Latin I With Double Grave
#522	&20A	#523	&20B	Latin I With Inverted Breve
#524	&20C	#525	&20D	Latin O With Double Grave
#526	&20E	#527	&20F	Latin O With Inverted Breve
#528	&210	#529	&211	Latin R With Double Grave
#530	&212	#531	&213	Latin R With Inverted Breve
#532	&214	#533	&215	Latin U With Double Grave
#534	&216	#535	&217	Latin U With Inverted Breve
#536	&218	#537	&219	Latin S With Comma Below
#538	&21A	#539	&21B	Latin T With Comma Below
#540	&21C	#541	&21D	Latin Yogh
#542	&21E	#543	&21F	Latin H With Caron
#544	&220	#414	&19E	Latin N With Long Right Leg
#546	&222	#547	&223	Latin Ou
#548	&224	#549	&225	Latin Z With Hook
#550	&226	#551	&227	Latin A With Dot Above
#552	&228	#553	&229	Latin E With Cedilla
#554	&22A	#555	&22B	Latin O With Diaeresis And Macron
#556	&22C	#557	&22D	Latin O With Tilde And Macron
#558	&22E	#559	&22F	Latin O With Dot Above
#560	&230	#561	&231	Latin O With Dot Above And Macron
#562	&232	#563	&233	Latin Y With Macron
#837	&345	#953	&3B9	Combining Greek Ypogegrammeni
#902	&386	#940	&3AC	Greek Alpha With Tonos
#904	&388	#941	&3AD	Greek Epsilon With Tonos
#905	&389	#942	&3AE	Greek Eta With Tonos
#906	&38A	#943	&3AF	Greek Iota With Tonos
#908	&38C	#972	&3CC	Greek Omicron With Tonos
#910	&38E	#973	&3CD	Greek Upsilon With Tonos

Uppercase		Lowercase		Name
#911	&38F	#974	&3CE	Greek Omega With Tonos
#913	&391	#945	&3B1	Greek Alpha
#914	&392	#946	&3B2	Greek Beta
#915	&393	#947	&3B3	Greek Gamma
#916	&394	#948	&3B4	Greek Delta
#917	&395	#949	&3B5	Greek Epsilon
#918	&396	#950	&3B6	Greek Zeta
#919	&397	#951	&3B7	Greek Eta
#920	&398	#952	&3B8	Greek Theta
#921	&399	#953	&3B9	Greek Iota
#922	&39A	#954	&3BA	Greek Kappa
#923	&39B	#955	&3BB	Greek Lamda
#924	&39C	#956	&3BC	Greek Mu
#925	&39D	#957	&3BD	Greek Nu
#926	&39E	#958	&3BE	Greek Xi
#927	&39F	#959	&3BF	Greek Omicron
#928	&3A0	#960	&3C0	Greek Pi
#929	&3A1	#961	&3C1	Greek Rho
#931	&3A3	#963	&3C3	Greek Sigma
#932	&3A4	#964	&3C4	Greek Tau
#933	&3A5	#965	&3C5	Greek Upsilon
#934	&3A6	#966	&3C6	Greek Phi
#935	&3A7	#967	&3C7	Greek Chi
#936	&3A8	#968	&3C8	Greek Psi
#937	&3A9	#969	&3C9	Greek Omega
#938	&3AA	#970	&3CA	Greek Iota With Dialytika
#939	&3AB	#971	&3CB	Greek Upsilon With Dialytika
#962	&3C2	#963	&3C3	Greek Small Letter Final Sigma
#976	&3D0	#946	&3B2	Greek Beta Symbol
#977	&3D1	#952	&3B8	Greek Theta Symbol
#981	&3D5	#966	&3C6	Greek Phi Symbol
#982	&3D6	#960	&3C0	Greek Pi Symbol
#984	&3D8	#985	&3D9	Greek Letter Archaic Koppa
#986	&3DA	#987	&3DB	Greek Letter Stigma
#988	&3DC	#989	&3DD	Greek Letter Digamma
#990	&3DE	#991	&3DF	Greek Letter Koppa
#992	&3E0	#993	&3E1	Greek Letter Sampi
#994	&3E2	#995	&3E3	Coptic Shei
#996	&3E4	#997	&3E5	Coptic Fei
#998	&3E6	#999	&3E7	Coptic Khei
#1000	&3E8	#1001	&3E9	Coptic Hori
#1002	&3EA	#1003	&3EB	Coptic Gangia
#1004	&3EC	#1005	&3ED	Coptic Shima
#1006	&3EE	#1007	&3EF	Coptic Dei
#1008	&3F0	#954	&3BA	Greek Kappa Symbol
#1009	&3F1	#961	&3C1	Greek Rho Symbol
#1010	&3F2	#963	&3C3	Greek Lunate Sigma Symbol
#1012	&3F4	#952	&3B8	Greek Capital Theta Symbol
#1013	&3F5	#949	&3B5	Greek Lunate Epsilon Symbol

Uppercase	
#1024	&400
#1025	&401
#1026	&402
#1027	&403
#1028	&404
#1029	&405
#1030	&406
#1031	&407
#1032	&408
#1033	&409
#1034	&40A
#1035	&40B
#1036	&40C
#1037	&40D
#1038	&40E
#1039	&40F
#1040	&410
#1041	&411
#1042	&412
#1043	&413
#1044	&414
#1045	&415
#1046	&416
#1047	&417
#1048	&418
#1049	&419
#1050	&41A
#1051	&41B
#1052	&41C
#1053	&41D
#1054	&41E
#1055	&41F
#1056	&420
#1057	&421
#1058	&422
#1059	&423
#1060	&424
#1061	&425
#1062	&426
#1063	&427
#1064	&428
#1065	&429
#1066	&42A
#1067	&42B
#1068	&42C
#1069	&42D
#1070	&42E
#1071	&42F
#1120	&460

Lowercase	
#1104	&450
#1105	&451
#1106	&452
#1107	&453
#1108	&454
#1109	&455
#1110	&456
#1111	&457
#1112	&458
#1113	&459
#1114	&45A
#1115	&45B
#1116	&45C
#1117	&45D
#1118	&45E
#1119	&45F
#1072	&430
#1073	&431
#1074	&432
#1075	&433
#1076	&434
#1077	&435
#1078	&436
#1079	&437
#1080	&438
#1081	&439
#1082	&43A
#1083	&43B
#1084	&43C
#1085	&43D
#1086	&43E
#1087	&43F
#1088	&440
#1089	&441
#1090	&442
#1091	&443
#1092	&444
#1093	&445
#1094	&446
#1095	&447
#1096	&448
#1097	&449
#1098	&44A
#1099	&44B
#1100	&44C
#1101	&44D
#1102	&44E
#1103	&44F
#1121	&461

Name
Cyrillic Ie With Grave
Cyrillic Io
Cyrillic Dje
Cyrillic Gje
Cyrillic Ukrainian Ie
Cyrillic Dze
Cyrillic Byelorussian-ukrainian I
Cyrillic Yi
Cyrillic Je
Cyrillic Lje
Cyrillic Nje
Cyrillic Tshe
Cyrillic Kje
Cyrillic I With Grave
Cyrillic Short U
Cyrillic Dzhe
Cyrillic A
Cyrillic Be
Cyrillic Ve
Cyrillic Ghe
Cyrillic De
Cyrillic Ie
Cyrillic Zhe
Cyrillic Ze
Cyrillic I
Cyrillic Short I
Cyrillic Ka
Cyrillic El
Cyrillic Em
Cyrillic En
Cyrillic O
Cyrillic Pe
Cyrillic Er
Cyrillic Es
Cyrillic Te
Cyrillic U
Cyrillic Ef
Cyrillic Ha
Cyrillic Tse
Cyrillic Che
Cyrillic Sha
Cyrillic Shcha
Cyrillic Hard Sign
Cyrillic Yeru
Cyrillic Soft Sign
Cyrillic E
Cyrillic Yu
Cyrillic Ya
Cyrillic Omega

Uppercase	
#1122	&462
#1124	&464
#1126	&466
#1128	&468
#1130	&46A
#1132	&46C
#1134	&46E
#1136	&470
#1138	&472
#1140	&474
#1142	&476
#1144	&478
#1146	&47A
#1148	&47C
#1150	&47E
#1152	&480
#1162	&48A
#1164	&48C
#1166	&48E
#1168	&490
#1170	&492
#1172	&494
#1174	&496
#1176	&498
#1178	&49A
#1180	&49C
#1182	&49E
#1184	&4A0
#1186	&4A2
#1188	&4A4
#1190	&4A6
#1192	&4A8
#1194	&4AA
#1196	&4AC
#1198	&4AE
#1200	&4B0
#1202	&4B2
#1204	&4B4
#1206	&4B6
#1208	&4B8
#1210	&4BA
#1212	&4BC
#1214	&4BE
#1217	&4C1
#1219	&4C3
#1221	&4C5
#1223	&4C7
#1225	&4C9
#1227	&4CB

Lowercase	
#1123	&463
#1125	&465
#1127	&467
#1129	&469
#1131	&46B
#1133	&46D
#1135	&46F
#1137	&471
#1139	&473
#1141	&475
#1143	&477
#1145	&479
#1147	&47B
#1149	&47D
#1151	&47F
#1153	&481
#1163	&48B
#1165	&48D
#1167	&48F
#1169	&491
#1171	&493
#1173	&495
#1175	&497
#1177	&499
#1179	&49B
#1181	&49D
#1183	&49F
#1185	&4A1
#1187	&4A3
#1189	&4A5
#1191	&4A7
#1193	&4A9
#1195	&4AB
#1197	&4AD
#1199	&4AF
#1201	&4B1
#1203	&4B3
#1205	&4B5
#1207	&4B7
#1209	&4B9
#1211	&4BB
#1213	&4BD
#1215	&4BF
#1218	&4C2
#1220	&4C4
#1222	&4C6
#1224	&4C8
#1226	&4CA
#1228	&4CC

Name
Cyrillic Yat
Cyrillic Iotified E
Cyrillic Little Yus
Cyrillic Iotified Little Yus
Cyrillic Big Yus
Cyrillic Iotified Big Yus
Cyrillic Ksi
Cyrillic Psi
Cyrillic Fita
Cyrillic Izhitsa
Cyrillic Izhitsa With Double Grave Accent
Cyrillic Uk
Cyrillic Round Omega
Cyrillic Omega With Titlo
Cyrillic Ot
Cyrillic Koppa
Cyrillic Short I With Tail
Cyrillic Semisoft Sign
Cyrillic Er With Tick
Cyrillic Ghe With Upturn
Cyrillic Ghe With Stroke
Cyrillic Ghe With Middle Hook
Cyrillic Zhe With Descender
Cyrillic Ze With Descender
Cyrillic Ka With Descender
Cyrillic Ka With Vertical Stroke
Cyrillic Ka With Stroke
Cyrillic Bashkir Ka
Cyrillic En With Descender
Cyrillic Capital Ligature En Ghe
Cyrillic Pe With Middle Hook
Cyrillic Abkhasian Ha
Cyrillic Es With Descender
Cyrillic Te With Descender
Cyrillic Straight U
Cyrillic Straight U With Stroke
Cyrillic Ha With Descender
Cyrillic Capital Ligature Te Tse
Cyrillic Che With Descender
Cyrillic Che With Vertical Stroke
Cyrillic Shha
Cyrillic Abkhasian Che
Cyrillic Abkhasian Che With Descender
Cyrillic Zhe With Breve
Cyrillic Ka With Hook
Cyrillic El With Tail
Cyrillic En With Hook
Cyrillic En With Tail
Cyrillic Khakassian Che

Uppercase	
#1229	&4CD
#1232	&4D0
#1234	&4D2
#1236	&4D4
#1238	&4D6
#1240	&4D8
#1242	&4DA
#1244	&4DC
#1246	&4DE
#1248	&4E0
#1250	&4E2
#1252	&4E4
#1254	&4E6
#1256	&4E8
#1258	&4EA
#1260	&4EC
#1262	&4EE
#1264	&4F0
#1266	&4F2
#1268	&4F4
#1272	&4F8
#1280	&500
#1282	&502
#1284	&504
#1286	&506
#1288	&508
#1290	&50A
#1292	&50C
#1294	&50E
#1329	&531
#1330	&532
#1331	&533
#1332	&534
#1333	&535
#1334	&536
#1335	&537
#1336	&538
#1337	&539
#1338	&53A
#1339	&53B
#1340	&53C
#1341	&53D
#1342	&53E
#1343	&53F
#1344	&540
#1345	&541
#1346	&542
#1347	&543
#1348	&544

Lowercase	
#1230	&4CE
#1233	&4D1
#1235	&4D3
#1237	&4D5
#1239	&4D7
#1241	&4D9
#1243	&4DB
#1245	&4DD
#1247	&4DF
#1249	&4E1
#1251	&4E3
#1253	&4E5
#1255	&4E7
#1257	&4E9
#1259	&4EB
#1261	&4ED
#1263	&4EF
#1265	&4F1
#1267	&4F3
#1269	&4F5
#1273	&4F9
#1281	&501
#1283	&503
#1285	&505
#1287	&507
#1289	&509
#1291	&50B
#1293	&50D
#1295	&50F
#1377	&561
#1378	&562
#1379	&563
#1380	&564
#1381	&565
#1382	&566
#1383	&567
#1384	&568
#1385	&569
#1386	&56A
#1387	&56B
#1388	&56C
#1389	&56D
#1390	&56E
#1391	&56F
#1392	&570
#1393	&571
#1394	&572
#1395	&573
#1396	&574

Name
Cyrillic Em With Tail
Cyrillic A With Breve
Cyrillic A With Diaeresis
Cyrillic Capital Ligature A Ie
Cyrillic Ie With Breve
Cyrillic Schwa
Cyrillic Schwa With Diaeresis
Cyrillic Zhe With Diaeresis
Cyrillic Ze With Diaeresis
Cyrillic Abkhastian Dze
Cyrillic I With Macron
Cyrillic I With Diaeresis
Cyrillic O With Diaeresis
Cyrillic Barred O
Cyrillic Barred O With Diaeresis
Cyrillic E With Diaeresis
Cyrillic U With Macron
Cyrillic U With Diaeresis
Cyrillic U With Double Acute
Cyrillic Che With Diaeresis
Cyrillic Yeru With Diaeresis
Cyrillic Komi De
Cyrillic Komi Dje
Cyrillic Komi Zje
Cyrillic Komi Dkje
Cyrillic Komi Lje
Cyrillic Komi Nje
Cyrillic Komi Sje
Cyrillic Komi Tje
Armenian Ayb
Armenian Ben
Armenian Gim
Armenian Da
Armenian Ech
Armenian Za
Armenian Eh
Armenian Et
Armenian To
Armenian Zhe
Armenian Ini
Armenian Liwn
Armenian Xeh
Armenian Ca
Armenian Ken
Armenian Ho
Armenian Ja
Armenian Ghad
Armenian Cheh
Armenian Men

Uppercase	
#1349	&545
#1350	&546
#1351	&547
#1352	&548
#1353	&549
#1354	&54A
#1355	&54B
#1356	&54C
#1357	&54D
#1358	&54E
#1359	&54F
#1360	&550
#1361	&551
#1362	&552
#1363	&553
#1364	&554
#1365	&555
#1366	&556
#7680	&1E00
#7682	&1E02
#7684	&1E04
#7686	&1E06
#7688	&1E08
#7690	&1E0A
#7692	&1E0C
#7694	&1E0E
#7696	&1E10
#7698	&1E12
#7700	&1E14
#7702	&1E16
#7704	&1E18
#7706	&1E1A
#7708	&1E1C
#7710	&1E1E
#7712	&1E20
#7714	&1E22
#7716	&1E24
#7718	&1E26
#7720	&1E28
#7722	&1E2A
#7724	&1E2C
#7726	&1E2E
#7728	&1E30
#7730	&1E32
#7732	&1E34
#7734	&1E36
#7736	&1E38
#7738	&1E3A
#7740	&1E3C

Lowercase	
#1397	&575
#1398	&576
#1399	&577
#1400	&578
#1401	&579
#1402	&57A
#1403	&57B
#1404	&57C
#1405	&57D
#1406	&57E
#1407	&57F
#1408	&580
#1409	&581
#1410	&582
#1411	&583
#1412	&584
#1413	&585
#1414	&586
#7681	&1E01
#7683	&1E03
#7685	&1E05
#7687	&1E07
#7689	&1E09
#7691	&1E0B
#7693	&1E0D
#7695	&1E0F
#7697	&1E11
#7699	&1E13
#7701	&1E15
#7703	&1E17
#7705	&1E19
#7707	&1E1B
#7709	&1E1D
#7711	&1E1F
#7713	&1E21
#7715	&1E23
#7717	&1E25
#7719	&1E27
#7721	&1E29
#7723	&1E2B
#7725	&1E2D
#7727	&1E2F
#7729	&1E31
#7731	&1E33
#7733	&1E35
#7735	&1E37
#7737	&1E39
#7739	&1E3B
#7741	&1E3D

Name
Armenian Yi
Armenian Now
Armenian Sha
Armenian Vo
Armenian Cha
Armenian Peh
Armenian Jheh
Armenian Ra
Armenian Seh
Armenian Vew
Armenian Tiwn
Armenian Reh
Armenian Co
Armenian Yiwn
Armenian Piwr
Armenian Keh
Armenian Oh
Armenian Feh
Latin A With Ring Below
Latin B With Dot Above
Latin B With Dot Below
Latin B With Line Below
Latin C With Cedilla And Acute
Latin D With Dot Above
Latin D With Dot Below
Latin D With Line Below
Latin D With Cedilla
Latin D With Circumflex Below
Latin E With Macron And Grave
Latin E With Macron And Acute
Latin E With Circumflex Below
Latin E With Tilde Below
Latin E With Cedilla And Breve
Latin F With Dot Above
Latin G With Macron
Latin H With Dot Above
Latin H With Dot Below
Latin H With Diaeresis
Latin H With Cedilla
Latin H With Breve Below
Latin I With Tilde Below
Latin I With Diaeresis And Acute
Latin K With Acute
Latin K With Dot Below
Latin K With Line Below
Latin L With Dot Below
Latin L With Dot Below And Macron
Latin L With Line Below
Latin L With Circumflex Below

Uppercase	
#7742	&1E3E
#7744	&1E40
#7746	&1E42
#7748	&1E44
#7750	&1E46
#7752	&1E48
#7754	&1E4A
#7756	&1E4C
#7758	&1E4E
#7760	&1E50
#7762	&1E52
#7764	&1E54
#7766	&1E56
#7768	&1E58
#7770	&1E5A
#7772	&1E5C
#7774	&1E5E
#7776	&1E60
#7778	&1E62
#7780	&1E64
#7782	&1E66
#7784	&1E68
#7786	&1E6A
#7788	&1E6C
#7790	&1E6E
#7792	&1E70
#7794	&1E72
#7796	&1E74
#7798	&1E76
#7800	&1E78
#7802	&1E7A
#7804	&1E7C
#7806	&1E7E
#7808	&1E80
#7810	&1E82
#7812	&1E84
#7814	&1E86
#7816	&1E88
#7818	&1E8A
#7820	&1E8C
#7822	&1E8E
#7824	&1E90
#7826	&1E92
#7828	&1E94
#7835	&1E9B
#7840	&1EA0
#7842	&1EA2
#7844	&1EA4
#7846	&1EA6

Lowercase	
#7743	&1E3F
#7745	&1E41
#7747	&1E43
#7749	&1E45
#7751	&1E47
#7753	&1E49
#7755	&1E4B
#7757	&1E4D
#7759	&1E4F
#7761	&1E51
#7763	&1E53
#7765	&1E55
#7767	&1E57
#7769	&1E59
#7771	&1E5B
#7773	&1E5D
#7775	&1E5F
#7777	&1E61
#7779	&1E63
#7781	&1E65
#7783	&1E67
#7785	&1E69
#7787	&1E6B
#7789	&1E6D
#7791	&1E6F
#7793	&1E71
#7795	&1E73
#7797	&1E75
#7799	&1E77
#7801	&1E79
#7803	&1E7B
#7805	&1E7D
#7807	&1E7F
#7809	&1E81
#7811	&1E83
#7813	&1E85
#7815	&1E87
#7817	&1E89
#7819	&1E8B
#7821	&1E8D
#7823	&1E8F
#7825	&1E91
#7827	&1E93
#7829	&1E95
#7777	&1E61
#7841	&1EA1
#7843	&1EA3
#7845	&1EA5
#7847	&1EA7

Name
Latin M With Acute
Latin M With Dot Above
Latin M With Dot Below
Latin N With Dot Above
Latin N With Dot Below
Latin N With Line Below
Latin N With Circumflex Below
Latin O With Tilde And Acute
Latin O With Tilde And Diaeresis
Latin O With Macron And Grave
Latin O With Macron And Acute
Latin P With Acute
Latin P With Dot Above
Latin R With Dot Above
Latin R With Dot Below
Latin R With Dot Below And Macron
Latin R With Line Below
Latin S With Dot Above
Latin S With Dot Below
Latin S With Acute And Dot Above
Latin S With Caron And Dot Above
Latin S With Dot Below And Dot Above
Latin T With Dot Above
Latin T With Dot Below
Latin T With Line Below
Latin T With Circumflex Below
Latin U With Diaeresis Below
Latin U With Tilde Below
Latin U With Circumflex Below
Latin U With Tilde And Acute
Latin U With Macron And Diaeresis
Latin V With Tilde
Latin V With Dot Below
Latin W With Grave
Latin W With Acute
Latin W With Diaeresis
Latin W With Dot Above
Latin W With Dot Below
Latin X With Dot Above
Latin X With Diaeresis
Latin Y With Dot Above
Latin Z With Circumflex
Latin Z With Dot Below
Latin Z With Line Below
Latin Small Letter Long S With Dot Above
Latin A With Dot Below
Latin A With Hook Above
Latin A With Circumflex And Acute
Latin A With Circumflex And Grave

Uppercase		Lowercase		Name
#7848	&1EA8	#7849	&1EA9	Latin A With Circumflex And Hook Above
#7850	&1EAA	#7851	&1EAB	Latin A With Circumflex And Tilde
#7852	&1EAC	#7853	&1EAD	Latin A With Circumflex And Dot Below
#7854	&1EAE	#7855	&1EAF	Latin A With Breve And Acute
#7856	&1EB0	#7857	&1EB1	Latin A With Breve And Grave
#7858	&1EB2	#7859	&1EB3	Latin A With Breve And Hook Above
#7860	&1EB4	#7861	&1EB5	Latin A With Breve And Tilde
#7862	&1EB6	#7863	&1EB7	Latin A With Breve And Dot Below
#7864	&1EB8	#7865	&1EB9	Latin E With Dot Below
#7866	&1EBA	#7867	&1EBB	Latin E With Hook Above
#7868	&1EBC	#7869	&1EBD	Latin E With Tilde
#7870	&1EBE	#7871	&1EBF	Latin E With Circumflex And Acute
#7872	&1EC0	#7873	&1EC1	Latin E With Circumflex And Grave
#7874	&1EC2	#7875	&1EC3	Latin E With Circumflex And Hook Above
#7876	&1EC4	#7877	&1EC5	Latin E With Circumflex And Tilde
#7878	&1EC6	#7879	&1EC7	Latin E With Circumflex And Dot Below
#7880	&1EC8	#7881	&1EC9	Latin I With Hook Above
#7882	&1ECA	#7883	&1ECB	Latin I With Dot Below
#7884	&1ECC	#7885	&1ECD	Latin O With Dot Below
#7886	&1ECE	#7887	&1ECF	Latin O With Hook Above
#7888	&1ED0	#7889	&1ED1	Latin O With Circumflex And Acute
#7890	&1ED2	#7891	&1ED3	Latin O With Circumflex And Grave
#7892	&1ED4	#7893	&1ED5	Latin O With Circumflex And Hook Above
#7894	&1ED6	#7895	&1ED7	Latin O With Circumflex And Tilde
#7896	&1ED8	#7897	&1ED9	Latin O With Circumflex And Dot Below
#7898	&1EDA	#7899	&1EDB	Latin O With Horn And Acute
#7900	&1EDC	#7901	&1EDD	Latin O With Horn And Grave
#7902	&1EDE	#7903	&1EDF	Latin O With Horn And Hook Above
#7904	&1EE0	#7905	&1EE1	Latin O With Horn And Tilde
#7906	&1EE2	#7907	&1EE3	Latin O With Horn And Dot Below
#7908	&1EE4	#7909	&1EE5	Latin U With Dot Below
#7910	&1EE6	#7911	&1EE7	Latin U With Hook Above
#7912	&1EE8	#7913	&1EE9	Latin U With Horn And Acute
#7914	&1EEA	#7915	&1EEB	Latin U With Horn And Grave
#7916	&1EEC	#7917	&1EED	Latin U With Horn And Hook Above
#7918	&1EEE	#7919	&1EEF	Latin U With Horn And Tilde
#7920	&1EF0	#7921	&1EF1	Latin U With Horn And Dot Below
#7922	&1EF2	#7923	&1EF3	Latin Y With Grave
#7924	&1EF4	#7925	&1EF5	Latin Y With Dot Below
#7926	&1EF6	#7927	&1EF7	Latin Y With Hook Above
#7928	&1EF8	#7929	&1EF9	Latin Y With Tilde
#7944	&1F08	#7936	&1F00	Greek Alpha With Psili
#7945	&1F09	#7937	&1F01	Greek Alpha With Dasia
#7946	&1F0A	#7938	&1F02	Greek Alpha With Psili And Varia
#7947	&1F0B	#7939	&1F03	Greek Alpha With Dasia And Varia
#7948	&1F0C	#7940	&1F04	Greek Alpha With Psili And Oxia
#7949	&1F0D	#7941	&1F05	Greek Alpha With Dasia And Oxia
#7950	&1F0E	#7942	&1F06	Greek Alpha With Psili And Perispomeni
#7951	&1F0F	#7943	&1F07	Greek Alpha With Dasia And Perispomeni

Uppercase	
#7960	&1F18
#7961	&1F19
#7962	&1F1A
#7963	&1F1B
#7964	&1F1C
#7965	&1F1D
#7976	&1F28
#7977	&1F29
#7978	&1F2A
#7979	&1F2B
#7980	&1F2C
#7981	&1F2D
#7982	&1F2E
#7983	&1F2F
#7992	&1F38
#7993	&1F39
#7994	&1F3A
#7995	&1F3B
#7996	&1F3C
#7997	&1F3D
#7998	&1F3E
#7999	&1F3F
#8008	&1F48
#8009	&1F49
#8010	&1F4A
#8011	&1F4B
#8012	&1F4C
#8013	&1F4D
#8025	&1F59
#8027	&1F5B
#8029	&1F5D
#8031	&1F5F
#8040	&1F68
#8041	&1F69
#8042	&1F6A
#8043	&1F6B
#8044	&1F6C
#8045	&1F6D
#8046	&1F6E
#8047	&1F6F
#8120	&1FB8
#8121	&1FB9
#8122	&1FBA
#8123	&1FBB
#8126	&1FBE
#8136	&1FC8
#8137	&1FC9
#8138	&1FCA
#8139	&1FCB

Lowercase	
#7952	&1F10
#7953	&1F11
#7954	&1F12
#7955	&1F13
#7956	&1F14
#7957	&1F15
#7968	&1F20
#7969	&1F21
#7970	&1F22
#7971	&1F23
#7972	&1F24
#7973	&1F25
#7974	&1F26
#7975	&1F27
#7984	&1F30
#7985	&1F31
#7986	&1F32
#7987	&1F33
#7988	&1F34
#7989	&1F35
#7990	&1F36
#7991	&1F37
#8000	&1F40
#8001	&1F41
#8002	&1F42
#8003	&1F43
#8004	&1F44
#8005	&1F45
#8017	&1F51
#8019	&1F53
#8021	&1F55
#8023	&1F57
#8032	&1F60
#8033	&1F61
#8034	&1F62
#8035	&1F63
#8036	&1F64
#8037	&1F65
#8038	&1F66
#8039	&1F67
#8112	&1FB0
#8113	&1FB1
#8048	&1F70
#8049	&1F71
#953	&3B9
#8050	&1F72
#8051	&1F73
#8052	&1F74
#8053	&1F75

Name
Greek Epsilon With Psili
Greek Epsilon With Dasia
Greek Epsilon With Psili And Varia
Greek Epsilon With Dasia And Varia
Greek Epsilon With Psili And Oxia
Greek Epsilon With Dasia And Oxia
Greek Eta With Psili
Greek Eta With Dasia
Greek Eta With Psili And Varia
Greek Eta With Dasia And Varia
Greek Eta With Psili And Oxia
Greek Eta With Dasia And Oxia
Greek Eta With Psili And Perispomeni
Greek Eta With Dasia And Perispomeni
Greek Iota With Psili
Greek Iota With Dasia
Greek Iota With Psili And Varia
Greek Iota With Dasia And Varia
Greek Iota With Psili And Oxia
Greek Iota With Dasia And Oxia
Greek Iota With Psili And Perispomeni
Greek Iota With Dasia And Perispomeni
Greek Omicron With Psili
Greek Omicron With Dasia
Greek Omicron With Psili And Varia
Greek Omicron With Dasia And Varia
Greek Omicron With Psili And Oxia
Greek Omicron With Dasia And Oxia
Greek Upsilon With Dasia
Greek Upsilon With Dasia And Varia
Greek Upsilon With Dasia And Oxia
Greek Upsilon With Dasia And Perispomeni
Greek Omega With Psili
Greek Omega With Dasia
Greek Omega With Psili And Varia
Greek Omega With Dasia And Varia
Greek Omega With Psili And Oxia
Greek Omega With Dasia And Oxia
Greek Omega With Psili And Perispomeni
Greek Omega With Dasia And Perispomeni
Greek Alpha With Vrachy
Greek Alpha With Macron
Greek Alpha With Varia
Greek Alpha With Oxia
Greek Prosgegrammeni
Greek Epsilon With Varia
Greek Epsilon With Oxia
Greek Eta With Varia
Greek Eta With Oxia

Uppercase		Lowercase		Name
#8152	&1FD8	#8144	&1FD0	Greek Iota With Vrachy
#8153	&1FD9	#8145	&1FD1	Greek Iota With Macron
#8154	&1FDA	#8054	&1F76	Greek Iota With Varia
#8155	&1FDB	#8055	&1F77	Greek Iota With Oxia
#8168	&1FE8	#8160	&1FE0	Greek Upsilon With Vrachy
#8169	&1FE9	#8161	&1FE1	Greek Upsilon With Macron
#8170	&1FEA	#8058	&1F7A	Greek Upsilon With Varia
#8171	&1FEB	#8059	&1F7B	Greek Upsilon With Oxia
#8172	&1FEC	#8165	&1FE5	Greek Rho With Dasia
#8184	&1FF8	#8056	&1F78	Greek Omicron With Varia
#8185	&1FF9	#8057	&1F79	Greek Omicron With Oxia
#8186	&1FFA	#8060	&1F7C	Greek Omega With Varia
#8187	&1FFB	#8061	&1F7D	Greek Omega With Oxia
#8486	&2126	#969	&3C9	Ohm Sign
#8490	&212A	#107	&6B	Kelvin Sign
#8491	&212B	#229	&E5	Angstrom Sign
#8544	&2160	#8560	&2170	Roman Numeral One
#8545	&2161	#8561	&2171	Roman Numeral Two
#8546	&2162	#8562	&2172	Roman Numeral Three
#8547	&2163	#8563	&2173	Roman Numeral Four
#8548	&2164	#8564	&2174	Roman Numeral Five
#8549	&2165	#8565	&2175	Roman Numeral Six
#8550	&2166	#8566	&2176	Roman Numeral Seven
#8551	&2167	#8567	&2177	Roman Numeral Eight
#8552	&2168	#8568	&2178	Roman Numeral Nine
#8553	&2169	#8569	&2179	Roman Numeral Ten
#8554	&216A	#8570	&217A	Roman Numeral Eleven
#8555	&216B	#8571	&217B	Roman Numeral Twelve
#8556	&216C	#8572	&217C	Roman Numeral Fifty
#8557	&216D	#8573	&217D	Roman Numeral One Hundred
#8558	&216E	#8574	&217E	Roman Numeral Five Hundred
#8559	&216F	#8575	&217F	Roman Numeral One Thousand
#9398	&24B6	#9424	&24D0	Circled Latin A
#9399	&24B7	#9425	&24D1	Circled Latin B
#9400	&24B8	#9426	&24D2	Circled Latin C
#9401	&24B9	#9427	&24D3	Circled Latin D
#9402	&24BA	#9428	&24D4	Circled Latin E
#9403	&24BB	#9429	&24D5	Circled Latin F
#9404	&24BC	#9430	&24D6	Circled Latin G
#9405	&24BD	#9431	&24D7	Circled Latin H
#9406	&24BE	#9432	&24D8	Circled Latin I
#9407	&24BF	#9433	&24D9	Circled Latin J
#9408	&24C0	#9434	&24DA	Circled Latin K
#9409	&24C1	#9435	&24DB	Circled Latin L
#9410	&24C2	#9436	&24DC	Circled Latin M
#9411	&24C3	#9437	&24DD	Circled Latin N
#9412	&24C4	#9438	&24DE	Circled Latin O
#9413	&24C5	#9439	&24DF	Circled Latin P
#9414	&24C6	#9440	&24E0	Circled Latin Q

Uppercase	
#9415	&24C7
#9416	&24C8
#9417	&24C9
#9418	&24CA
#9419	&24CB
#9420	&24CC
#9421	&24CD
#9422	&24CE
#9423	&24CF
#65313	&FF21
#65314	&FF22
#65315	&FF23
#65316	&FF24
#65317	&FF25
#65318	&FF26
#65319	&FF27
#65320	&FF28
#65321	&FF29
#65322	&FF2A
#65323	&FF2B
#65324	&FF2C
#65325	&FF2D
#65326	&FF2E
#65327	&FF2F
#65328	&FF30
#65329	&FF31
#65330	&FF32
#65331	&FF33
#65332	&FF34
#65333	&FF35
#65334	&FF36
#65335	&FF37
#65336	&FF38
#65337	&FF39
#65338	&FF3A

Lowercase	
#9441	&24E1
#9442	&24E2
#9443	&24E3
#9444	&24E4
#9445	&24E5
#9446	&24E6
#9447	&24E7
#9448	&24E8
#9449	&24E9
#65345	&FF41
#65346	&FF42
#65347	&FF43
#65348	&FF44
#65349	&FF45
#65350	&FF46
#65351	&FF47
#65352	&FF48
#65353	&FF49
#65354	&FF4A
#65355	&FF4B
#65356	&FF4C
#65357	&FF4D
#65358	&FF4E
#65359	&FF4F
#65360	&FF50
#65361	&FF51
#65362	&FF52
#65363	&FF53
#65364	&FF54
#65365	&FF55
#65366	&FF56
#65367	&FF57
#65368	&FF58
#65369	&FF59
#65370	&FF5A

Name
Circled Latin R
Circled Latin S
Circled Latin T
Circled Latin U
Circled Latin V
Circled Latin W
Circled Latin X
Circled Latin Y
Circled Latin Z
Fullwidth Latin A
Fullwidth Latin B
Fullwidth Latin C
Fullwidth Latin D
Fullwidth Latin E
Fullwidth Latin F
Fullwidth Latin G
Fullwidth Latin H
Fullwidth Latin I
Fullwidth Latin J
Fullwidth Latin K
Fullwidth Latin L
Fullwidth Latin M
Fullwidth Latin N
Fullwidth Latin O
Fullwidth Latin P
Fullwidth Latin Q
Fullwidth Latin R
Fullwidth Latin S
Fullwidth Latin T
Fullwidth Latin U
Fullwidth Latin V
Fullwidth Latin W
Fullwidth Latin X
Fullwidth Latin Y
Fullwidth Latin Z

Bibliography

American National Standards Institute. (ANSI). (1974). American National Standard programming language COBOL (ANSI ; X3.23-1974).

Appel, Andrew W. (1998). Modern Compiler Implementation in C. Cambridge University Press, 40 West 20th Street, New York City, New York 10011-4211

Astudillo, Manuel. C++ GOLD Parser Engine. e-mail: d00mas@efd.lth.se

Cohan, Daniel I. A. (1991), Introduction to Computer Theory, John Wiley & Sons, Inc. New York, ISBN 0-471-51010-6

Fischer, Charles N. & LeBlanc Jr., Richard J. (1988). Crafting a Compiler, The Benjamin/Cummings Publishing Company Inc., 2727 Sand Hill Road, Menlo Park, California 94025

GOLD Parser Website. <http://www.devincook.com/goldparser>.

Ganssle, Jack & Barr, Michael. (2003). Embedded Systems Dictionary (55-56). CMP Books., 6600 Silacci Way, Gilroy, CA 95020

Hawkins, Matthew. Java GOLD Parser Engine. <http://www.hawkini.com>

ISO/IEC, International Standards Organization / International Engineering Consortium. 1, rue de Varembé,
Case postale 56, CH-1211 Geneva 20, Switzerland

Khachab, Ibrahim. Modified Delphi GOLD Parser Engine. e-mail: ibrahim@euronia.it

Klimstra, Marcus. C# GOLD Parser Engine. e-mail: klimstra@home.nl

Johnson, Stephen C. (1979). Yacc: Yet Another Compiler-Compiler, AT&T Bell Laboratories, Murray
Hill, New Jersey 07974

Kernighan, Brian W. and Ritchie, Dennis M. (1988). The C Programming Language Second Edition.
Prentice-Hall, Inc.

Loudan, Kenneth C . (1997). Compiler Construction, PWS Publishing Company, 20 Park Plaza, Boston,
MA 02116-4324

Microsoft Corporation. (2003). Visual Basic .NET, <http://msdn.microsoft.com/vbasic/>

Parr, Terrence. (2000). ANTLR Reference Manual, Retrieved 08-10-2001 from <http://wwwantlr.org/doc>

Rai, Alexandre. Delphi GOLD Parser Engine. e-mail: riccio@gmx.at

Smalltalk Industry Council (STIC), Smalltalk Programming Language Website. <http://www.smalltalk.org>

Ugurel, Eylem. C++ GOLD Parser Engine. e-mail: eylemugurel@hotmail.com

van der Geer , Martin. Delphi GOLD Parser Engine. email: Beany@cloud.demon.nl

van Loenhout, Robert. C# GOLD Parser Engine. e-mail: rvl@software-engineer.net

Unicode Consortium, The. (2003). Unicode Code Charts, Retrieved 2003 from <http://www.unicode.org>

Wilbanks, Reggie. Visual Basic .NET GOLD Parser Engine. e-mail: rwilbanks@starband.net

Wilensky, Robert. (1984). LISPcraft. W.W. Norton & Company, Inc., 500 Fifth Avenue, New York, NY 10110

World Wide Web Consortium (W3C) (2003). Extensible Markup Language (XML). Retrieved 2003 from <http://www.w3.org/XML/>

World Wide Web Consortium (W3C) (2003). Cascading Style Sheets Home Page (CSS). Retrieved from <http://www.w3.org/Style/CSS/>

World Wide Web Consortium (W3C) (1995-2004). HyperText Markup Language (HTML) Home Page. Retrieved from <http://www.w3.org/MarkUp/>