# Table of Contents

1 Creating Parsers...........................................................................................................6  
1.1 Starting the Program.................................................................................................6  
1.2 Specifying Grammars...............................................................................................9  
  1.2.1 Terminals...........................................................................................................9  
  1.2.2 Error Tokens..................................................................................................10  
  1.2.3 Nonterminals..................................................................................................10  
  1.2.4 Rules..............................................................................................................11  
  1.2.5 Precedences..................................................................................................12  
  1.2.6 Start Context.................................................................................................12  
1.3 Calculating Automatons............................................................................................13  
1.4 Resolving Parsing Conflicts......................................................................................15  
1.5 Calculating Essential Error Entries..........................................................................15  
1.6 Calculating Unit Rule Eliminations..........................................................................16  
1.7 Simulating Parsers....................................................................................................16  
  1.7.1 Regular Expression Parser Simulation..............................................................20  
  1.7.2 Deterministic Context Free Parser Simulation................................................20  
1.8 Specifying Code Sections..........................................................................................20  
  1.8.1 Terminal Code................................................................................................20  
  1.8.2 Rule Code......................................................................................................20  
  1.8.3 Error Code....................................................................................................23  
  1.8.4 Conflict Code................................................................................................23  
  1.8.5 Class Code....................................................................................................23  
  1.8.6 File Source Code...........................................................................................23  
1.9 Generating Parser Code............................................................................................26  
2 User Interface Components............................................................................................27  
  2.1 Default Mainviews.................................................................................................27  
  2.2 Edit Window Operations.........................................................................................28  
  2.3 Subviews...............................................................................................................29  
    2.3.1 Grammar Input...............................................................................................29  
      2.3.1.1 Terminal................................................................................................29  
      2.3.1.2 Error Token............................................................................................30  
      2.3.1.3 Nonterminal...........................................................................................31  
      2.3.1.4 Implicit...................................................................................................32  
      2.3.1.5 Rule.......................................................................................................32  
      2.3.1.6 Precedence.............................................................................................33  
    2.3.2 Conflict Resolution and Source Code..............................................................34  
      2.3.2.1 Conflict Select.......................................................................................34  
      2.3.2.2 Action....................................................................................................35  
      2.3.2.3 Status....................................................................................................35  
      2.3.2.4 Conflict Code.......................................................................................36  
    2.3.3 Error Source Code.........................................................................................37  
      2.3.3.1 Error Select.............................................................................................37  
      2.3.3.2 Error Code.............................................................................................37
2.3.4 Terminal Source Code
2.3.4.1 Terminal Select
2.3.4.2 Terminal Code
2.3.5 Rule Source Code
2.3.5.1 Rule Select
2.3.5.2 Rule Code
2.3.6 Class Source Code
2.3.7 File Source Code
2.3.8 Automaton Machine
2.3.9 Automaton Simulate
2.3.9.1 Control
2.3.9.2 Input
2.3.9.3 Parse
2.4 Dialogs
2.4.1 File
2.4.2 Grammar Edit
2.4.2.1 Start Context
2.4.2.2 Terminal
2.4.2.3 Error Token
2.4.2.4 Nonterminal
2.4.2.5 Rule
2.4.2.6 Unit Rule Elimination
2.4.2.7 Precedence
2.4.3 Automaton Calculation
2.4.3.1 Calculate
2.4.3.2 Error Entries
2.4.3.3 Unit Rules
2.4.4 Generate Code
2.4.5 Find String
2.4.6 Simulator Error Token
2.4.7 Simulator Options
3 Parsing Calculations and Code Generation
3.1 Regular Expression Parsers
3.1.1 Language Grammar
3.1.1.1 Special Characters
3.1.1.2 Non-special Characters
3.1.1.3 Subexpression Grouping
3.1.1.4 Repeat Characters
3.1.1.5 Or-Operator Character
3.1.1.6 Escape Character
3.1.1.7 Double Quote Character
3.1.1.8 Character Class Specifications
3.1.1.9 Match All Character
3.1.1.10 Repeat Ranges
3.1.1.11 Trailing Context
Illustration Index

Illustration 1  CPG Application Main Window  
Illustration 2  Open Data File Dialog Window  
Illustration 3  Grammar Input Mainviews  
Illustration 4  Symbol Input Subviews  
Illustration 5  Terminal Edit Dialog  
Illustration 6  Error Token Edit Dialog  
Illustration 7  Nonterminal Edit Dialog  
Illustration 8  Rule Edit Dialog  
Illustration 9  Unit Rule Elimination Flag Dialog  
Illustration 10  Precedence Edit Dialog  
Illustration 11  Parser Calculation Dialog  
Illustration 12  Conflict Resolution and Code Mainview  
Illustration 13  Essential Error Entry Calculation Dialog  
Illustration 14  Unit Rule Elimination Calculation Dialog  
Illustration 15  Parser Simulate Mainview - Regular Expression Automaton  
Illustration 16  Parser Simulate Mainview - Deterministic Context Free Automaton  
Illustration 17  Simulator Options Dialog  
Illustration 18  Terminal Code Mainview  
Illustration 19  Rule Code Mainview  
Illustration 20  Simulator Error Token Dialog  
Illustration 21  Error Code Mainview  
Illustration 22  File Source Code Mainview  
Illustration 23  Generate Code Dialog  
Illustration 24  Generated Regular Expression Type Parser Data  
Illustration 25  Structure and Numbering Scheme of Regular Expression Parser Tables - 1  
Illustration 26  Structure and Numbering Scheme of Regular Expression Parser Tables - 2  
Illustration 27  Regular Expression Parser - Table Based Logic - 1  
Illustration 28  Regular Expression Parser - Table Based Logic - 2  
Illustration 29  Regular Expression Parser - Table Based Logic - 3  
Illustration 30  Regular Expression Parser - Table Based Logic - 4  
Illustration 31  Regular Expression Parser - Table Based Logic - 5  
Illustration 32  Generated Context Free Parser Data  
Illustration 33  Grammar Symbol Numbering Scheme  
Illustration 34  Action Table Structure and Numbering Scheme  
Illustration 35  Goto Table Structure and Numbering Scheme  
Illustration 36  Error Entry Numbering  
Illustration 37  Error Function Aggregation - Default Error Entries  
Illustration 38  Error Function Aggregation - Essential Error Entries  
Illustration 39  Deterministic Context Free Parser - Table Based Logic - 1  
Illustration 40  Deterministic Context Free Parser - Table Based Logic - 2  
Illustration 41  Deterministic Context Free Parser - Table Based Logic - 3  
Illustration 42  Deterministic Context Free Parser - Table Based Logic - 4
1 Creating Parsers

Compiler – a Parser Generator Program (CPG) is a system for specifying formal languages and constructing the corresponding automaton parsers to perform language recognition and code generation. The system is designed to assist in creating parsers used for conventional compiler implementation and special application compilers. This section illustrates the steps that are typical for using the program to create a parser.

1.1 Starting the Program

The program executable is named compiler. The program can be invoked by any means available in the operating system desktop including command execution in a terminal window.

A parser specification file is opened using a dialog window accessed from the File submenu: File -> Open activates
1.2 Specifying Grammars

The grammar input mainviews for an open parser specification file are shown above in illustration 3. The mainviews in the illustration can be activated by the View submenu:

View -> GrammarView activates the mainview that contains the subviews for input of symbols and precedences. The subviews are the terminal, error token, nonterminal, implicit, and precedence subviews. An expanded view of the mainview is shown in illustration 4 above.

View -> RuleView activates the mainview that contains the subview for input of grammar rules. The subview is the rule subview.

Each of the subviews for grammar input are organized as lists which show an index and data related to the grammar element type. A quick guide to entering and editing data in the grammar subviews follows.

1) [insert] – insert new element after selected element
2) [shift-insert] – insert new element before selected element
3) [enter] – edit selected element
4) [shift-up_arrow] – move selected element up one position
5) [shift-down_arrow] – move selected element down one position
6) [shift-delete] – delete selected element

These editing functions can also be accessed for each grammar subview by the Grammar submenu. Operations that insert or edit data will show a dialog window specific for the grammar element type.

1.2.1 Terminals

Terminal symbols are character strings recognized by a regular expression parser. The parser driver accepts a character input stream and partitions the stream into consecutive substrings corresponding to matched regular expressions. The longest possible string is matched and the regular expression with the least index is matched for strings with multiple regular expression matches.
The terminal edit dialog has the following inputs:

1) **Type** – TOKEN type terminals are processed and input into a subsequent parser stage, FILTER type terminals are trapped and processed but are not input into a subsequent parser stage. The TOKEN type terminals are used in the formation of grammar rules and precedences, the FILTER type terminals are not.

2) **Driver** – REGEXP terminals are intended to be processed for data content by terminal source code, LITERAL terminals are not.

3) **Name** – the name of the terminal as it appears in the formation of rules and precedences.

4) **Regexp** – the regular expression that matches the terminal.

### 1.2.2 Error Tokens

Error tokens are token input symbols that are inserted into the parser input by error code that is invoked upon the occurrence of a syntax error.

The error token edit dialog has the following inputs:

1) **Name** – the name of the error token as it appears in the formation of rules and precedences.

2) **Synchronization Length** – the number of input tokens that must be accepted without error after this error token is inserted into the parser input for the parser driver to become synchronized with the parser input.

### 1.2.3 Nonterminals

Nonterminals are symbols that appear in the left and right sides of grammar rules for constructing a recursive definition of a formal language.

The nonterminal edit dialog has the following inputs:

1) **Name** – the name of the nonterminal as it appears in the formation of rules.

2) **Start Symbol** – indicates the symbol is the language grammar start symbol.
1.2.4 Rules

Rules are formed from token terminals, error tokens, and nonterminals. The format of rules for a context free language is a left side consisting of a single nonterminal and a right side consisting of zero or more token terminals, error tokens, and nonterminals.

The rule edit dialog has the following inputs:

1) **Rule** – line edit input of grammar rule consisting of nonterminal symbol followed by a \( \rightarrow \) string followed by zero or more terminals, error tokens, and nonterminals.

2) **Add** – append symbol selected in the multiple column list view to the contents of the rule line edit control.

3) **Delete** – delete rightmost space delimited substring from the contents of the rule line edit control.

4) **Unit** – activate the unit rule elimination flag dialog.

The unit rule elimination flag setting applies to unit rules, which are rules with one symbol on the right side, and marks the rule for subsequent use in a parser optimization calculation.
1.2.5 Precedences

Precedence specifications are used to establish default resolution of conflicts. All of the symbols and rules in a single entry have the same precedence level. Precedence level ordering matches the entry indexes so that the entry with the highest index has the highest precedence.

The precedence edit dialog has the following inputs:

1) **Associativity** – Left associativity resolves parsing conflicts with a reduce action, Right associativity resolves parsing conflicts with a shift action, Non Associative resolves parsing conflicts by maintaining a conflict action entry.

2) **Entries** – line edit input of terminals, error tokens, and rule indexes that are of the same precedence and associativity.

3) **Add** - append symbol selected in the multiple column list view to the contents of the entries line edit control.

4) **Delete** - delete rightmost space delimited substring from the contents of the entries line edit control.

1.2.6 Start Context

The modify start context dialog is activated by the **start context** control in the **terminal** subview. The dialog contains a line edit control for input of a start context assignment that initializes the context of the regular expression automaton parser.
1.3 Calculating Automatons

The parser calculation dialog controls the fundamental parser automaton calculation operations. Parser -> Calculate submenu item activates the dialog.

The calculations invoked by this dialog are performed after the formal language grammar elements have been input. The multiline edit control shows the status of calculations performed. The dialog has the following inputs:

1) **Calculate Lexical** – indicates the regular expression parser stage should be calculated.

2) **Input Range** – determines the value range of input characters processed by the regular expression parser, from 0 to the value selected minus 1.

3) **Lexical Type** – selects the regular expression driver type. SingleChar is used to develop regular expression syntax parsers themselves, RegExp for all other applications.

4) **Calculate Parser** – indicates the deterministic context free parser stage should be calculated.

5) **Parser Type** – selects the parser automaton calculation to perform.
Illustration 12: Conflict Resolution and Code Mainview

Compiler – a Parser Generator Program

status

Entry 0
State 16
Symbol +
Action Reduce By Rule 2
Default Action Default All
Clear Code Clear All

action

Shift +
* Rule 2 expr -> expr + expr
Conflict Entry

conflict

index entry actions resolve
0 State = 16 Symbol = 4, + Shift, 1 Reduce Reduce By Rule 2
1 State = 16 Symbol = 5, - Shift, 1 Reduce Reduce By Rule 2
2 State = 16 Symbol = 6, * Shift, 1 Reduce Shift *
3 State = 16 Symbol = 7, / Shift, 1 Reduce Shift /
4 State = 17 Symbol = 4, + Shift, 1 Reduce Reduce By Rule 3
5 State = 17 Symbol = 5, - Shift, 1 Reduce Reduce By Rule 3
6 State = 17 Symbol = 6, * Shift, 1 Reduce Shift *
7 State = 17 Symbol = 7, / Shift, 1 Reduce Shift /
8 State = 18 Symbol = 4, + Shift, 1 Reduce Reduce By Rule 4

conflict code

// conflict 0 code

current file: precedence.grammar options: precedence.option
1.4 Resolving Parsing Conflicts

The conflict resolution and code mainview is shown above in illustration 12.

View -> ConflictView submenu item activates the mainview. The mainview contains the subviews for resolving parsing conflicts and input of conflict section code. The subviews are the `conflict`, `action`, `status`, and `conflict code` subviews. The `conflict` subview selects the conflict for modification. The `action` subview shows the set of possible actions for the selected conflict. The parsing conflict is resolved to the action selected in the `action` subview. The action prefixed by a * indicates the action is the one selected by default rules and precedence specifiers. The `status` subview shows automaton machine information for a conflict action and has controls for reset of actions to default * actions and for clearing conflict code sections. Conflict code is executed by the parser driver only when the conflict action is resolved to Conflict Entry.

1.5 Calculating Essential Error Entries

The essential error entry calculation dialog is used to control the calculation to determine which parsing table error entries are accessible and how the entries are used to invoke error processing code.

Parser -> Error Entries submenu item activates the dialog.

The dialog has the following inputs:

1) **Essential Error Entry Calculation** – selects the type of calculation; No Calculation is used for simple error entry numbering where all error entries are considered essential, Approximate for an efficient determination of a superset of the essential entries, Exact for exact determination of the essential entries.
2) **Error Function Numbering** – determines the numbering of parser table error entries; One Number sets all entries to a single number, Number Per State sets error entries to the negative integer values assigned to the automaton machine states in which the errors occur, Number Per Entry sets error entries to consecutive decreasing negative integers.

3) **Error Function Aggregation** – determines how error entry indexes are mapped to error processing functions; One Function maps all error index values to one error function, Function Per State maps all error index values for an automaton machine state to a single function with a separate function for each machine state, Function Per Entry maps each error index value to an individual error function.

The dialog has a multiline edit control showing the status of calculations performed.

### 1.6 Calculating Unit Rule Eliminations

The unit rule elimination calculation dialog is used to calculate parser driver tables that eliminate unnecessary rule reduction actions by unit rules.

Parser -> Unit Rules submenu item activates the dialog.

The list control at the top left shows the rules that have been marked for potential elimination. Controls at the right are used to order the rules to optimize the unit rule elimination calculation result. The control at the bottom shows the status of the calculations.

### 1.7 Simulating Parsers

The simulation mainview is shown below in illustrations 15 and 16.

View -> SimulateView submenu item activates the mainview. The mainview contains the subviews for simulating the operation of automaton parsers. The subviews are the control, input, and parse subviews. The control and input subview operations can be accessed by the Simulate -> Control and the Simulate -> Input submenus.
Illustration 15: Parser Simulate Mainview - Regular Expression Automaton

Compiler – a Parser Generator Program

1.23 + 2.34 * (3.45 - 4.56) /
/* comment */ 5.67 -
- 2.0 * 3.0 // comment
Illustration 16: Parser Simulate Mainview - Deterministic Context Free Automaton

input: file - /tmp/simulate/precedence.in

1.23 + 2.34 * (3.45 - 4.56) /
/* comment */ 5.67 -
- 2.0 * 3.0 // comment
A simulation is configured by inputs in the control subview and by the simulator options dialog. The control subview has the following inputs:

1) **Timer** – specifies the frequency of parser actions in milliseconds.
2) **Expand** – expands a parse tree to the depth specified by the accompanying input control.
3) **Options** – activates the simulator options dialog.
4) **Reset** – resets parsers, input stream, and parse tree display.
5) **Next** – performs next parser action.
6) **Run** – performs continuous sequence of parsing actions at the timer frequency until input stream is exhausted or until unrecoverable error condition arises.

The control subview also displays current automaton machine state and parsing action information.

The simulator options dialog has the following inputs:

1) **Simulate** – specifies the parser type to simulate; Lexical requires the regular expression automaton calculation be performed, Parser requires both the regular expression and the deterministic context free automaton calculations be performed.
2) **Discard Lexical Error** – enables lexical errors from the regular expression automaton parser to be discarded to allow a parser simulation to continue.
3) **With Unit Rule Eliminations** – parser is configured to use the driver tables optimized for elimination of unit rules; the unit rule elimination calculation must be performed.

The *trap symbol* check list selects symbols for halting the simulation when executing continuous parsing actions. When a parsing action occurs that involves a symbol which is checked then the continuous execution of the simulation will halt.

The *input* subview contains the character data that is input to the regular expression automaton parser. Each terminal is highlighted as it is recognized during parsing simulation. The character data can be modified through *Simulate* -> *Input* submenu operations or directly edited.

### 1.7.1 Regular Expression Parser Simulation

A regular expression automaton parser simulation is shown in illustration 15. The *parse* subview shows the terminals recognized in sequence by the parser and the automaton machine information for each action.

### 1.7.2 Deterministic Context Free Parser Simulation

A deterministic context free parser simulation is shown in illustration 16. The *parse* subview shows the terminal tokens, error tokens, and nonterminals in a tree structure representation of the parse tree of the input. The parse tree has been expanded to full depth using the expand control.

### 1.8 Specifying Code Sections

The source code that implements code generation for a parser is contained in code sections that are invoked by a parser driver in response to terminal recognition, rule reduction, and error condition events. Each type of code section is handled by a mainview that contains subviews for selecting and modifying the code for each section entry.

#### 1.8.1 Terminal Code

The *View* -> *TerminalCodeView* submenu item selects the mainview that contains the *terminal select* and *terminal code* subviews. The selected entry in the *terminal select* subview is modified in the *terminal code* subview. The code for the entry is invoked by the parser driver when the corresponding terminal is recognized by the regular expression automaton parser and the terminal driver specifier is `REGEXP`. The mainview is shown in illustration 18 below.

#### 1.8.2 Rule Code

The *View* -> *RuleCodeView* submenu item selects the mainview that contains the *rule select* and *rule code* subviews. The selected entry in the *rule select* subview is modified in the *rule code* subview. The code for the entry is invoked by the parser driver when the corresponding rule reduction action is performed by the deterministic context free automaton parser. The mainview is shown in illustration 19 below.
// terminal 0 code
rule select

<table>
<thead>
<tr>
<th>index</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>expr -&gt; ( expr )</td>
</tr>
<tr>
<td>1</td>
<td>expr -&gt; ( error1 )</td>
</tr>
<tr>
<td>2</td>
<td>expr -&gt; expr + expr</td>
</tr>
<tr>
<td>3</td>
<td>expr -&gt; expr - expr</td>
</tr>
<tr>
<td>4</td>
<td>expr -&gt; expr * expr</td>
</tr>
<tr>
<td>5</td>
<td>expr -&gt; expr / expr</td>
</tr>
<tr>
<td>6</td>
<td>expr -&gt; - expr</td>
</tr>
<tr>
<td>7</td>
<td>* expr -&gt; variable</td>
</tr>
<tr>
<td>8</td>
<td>* expr -&gt; constant</td>
</tr>
</tbody>
</table>

rule code

// rule 0 code
1.8.3 Error Code

The View -> ErrorView submenu item selects the mainview that contains the error and error code subviews. The selected entry in the error subview is modified in the error code subview. The code for the entry is invoked by the parser driver when an error with the corresponding index is encountered. The mainview is shown in illustration 21 below.

The simulator error token dialog is used to specify an error token that will be inserted into the parser input token stream when an error corresponding to the entry index occurs. The edit token control activates the dialog.

![Simulator Error Token Dialog](Illustration 20  Simulator Error Token Dialog)

1.8.4 Conflict Code

The View -> ConflictView submenu item selects the mainview that contains the conflict and conflict code subviews. The selected entry in the conflict subview is modified in the conflict code subview. The code for the entry is invoked by the parser driver when a conflict entry with the corresponding index is encountered. The mainview is shown in illustration 12 above.

1.8.5 Class Code

See section 2.3.6 below.

1.8.6 File Source Code

The file source code mainview is shown in illustration 22 below. The View -> SourceView submenu item selects the mainview which is configured for two source code subviews. The subviews are used for general source code manipulation and are controlled through the Code -> Source submenu.
Illustration 21  Error Code Mainview
lexical->Initialize();
parser->Initialize();
parser->SetLexical(lexical);

parse_tree = new SttyTreeNode(0, SttyTreeNode::NonTerminal, "ASTART");

CParseDriver::StatusCode sc = parser->ParseStream();

int lexical_error = sc & CParseDriver::LexicalError;
int parser_error = sc & CParseDriver::ParserError;

if (lexical_error)
    cout << "LEXICAL ERROR" << endl;

#ifdef PRECEDENCEPARSERDATA_H
#define PRECEDENCEPARSERDATA_H

// parser data tables for CParseDriver

const int Precedence_Terminal_Count_E = 10;
const int Precedence_Symbol_Count_A = 12;
const int Precedence_Rule_Count_A = 10;
const int Precedence_Parser_Empty_Symbol_Count = 9;
const int Precedence_Shift_Action = 10;
const int Precedence_Accept_Action = 11;
const int Precedence_Conflict_Entry = 12;
const int Precedence_State_Count = 34;
#endif
1.9 Generating Parser Code

The generate code dialog controls the configuration and generation of parser source code. Parser -> Generate Code submenu item activates the dialog.

Illustration 23  Generate Code Dialog

The generate code dialog has the following inputs:

1) **Generate Lexical Code** – selects generation of regular expression automaton parser source code.

2) **Generate Parser Code** – selects generation of deterministic context free automaton parser source code.

3) **Without/With Eliminations** – selects parser tables without or with unit rule elimination optimization.

4) **Code Directory** – target directory that contains generated source code files.

5) **Lexical Base Name** – prefix used in creating identifiers for regular expression automaton parser source code.

6) **Parser Base Name** – prefix used in creating identifiers for deterministic context free automaton parser source code.
7) **TTY Code** – selects generation of source code set that can be built to produce a test application that accepts input on standard input and output on standard output in the form of a character based parse tree.

8) **Regexp Table** – selects generation of base regular expression match table for use in parser driver development.

The dialog contains a multiline edit control which displays the files generated.

## 2 User Interface Components

The user interface of the CPG application is based on the CPG application framework. The framework provides constructs for implementation of main application windows, menus, mainviews, subviews, and interface attributes and configuration. The default mainviews are built in to an interface configuration internal to the CPG application. The mainviews and their subview configuration and attributes can be arbitrarily modified by the option edit functions accessed through the View -> Options -> Edit Option Data submenu item.

### 2.1 Default Mainviews

The mainviews for an open parser specification file are accessed through the View submenu and the submenu items for a file are grouped contiguously. A mainview with a submenu item that is prefixed with an asterisk is currently active and will be involved in a window tiling operation. The default mainviews are:

1) **GrammarView** – contains the terminal, error token, nonterminal, implicit, and precedence subviews. Coordinates the subviews for symbols plus the precedences.

2) **RuleView** - contains the rule subview for rule input.

3) **ConflictView** – contains the conflict, action, status, and conflict code subviews. Coordinates the subviews for conflict resolution and code.

4) **ErrorView** – contains the error and error code subviews. Coordinates the subviews for error code and simulator error tokens.

5) **TerminalCodeView** – contains the terminal select and terminal code subviews. Coordinates the subviews for terminal code.

6) **RuleCodeView** – contains the rule select and rule code subviews. Coordinates the subviews for rule code.

7) **MachineView** – contains the machine subview for automaton machine information display.

8) **SourceView** – contains the source code subview for file source editing.

9) **SimulateView** – contains the control, input, and parse subviews. Coordinates the subviews for automaton parser simulation.
2.2 Edit Window Operations

All subview multiline edit windows for editing source code have the following keyboard operations:

1) \([\text{Left Arrow}]\) – move cursor one character left
2) \([\text{Right Arrow}]\) – move cursor one character right
3) \([\text{Up Arrow}]\) – move cursor one line up
4) \([\text{Down Arrow}]\) – move cursor one line down
5) \([\text{Page Up}]\) – move cursor one page up
6) \([\text{Page Down}]\) – move cursor one page down
7) \([\text{Backspace}]\) – delete character to left of cursor
8) \([\text{Home}]\) – move cursor to beginning of line
9) \([\text{End}]\) – move cursor to end of line
10) \([\text{Delete}]\) – delete character to right of cursor
11) \([\text{Shift}][\text{Left Arrow}]\) – mark text one character left
12) \([\text{Shift}][\text{Right Arrow}]\) – mark text one character right
13) \([\text{Control}-\text{A}]\) – move cursor to beginning of line
14) \([\text{Control}-\text{B}]\) – move cursor one character left
15) \([\text{Control}-\text{C}]\) – copy marked text to clipboard
16) \([\text{Control}-\text{D}]\) – delete character to right of cursor
17) \([\text{Control}-\text{E}]\) – move cursor to end of line
18) \([\text{Control}-\text{F}]\) – move cursor one character right
19) \([\text{Control}-\text{G}]\) – activate find and goto string dialog
20) \([\text{Control}-\text{H}]\) – delete character to left of cursor
21) \([\text{Control}-\text{K}]\) – delete to end of line
22) \([\text{Control}-\text{N}]\) – move cursor one line down
23) \([\text{Control}-\text{P}]\) – move cursor one line up
24) \([\text{Control}-\text{V}]\) – paste clipboard text
25) \([\text{Control}-\text{X}]\) – cut and move marked text to clipboard
26) \([\text{Control}-\text{Y}]\) – redo last operation
27) \([\text{Control}-\text{Z}]\) – undo last operation
28) \([\text{Control}-\text{Left Arrow}]\) – move cursor one word left
29) \([\text{Control}-\text{Right Arrow}]\) – move cursor one word right
30) [Control-Home] – move cursor to beginning of text
31) [Control-End] – move cursor to end of text

Those multiline edit windows that have open and save functions have the following keyboard operations:
1) [Control-O] – activate open file dialog
2) [Control-S] – activate save file dialog

2.3 Subviews

2.3.1 Grammar Input

The grammar input subviews are organized as list views with each element of a list specifying the information for a single grammar element. The first column of the list element data for all types is an index value of the element storage in the parser specification file data representation.

2.3.1.1 Terminal

The terminal subview contains the controls for input and modification of the grammar specification of a regular expression automaton parser. Terminals are specified in a list view control and are matched for the longest input character string for the terminal with the least index.

<table>
<thead>
<tr>
<th>Button - start context, with a line edit display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
</tr>
<tr>
<td>Menu</td>
</tr>
<tr>
<td>Dialog</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>List view with six columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>symbol</td>
</tr>
<tr>
<td>type</td>
</tr>
<tr>
<td>driver</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>regexp</td>
</tr>
</tbody>
</table>

user interface interaction
### Menu, Keyboard, Mouse

| Menu: Grammar -> Terminal -> Insert After  
| Keyboard: [Insert]  
| Insert new terminal element after selected element.  |
| Menu: Grammar -> Terminal -> Insert Before  
| Keyboard: [Shift][Insert]  
| Insert new terminal element before selected element.  |
| Menu: Grammar -> Terminal -> Edit  
| Keyboard: [Enter]  Mouse: double click  
| Edit selected element.  |
| Menu: Grammar -> Terminal -> Delete  
| Keyboard: [Shift][Delete]  
| Delete selected element.  |
| Menu: Grammar -> Terminal -> Move Up  
| Keyboard: [Shift][Up Arrow]  
| Move selected element up one position.  |
| Menu: Grammar -> Terminal -> Move Down  
| Keyboard: [Shift][Down Arrow]  
| Move selected element down one position.  |

#### Dialog

| Terminal dialog activated by Insert After, Insert Before, and Edit actions. |

#### Subview

| error token | Updated on terminal symbol change. |
| nonterminal | Updated on terminal symbol change. |
| implicit | Updated on terminal symbol change. |
| rule | Updated on terminal symbol or name change. |
| precedence | Updated on terminal symbol or name change. |
| terminal select | Updated on terminal addition, deletion, or change. |

### 2.3.1.2 Error Token

The **error token** subview contains a list view for input and modification of error tokens which are inserted into the input stream of a deterministic context free automaton parser on the occurrence of a syntax error. The insertion of an error token is accomplished by statements in an error code section that is invoked by the parser driver due to a syntax error.

<table>
<thead>
<tr>
<th>List view with four columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>symbol</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>sync</td>
</tr>
</tbody>
</table>
### user interface interaction

| Menu, Keyboard, Mouse | Menu: Grammar -> Error Token -> Insert After  
Keyboard: [Insert]  
Insert new error token element after selected element. |
|-----------------------|---------------------------------------------------------------------------------------------------|
|                       | Menu: Grammar -> Error Token -> Insert Before  
Keyboard: [Shift][Insert]  
Insert new error token element before selected element. |
|                       | Menu: Grammar -> Error Token -> Edit  
Keyboard: [Enter]  
Mouse: double click  
Edit selected element. |
|                       | Menu: Grammar -> Error Token -> Delete  
Keyboard: [Shift][Delete]  
Delete selected element. |
|                       | Menu: Grammar -> Error Token -> Move Up  
Keyboard: [Shift][Up Arrow]  
Move selected element up one position. |
|                       | Menu: Grammar -> Error Token -> Move Down  
Keyboard: [Shift][Down Arrow]  
Move selected element down one position. |

#### Dialog

**Error Token** dialog activated by Insert After, Insert Before, and Edit actions.

#### Subview

<table>
<thead>
<tr>
<th>Subview</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nonterminal</strong></td>
<td>Updated on error token symbol change.</td>
</tr>
<tr>
<td><strong>implicit</strong></td>
<td>Updated on error token symbol change.</td>
</tr>
<tr>
<td><strong>rule</strong></td>
<td>Updated on error token symbol or name change.</td>
</tr>
<tr>
<td><strong>precedence</strong></td>
<td>Updated on error token symbol or name change.</td>
</tr>
</tbody>
</table>

### 2.3.1.3 Nonterminal

The **nonterminal** subview contains a list view for input and modification of nonterminals.

#### List view with four columns

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>index</strong></td>
<td>Grammar element data storage index.</td>
</tr>
<tr>
<td><strong>symbol</strong></td>
<td>Index of nonterminal used by a subsequent parser stage.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>Name of nonterminal as used in the formation of rules.</td>
</tr>
<tr>
<td><strong>start</strong></td>
<td>Asterisk indicates symbol is language grammar start symbol.</td>
</tr>
</tbody>
</table>
Menu, Keyboard, Mouse

Menu: Grammar -> NonTerminal -> Insert After
Keyboard: [Insert]
Insert new nonterminal element after selected element.

Menu: Grammar -> NonTerminal -> Insert Before
Keyboard: [Shift][Insert]
Insert new nonterminal element before selected element.

Menu: Grammar -> NonTerminal -> Edit
Keyboard: [Enter] Mouse: double click
Edit selected element.

Menu: Grammar -> NonTerminal -> Delete
Keyboard: [Shift][Delete]
Delete selected element.

Menu: Grammar -> NonTerminal -> Move Up
Keyboard: [Shift][Up Arrow]
Move selected element up one position.

Menu: Grammar -> NonTerminal -> Move Down
Keyboard: [Shift][Down Arrow]
Move selected element down one position.

Dialog
NonTerminal dialog activated by Insert After, Insert Before, and Edit actions.

Subview
implicit Updated on nonterminal symbol change.
rule Updated on nonterminal symbol or name change.

2.3.1.4 Implicit

The implicit subview contains a display label showing the grammar elements that are implicitly added to a grammar specification in order to introduce the end of file symbol and simplify the determination of an accept action for a deterministic context free automaton parser.

<table>
<thead>
<tr>
<th>EOF</th>
<th>End Of File symbol inserted at the end of a character input stream.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTART</td>
<td>Augmented start symbol.</td>
</tr>
<tr>
<td>ASTART -&gt; (start symbol) EOF</td>
<td>Augmented start rule.</td>
</tr>
</tbody>
</table>

2.3.1.5 Rule

The rule subview contains a list view for input and modification of rules.

<table>
<thead>
<tr>
<th>List view with four columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>unit</td>
</tr>
<tr>
<td>left</td>
</tr>
</tbody>
</table>
**Right**
Right side of rule consisting of zero or more token terminals, error tokens, and nonterminals.

user interface interaction

| Menu, Keyboard, Mouse | Menu: Grammar -> Rule -> Insert After  
Keyboard: [Insert]  
Insert new rule element after selected element. |
| --- | --- |
| | Menu: Grammar -> Rule -> Insert Before  
Keyboard: [Shift][Insert]  
Insert new rule element before selected element. |
| | Menu: Grammar -> Rule -> Edit  
Keyboard: [Enter]  
Mouse: double click  
Edit selected element. |
| | Menu: Grammar -> Rule -> Delete  
Keyboard: [Shift][Delete]  
Delete selected element. |
| | Menu: Grammar -> Rule -> Move Up  
Keyboard: [Shift][Up Arrow]  
Move selected element up one position. |
| | Menu: Grammar -> Rule -> Move Down  
Keyboard: [Shift][Down Arrow]  
Move selected element down one position. |
| Dialog | Rule dialog activated by Insert After, Insert Before, and Edit actions. |
| Subview | precedence | Updated on rule addition or deletion. |
| | rule select | Updated on rule addition, deletion, or change. |

### 2.3.1.6 Precedence

The **precedence** subview contains a list view for input and modification of precedences. The precedences are ordered by their index so that the entries with the highest index have the highest precedence. The precedences determine a default conflict resolution applied after the deterministic context free automaton calculations are performed. The default conflict resolution rules are:

1) The precedence level assigned to a rule is the same as the last token terminal or error token appearing in the rule which has a precedence level, or the level assigned to the rule as an entry in a precedence specification element, or none.

2) If conflict is a shift, 1 reduce conflict and the rule and lookahead terminal token have an assigned precedence then if the precedences differ the conflict is resolved to the action with the highest precedence, which is reduce for a rule and shift for a terminal token, and if the precedences are the same then associativity is used.

3) If conflict is a shift, 1 reduce conflict and the rule or the lookahead terminal token do not have assigned precedences then the conflict is resolved to shift.
4) If conflict is a multiple reduce conflict then if shift is a valid action the conflict is resolved to
a shift, and if shift is not a valid action the conflict is resolved to the valid rule reduce action
with the lowest index.

<table>
<thead>
<tr>
<th>List view with three columns</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>index</strong></td>
<td>Grammar element data storage index.</td>
</tr>
<tr>
<td><strong>associativity</strong></td>
<td>Associativity of element entries. For entries of the same precedence level, LEFT associativity resolves conflicts to reduce action, RIGHT associativity resolves conflicts to shift action, NONASSOCIATIVE causes conflicts to remain as conflict entries.</td>
</tr>
<tr>
<td><strong>entries</strong></td>
<td>TOKEN terminals, error tokens, and rule indexes specified as having the same precedence level.</td>
</tr>
</tbody>
</table>

**user interface interaction**

| Menu, Keyboard, Mouse | Menu: Grammar -> Precedence -> Insert After  
Keyboard: [Insert]  
Insert new precedence element after selected element. |
|------------------------|--------------------------------------------------|
|                        | Menu: Grammar -> Precedence -> Insert Before  
Keyboard: [Shift][Insert]  
Insert new precedence element before selected element. |
|                        | Menu: Grammar -> Precedence -> Edit  
Keyboard: [Enter]  
Mouse: double click  
Edit selected element. |
|                        | Menu: Grammar -> Precedence -> Delete  
Keyboard: [Shift][Delete]  
Delete selected element. |
|                        | Menu: Grammar -> Precedence -> Move Up  
Keyboard: [Shift][Up Arrow]  
Move selected element up one position. |
|                        | Menu: Grammar -> Precedence -> Move Down  
Keyboard: [Shift][Down Arrow]  
Move selected element down one position. |

**Dialog**

**Precedence** dialog activated by Insert After, Insert Before, and Edit actions.

### 2.3.2 Conflict Resolution and Source Code

The conflict resolution and source code subviews provide complete information and control of parsing conflicts arising in automaton calculations.

#### 2.3.2.1 Conflict Select

The **conflict** select subview contains a list view control for selecting a conflict. The selected conflict is modified by the **action** subview to resolve the conflict and by the **conflict code** subview to input source code that is invoked by the parser driver when the conflict is
maintained as a conflict entry.

<table>
<thead>
<tr>
<th>column</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Grammar element data storage index.</td>
</tr>
<tr>
<td>entry</td>
<td>Description of action table entry where conflict occurs including automaton machine state and lookahead token symbol.</td>
</tr>
<tr>
<td>actions</td>
<td>Description of type and number of actions in conflict.</td>
</tr>
<tr>
<td>resolve</td>
<td>The action to which the conflict is resolved or the conflict entry.</td>
</tr>
</tbody>
</table>

**2.3.2.2 Action**

The action subview contains a list control showing the actions in a parsing conflict entry plus a special value designated Conflict Entry. A parsing conflict is resolved to one of its actions or to the Conflict Entry value by selecting the corresponding item in the list control. If the conflict is resolved to Conflict Entry then the source code for the conflict is invoked by the parser driver when the parsing conflict action table entry occurs.

<table>
<thead>
<tr>
<th>Subview</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Updated on conflict selection change.</td>
</tr>
<tr>
<td>status</td>
<td>Updated on conflict selection change.</td>
</tr>
<tr>
<td>conflict code</td>
<td>Updated on conflict selection change.</td>
</tr>
</tbody>
</table>

**2.3.2.3 Status**

The status subview contains controls for displaying a parsing conflict entry and its resolve status plus controls for individual and group modification of the conflict entries.

<table>
<thead>
<tr>
<th>Display fields</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>Grammar element data storage index.</td>
</tr>
<tr>
<td>State</td>
<td>Automaton machine state of conflict.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Lookahead token symbol of conflict.</td>
</tr>
<tr>
<td>Action</td>
<td>Current parsing conflict resolve action.</td>
</tr>
</tbody>
</table>

**Button - Default Action**

<table>
<thead>
<tr>
<th>Operation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset parsing conflict action to default resolution action.</td>
<td></td>
</tr>
</tbody>
</table>
### 2.3.2.4 Conflict Code

The **conflict code** subview contains a multilne edit control for entering and editing the source code associated with a conflict entry. The code for a conflict entry is invoked only if the conflict is resolved to the special value **Conflict Entry**. In this case the code is invoked by the parser driver when the parsing conflict action table entry occurs.

#### Multiline edit control

<table>
<thead>
<tr>
<th>Menu, Keyboard</th>
<th>User interface interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu: Code -&gt; Conflict -&gt; Find String</td>
<td>Activate <strong>Find String</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Conflict -&gt; Undo Edit</td>
<td>Undo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Conflict -&gt; Redo Edit</td>
<td>Redo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Conflict -&gt; Clear</td>
<td>Clear multiline edit control.</td>
</tr>
<tr>
<td>Keyboard: [F1]</td>
<td>Toggle display of conflict information in subview title bar.</td>
</tr>
</tbody>
</table>
2.3.3 Error Source Code

The error source code subviews are used to input source code for parser error functions. The code for an error function is invoked by the parser driver when the error entry that is mapped to the function occurs. The functions for which code can be entered are determined by the current error calculation and function aggregation configuration.

2.3.3.1 Error Select

The error select subview contains a list view control which selects an error function for source code input. The list view control also sets error tokens for the functions which are used during parser simulation. Other controls are available for clearing error function code.

<table>
<thead>
<tr>
<th>Button - Clear Code</th>
<th>Operation</th>
<th>Clear source code of selected error function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu</td>
<td>Menu: Code -&gt; Error -&gt; Clear</td>
<td></td>
</tr>
<tr>
<td>Subview</td>
<td>error code</td>
<td>Clear multiline edit control.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Button - Clear All</th>
<th>Operation</th>
<th>Clear source code of all error functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subview</td>
<td>error code</td>
<td>Clear multiline edit control.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Button - Edit Token</th>
<th>Operation</th>
<th>Set simulator error token for selected error function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog</td>
<td>Simulator Error Token dialog activated by button.</td>
<td></td>
</tr>
</tbody>
</table>

List view with three columns

<table>
<thead>
<tr>
<th>column</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Grammar element data storage index.</td>
</tr>
<tr>
<td>function</td>
<td>Description of error function.</td>
</tr>
<tr>
<td>token</td>
<td>Error token used by parser driver during simulation when the syntax error entry occurs which is mapped to the error function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keyboard, Mouse</th>
<th>Keyboard: [Enter] Mouse: double click Edit simulator error token for selected error function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subview</td>
<td>error code</td>
</tr>
</tbody>
</table>

2.3.3.2 Error Code

The error code subview contains a multiline edit control for entering and editing the source
code of an error function. An error function is invoked by the parser driver when a syntax error entry which is mapped to the function occurs. In order for the parser driver to attempt a recovery from a syntax error the error function that handles the error must set two variables:

1) `error_token` – the parser symbol value for the error token to be inserted into the lookahead token input stream. The value is the integer value in the `symbol` column of the `error token` subview entry for the error token.

2) `synchronization_length` – the number of lookahead tokens the parser driver must accept after the error token in order for the parser driver to be resynchronized with the input stream. The value is the integer value in the `sync` column of the `error token` subview entry for the error token.

In order to facilitate the required source code several generated constants and parser driver base class functions are available.

1) `Error_Token_name` – constant equal to parser symbol value of the error token `name`.
2) `Error_Synchronization_name` – synchronization length of error token `name`.
3) `void SetErrorToken(int et)` – function which sets `error_token` variable.
4) `void SetSynchronizationLength(int sl)` – function which sets `synchronization_length` variable.

Thus, an error function which attempts to resynchronize the input stream by inserting error token `name` into the lookahead token stream should include the following code:

```cpp
SetErrorToken(Error_Token_name);
SetSynchronizationLength(Error_Synchronization_name);
```

An error function has the following signature:

```cpp
void DerivedClass::Error_index()
```

where `DerivedClass` is the derived context free automaton parser class and `index` is the error function index.

### Multiline edit control

<table>
<thead>
<tr>
<th>Menu, Keyboard</th>
<th>user interface interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu: Code -&gt; Error -&gt; Find String</td>
<td>Activate <strong>Find String</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Error -&gt; Undo Edit</td>
<td>Undo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Error -&gt; Redo Edit</td>
<td>Redo last edit operation.</td>
</tr>
<tr>
<td>Keyboard: [F1]</td>
<td>Toggle display of error function information in subview title bar.</td>
</tr>
</tbody>
</table>
2.3.4 Terminal Source Code

The terminal source code subviews are used to input source code for regular expression terminal functions. The code for a terminal function is invoked by the parser driver when the regular expression match and a shift action for the terminal occurs.

2.3.4.1 Terminal Select

The terminal select subview contains a list view control which selects a terminal function for source code input.

<table>
<thead>
<tr>
<th>List view with two columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>function</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Subview</td>
</tr>
</tbody>
</table>

2.3.4.2 Terminal Code

The terminal code subview contains a multiline edit control for entering and editing the source code of a terminal function.

<table>
<thead>
<tr>
<th>Multiline edit control</th>
</tr>
</thead>
<tbody>
<tr>
<td>user interface interaction</td>
</tr>
<tr>
<td>Menu, Keyboard</td>
</tr>
<tr>
<td>Menu: Code -&gt; Terminal -&gt; Find String</td>
</tr>
<tr>
<td>Activate <strong>Find String</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Terminal -&gt; Undo Edit</td>
</tr>
<tr>
<td>Undo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Terminal -&gt; Redo Edit</td>
</tr>
<tr>
<td>Redo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Terminal -&gt; Clear</td>
</tr>
<tr>
<td>Clear multiline edit control.</td>
</tr>
<tr>
<td>Keyboard: [F1]</td>
</tr>
<tr>
<td>Toggle display of terminal function information in subview title bar.</td>
</tr>
</tbody>
</table>

Terminal functions have two different types of signatures:

1) `void DerivedClass::Filter_symbol()` - for FILTER type terminals.
2) `void DerivedClass::Token_symbol(CStackElement *se)` - for TOKEN type terminals.

DerivedClass is the derived regular expression automaton parser class, for tokens `symbol` is the value of the symbol column in the terminal subview list view control, and for filters `symbol` is an integer value analogous to that for the tokens which results from a sequential numbering from zero of the filter terminals. CStackElement is the class used to implement a
parsing stack and to coordinate the operation of a regular expression automaton parser and a context free automaton parser. Terminal match strings for tokens and their data conversions are stored in the `CStackElement` passed by a pointer argument. Two base class functions can be called in terminal function code to access the matched terminal string:

1) `unsigned char *GetTerminalString()` – returns a pointer to the terminal string data. The array is null terminated.

2) `int GetTerminalLength()` - returns the length of the terminal string data array not including the null termination element.

### 2.3.5 Rule Source Code

The rule source code subviews are used to input source code for rule reduce functions. The code for a rule function is invoked by the parser driver when a reduce action is performed for the rule.

#### 2.3.5.1 Rule Select

The `rule select` subview contains a list view control which selects a rule function for source code input.

<table>
<thead>
<tr>
<th>List view with two columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>column</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>function</td>
</tr>
</tbody>
</table>

| Subview | rule code | Updated on rule function selection change. |

#### 2.3.5.2 Rule Code

The `rule code` subview contains a multiline edit control for entering and editing the source code of a rule function.

<table>
<thead>
<tr>
<th>Multiline edit control</th>
</tr>
</thead>
<tbody>
<tr>
<td>user interface interaction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Menu, Keyboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu: Code -&gt; Rule -&gt; Find String</td>
</tr>
<tr>
<td>Activate <strong>Find String</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Rule -&gt; Undo Edit</td>
</tr>
<tr>
<td>Undo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Rule -&gt; Redo Edit</td>
</tr>
<tr>
<td>Redo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Rule -&gt; Clear</td>
</tr>
<tr>
<td>Clear multiline edit control.</td>
</tr>
<tr>
<td>Keyboard: [F1]</td>
</tr>
<tr>
<td>Toggle display of rule function information in subview title bar.</td>
</tr>
</tbody>
</table>
A rule function has the following signature:

```c
int DerivedClass::Reduce_index(void *data)
```

where `DerivedClass` is the derived context free automaton parser class and `index` is the rule index. The argument `data` is used primarily for syntax parsing mode and is unused in most default mode parsing applications. The function should return -1 for a code generation error.

The parsing stack elements of type `CStackElement` are accessed through the pointer variable:

```c
CStackElement *stack_ptr
```

Successive stack elements that correspond to the right side symbols of a context free rule are

```c
stack_ptr[0], stack_ptr[1], ..., stack_ptr[m-1]
```

where `m` is the right rule length.

### 2.3.6 Class Source Code

The `class code` subview contains a multiline edit control for entering and editing the source code of derived parser driver classes. A list selection control is used to determine which code section is being edited. The regular expression automaton parser files are indicated with a prefix of `lbase` and the context free automaton parser files are indicated with a prefix of `pbase`.

<table>
<thead>
<tr>
<th>Dropdown list control</th>
<th>Code Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>code section location</td>
</tr>
<tr>
<td>Lexical Include</td>
<td>Derived regular expression automaton parser include file: <code>lbaseLexicalDriver.h</code>, after default includes, before class declaration.</td>
</tr>
<tr>
<td>Lexical Class</td>
<td>Derived regular expression automaton parser include file: <code>lbaseLexicalDriver.h</code>, inside derived class declaration with public access.</td>
</tr>
<tr>
<td>Lexical Source</td>
<td>Derived regular expression automaton parser source file: <code>lbaseLexicalDriver.cpp</code>, after includes, before constructor function.</td>
</tr>
<tr>
<td>Lexical Constructor</td>
<td>Derived regular expression automaton parser source file: <code>lbaseLexicalDriver.cpp</code>, at bottom of constructor function.</td>
</tr>
<tr>
<td>Lexical Destructor</td>
<td>Derived regular expression automaton parser source file: <code>lbaseLexicalDriver.cpp</code>, at bottom of destructor function.</td>
</tr>
<tr>
<td>Parser Include</td>
<td>Derived context free automaton parser include file: <code>pbaseParserDriver.h</code>, after default includes, before class declaration.</td>
</tr>
<tr>
<td>Parser Class</td>
<td>Derived context free automaton parser include file: <code>pbaseParserDriver.h</code>, inside derived class declaration with public access.</td>
</tr>
<tr>
<td>Parser Source</td>
<td>Derived context free automaton parser source file: <code>pbaseParserDriver.cpp</code>, after includes, before constructor function.</td>
</tr>
<tr>
<td>Parser Constructor</td>
<td>Derived context free automaton parser source file: <code>pbaseParserDriver.cpp</code>, at bottom of constructor function.</td>
</tr>
</tbody>
</table>
### Compiler – a Parser Generator Program

<table>
<thead>
<tr>
<th><strong>Parser Destructor</strong></th>
<th>Derived context free automaton parser source file: <code>pbaseParserDriver.cpp</code>, at bottom of destructor function.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack Element</strong></td>
<td>Inside CStackElement class declaration.</td>
</tr>
</tbody>
</table>

#### Multiline edit control

**User interface interaction**

<table>
<thead>
<tr>
<th><strong>Menu, Keyboard</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu: Code -&gt; Class -&gt; Find String</td>
<td>Activate <strong>Find String</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Class -&gt; Undo Edit</td>
<td>Undo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Class -&gt; Redo Edit</td>
<td>Redo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Class -&gt; Clear</td>
<td>Clear multiline edit control.</td>
</tr>
<tr>
<td>Keyboard: [F1]</td>
<td>Toggle display of code section information in subview title bar.</td>
</tr>
</tbody>
</table>

#### 2.3.7 File Source Code

The file **source code** subview is used for general source code editing operations. The subview can be used to open and save files for transferring code to and from other source code subviews.

<table>
<thead>
<tr>
<th><strong>Menu, Keyboard</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu: Code -&gt; Source -&gt; Open</td>
<td>Activate <strong>Open File</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Source -&gt; Save</td>
<td>Activate <strong>Save File</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Source -&gt; Find String</td>
<td>Activate <strong>Find String</strong> dialog.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Source -&gt; Undo Edit</td>
<td>Undo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Source -&gt; Redo Edit</td>
<td>Redo last edit operation.</td>
</tr>
<tr>
<td>Menu: Code -&gt; Source -&gt; Clear</td>
<td>Clear multiline edit control.</td>
</tr>
<tr>
<td>Keyboard: [F1]</td>
<td>Toggle display of source code file information in subview title bar.</td>
</tr>
</tbody>
</table>
2.3.8 Automaton Machine

The automaton *machine* subview is used to display automaton calculation results for the available machine types. A selection list control selects the type of information to display and a numeric input field selects the machine state.

<table>
<thead>
<tr>
<th>Dropdown list control - <em>Parser Data Type</em></th>
<th>Calculation Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>LR(0) Machine</em></td>
<td>Displays LR(0) machine state item cores and goto transitions on symbols.</td>
</tr>
<tr>
<td><em>LR(1) Machine</em></td>
<td></td>
</tr>
<tr>
<td><em>SLR(1) Lookahead</em></td>
<td></td>
</tr>
<tr>
<td><em>LALR(1) Lookahead</em></td>
<td>Displays reduce item cores and the associated lookahead token symbols.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Numeric input field - <em>Machine State</em></th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Selects machine state to display information for.</td>
</tr>
</tbody>
</table>

2.3.9 Automaton Simulate

The automaton simulate subviews are used to produce a dynamic graphical display of the operation of an automaton parser.

2.3.9.1 Control

The *control* subview contains input controls for initializing, running, and displaying the execution of an automaton parser.

<table>
<thead>
<tr>
<th>Display fields</th>
<th>State</th>
<th>Automaton machine state.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>Terminal</em></td>
<td><em>Current input lookahead token terminal.</em></td>
</tr>
<tr>
<td></td>
<td><em>Action</em></td>
<td>Current parser action.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Numeric input field - <em>Timer</em></th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets simulator parser action frequency in milliseconds.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Button - <em>Expand</em>, plus numeric input field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expand parse tree to depth indicated by numeric input field.</td>
</tr>
<tr>
<td></td>
<td>user interface interaction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Button - <em>Options</em></th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set simulator execution options.</td>
</tr>
<tr>
<td></td>
<td>user interface interaction</td>
</tr>
</tbody>
</table>
### Menu

**Menu**: Simulate -> Control -> Set Options

### Dialog

**Simulator Options** dialog activated by button.

### Button - Reset

**Operation**: Initialize automaton parsers, set input cursor to the beginning of data, and clear parse tree.

**User interface interaction**

- **Menu**: Menu: Simulate -> Control -> Reset Simulator

### Subview

**input**
- Unmark text and set cursor to beginning of data.

**parse**
- Clear parse tree and show initial stack configuration.

### Button - Next

**Operation**: Execute next automaton parser operation.

**User interface interaction**

- **Menu**: Menu: Simulate -> Control -> Next Operation

### Subview

**input**
- Advance cursor and mark text to show current lookahead token terminal.

**parse**
- For a shift action append token terminal to parse tree list, for a reduce action remove rule right side symbols from end of parse tree list, append rule left side nonterminal to parse tree list, add rule right side symbols to nonterminal as child nodes.

### Button - Run

**Operation**: Toggle execution of automaton parser until input exhausted or error.

**User interface interaction**

- **Menu**: Menu: Simulate -> Control -> Run Simulator

### Subview

**input**
- Advance cursor and mark text to show current lookahead token terminal.

**parse**
- For a shift action append token terminal to parse tree list, for a reduce action remove rule right side symbols from end of parse tree list, append rule left side nonterminal to parse tree list, add rule right side symbols to nonterminal as child nodes.

### 2.3.9.2 Input

The **input** subview contains a multiline edit control used to provide character stream input to a regular expression automaton parser. As terminal match strings are recognized the corresponding text in the edit control is highlighted. The edit control contents can be transferred to and from files.
### 2.3.9.3 Parse

The *parse* subview contains a list view control which displays the parsing stack of an automaton parser during operation and can be expanded to display a parse tree that results from parsing the contents of the *input* subview.

<table>
<thead>
<tr>
<th>column</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>Displays terminals and nonterminals organized as a tree that corresponds to the parse tree or the terminal recognition list produced by the simulation of the automaton parser drivers.</td>
</tr>
<tr>
<td>state</td>
<td>Machine state of automaton during action that produced the terminal or nonterminal.</td>
</tr>
<tr>
<td>action</td>
<td>Action of automaton during operation.</td>
</tr>
<tr>
<td>value</td>
<td>Value of match string for terminals.</td>
</tr>
</tbody>
</table>

### 2.4 Dialogs

#### 2.4.1 File

The *File* dialog is used for opening and saving files. When activated in save mode the file specified for saving to is checked for existence. If the file exists and the save operation is started then a continuation dialog is activated to allow canceling a file overwrite operation.
2.4.2 Grammar Edit

2.4.2.1 Start Context

The Start Context dialog is used to specify an assignment start context for the regular expression automaton. The assignment start context has the format:

<string_list> - where string_list is a list of character strings separated by commas.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Context</td>
<td>Line edit control for entering assignment start context.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm edit operation and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

2.4.2.2 Terminal

The Terminal dialog is used to input and edit regular expression terminal specifications. Regular expression syntax is based on the following special character rules and non-special characters:

Special Characters – must be preceded by a backslash '\\' character to have literal interpretation:

1) space character: '"'
2) ( start of subexpression grouping
3) ) end of subexpression grouping
4) * zero or more instances of preceding expression
5) + one or more instances of preceding expression
6) ? zero or one instance of preceding expression
7) | or-operator; match one of two expressions
8) \ escape character
9) " quotation of literal character string expressions
10) [ start of character class specification
11) ] end of character class specification
12) – separates two characters to form a character range
13) ^ complement of character class specification
14) . match all characters except newline
15) { start of repeat range
16) } end of repeat range
17) , delimiter character
18) $ reserved character
19) / start of trailing context
20) < start of activation/assignment list
21) > end of activation/assignment list

Non-special Characters – interpreted literally:
1) upper and lower case alphabetic characters: A–Z a–z
2) digit characters: 0–9
3) other non-special characters: !#$&‘;==@_`~

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Mutually exclusive selection of FILTER or TOKEN terminal type. FILTER terminals are trapped, TOKEN terminals are passed to subsequent parser stages.</td>
</tr>
<tr>
<td>Driver</td>
<td>Mutually exclusive selection of LITERAL or REGEXP driver processing. REGEXP terminals are processed for data content, LITERAL terminals are not.</td>
</tr>
<tr>
<td>Name</td>
<td>Line edit control for entering name of terminal as used in the formation of rules and precedences.</td>
</tr>
<tr>
<td>Regexp</td>
<td>Line edit control for entering regular expression match specification.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm edit operation and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

### 2.4.2.3 Error Token

The **Error Token** dialog is used to input and edit error token specifications.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Line edit control for entering name of error token as used in the formation of rules and precedences.</td>
</tr>
</tbody>
</table>
**2.4.2.4 Nonterminal**

The **Nonterminal** dialog is used to input and edit nonterminal specifications.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Line edit control for entering name of nonterminal as used in the formation of rules.</td>
</tr>
<tr>
<td>Start Symbol</td>
<td>Check box control that indicates symbol is language grammar start symbol.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm edit operation and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

**2.4.2.5 Rule**

The **Rule** dialog is used to input and edit rule specifications.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List selection control</td>
<td>List of all token terminals, error tokens, and nonterminals used in rule formation.</td>
</tr>
<tr>
<td>Rule</td>
<td>Line edit control for entering a rule. Rule syntax is: nonterminal (\rightarrow) symbol_list where symbol_list is sequence of zero or more token terminals, error tokens, and nonterminals.</td>
</tr>
<tr>
<td>Add</td>
<td>Button which appends symbol selected in list selection control to the contents of the Rule line edit control.</td>
</tr>
<tr>
<td>Delete</td>
<td>Button which deletes rightmost space delimited substring from the contents of the Rule line edit control.</td>
</tr>
<tr>
<td>Unit</td>
<td>Button which activates the <strong>Unit Rule Elimination</strong> dialog.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm edit operation and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

**2.4.2.6 Unit Rule Elimination**

The **Unit Rule Elimination** dialog is used to designate a rule for inclusion in unit rule elimination optimization.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Rule Elimination Flag</td>
<td>Check box control that indicates rule is to be included in unit rule elimination optimization.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm edit operation and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>
2.4.2.7 Precedence

The **Precedence** dialog is used to input and edit precedence specifications.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List selection</td>
<td>List of all token terminals, error tokens, and rule indexes used in precedence formation.</td>
</tr>
<tr>
<td>Associativity</td>
<td>Mutually exclusive selection of Left, Right, or Non Associative.</td>
</tr>
<tr>
<td>Entries</td>
<td>Line edit control for entering a precedence. Precedence syntax is a sequence of one or more token terminals, error tokens, and rule indexes.</td>
</tr>
<tr>
<td>Add</td>
<td>Button which appends symbol selected in list selection control to the contents of the Entries line edit control.</td>
</tr>
<tr>
<td>Delete</td>
<td>Button which deletes rightmost space delimited substring from the contents of the Entries line edit control.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm edit operation and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

2.4.3 Automaton Calculation

2.4.3.1 Calculate

The **Calculate** dialog controls and configures the basic automaton calculations. The dialog is activated by the Parser -> Calculate submenu item.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiline edit</td>
<td>Displays status of calculation results.</td>
</tr>
<tr>
<td>Calculate Lexical</td>
<td>Check box control that indicates regular expression automaton calculation will be performed.</td>
</tr>
<tr>
<td>Input Range</td>
<td>Mutually exclusive selection of input character range equal to 128 or 256.</td>
</tr>
<tr>
<td>Lexical Type</td>
<td>Mutually exclusive selection of SingleChar or Regexp regular expression automaton parser driver type.</td>
</tr>
<tr>
<td>Calculate Parser</td>
<td>Check box control that indicates context free automaton calculation will be performed.</td>
</tr>
<tr>
<td>Parser Type</td>
<td>Mutually exclusive selection of SLR(1), LALR(1), or LR (1) context free automaton type.</td>
</tr>
<tr>
<td>Calculate</td>
<td>Button to perform configured operations.</td>
</tr>
<tr>
<td>Close</td>
<td>Button to close dialog.</td>
</tr>
</tbody>
</table>

2.4.3.2 Error Entries

The **Error Entries** dialog controls and configures the calculation of essential error entries, the numbering of error entries, and the mapping and aggregation of error functions. The dialog is activated by the Parser -> Error Entries submenu item.
2.4.3.3 Unit Rules

The Unit Rules dialog controls and configures the calculation of the unit rule elimination optimization. The dialog is activated by the Parser -> Unit Rules submenu item.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List selection control</td>
<td>List of all unit rules that are marked for inclusion in the unit rule elimination optimization calculation.</td>
</tr>
<tr>
<td>Multiline edit control</td>
<td>Displays status of calculation results.</td>
</tr>
<tr>
<td>Up</td>
<td>Button move selected unit rule up one position.</td>
</tr>
<tr>
<td>Down</td>
<td>Button move selected unit rule down one position.</td>
</tr>
<tr>
<td>First</td>
<td>Button move selected rule to first position.</td>
</tr>
<tr>
<td>Last</td>
<td>Button move selected rule to last position.</td>
</tr>
<tr>
<td>Calculate</td>
<td>Button to perform configured operations.</td>
</tr>
<tr>
<td>Close</td>
<td>Button to close dialog.</td>
</tr>
</tbody>
</table>

2.4.4 Generate Code

The Generate Code dialog controls and configures the generation of parser driver and table code. The dialog is activated by the Parser -> Generate Code submenu item.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiline edit control</td>
<td>Displays status of calculation results.</td>
</tr>
<tr>
<td>Generate Lexical Code</td>
<td>Check box control that indicates regular expression automaton parser code will be generated. Disabled if regular expression automaton calculation has not been completed.</td>
</tr>
<tr>
<td>Generate Parser Code</td>
<td>Check box control that indicates context free automaton parser code will be generated. Disabled if context free automaton calculation has not been completed.</td>
</tr>
<tr>
<td>Parser Table Unit Rule Eliminations</td>
<td>Mutually exclusive selection of option to generate parser tables with or without unit rule elimination optimization. Option to generate tables with the optimization is disabled if the optimization calculation has not been completed.</td>
</tr>
</tbody>
</table>
### Code Directory
Line edit control for entering target directory to contain generated files.

### Lexical Base Name
Line edit control for entering prefix of source file names and identifiers generated in source files for regular expression automaton parser code.

### Parser Base Name
Line edit control for entering prefix of source file names and identifiers generated in source files for context free automaton parser code.

### TTY Code
Check box control that indicates source code for a standard I/O test application will be generated. Calculations for both the regular expression and context free automatons must be completed for the control to be enabled.

### Regexp Table
Check box control that indicates a primary regular expression match table be generated.

### Generate
Button to generate selected code.

### Close
Button to close dialog.

#### 2.4.5 Find String
The **Find String** dialog performs a text string search on a multiline edit control window. The dialog is activated by submenu items and from the keyboard by [Control-G].

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Line edit control for entering string to search for in multiline edit control.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm operation and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

#### 2.4.6 Simulator Error Token
The **Simulator Error Token** dialog is used to select an error token that will be used by the parser simulator when a related error entry occurs. If an error entry occurs which is mapped to an error function for which the error token is selected then the error token is inserted into the input token stream and the parser driver attempts to resynchronize.

<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List selection</td>
<td>List of all error tokens plus an item indicating no error token is to be selected for the error function.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm operation and close dialog.</td>
</tr>
<tr>
<td>Close</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

#### 2.4.7 Simulator Options
The **Simulator Options** dialog controls and configures the operation of the parser simulator.
<table>
<thead>
<tr>
<th>control</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check list control</td>
<td>Selects subset of symbols to trap for halting simulation. Tokens are trapped on shift actions and nonterminals are trapped on reduce actions.</td>
</tr>
<tr>
<td>Simulate</td>
<td>Mutually exclusive selection of Lexical or Parser simulation operation. Lexical simulation runs the regular expression automaton parser and Parser runs both the regular expression and context free automaton parsers in cascade.</td>
</tr>
<tr>
<td>Discard Lexical Error</td>
<td>Check box control that indicates regular expression automaton parser errors are discarded and operation continued.</td>
</tr>
<tr>
<td>With Unit Rule Eliminations</td>
<td>Check box control that indicates parsing tables with unit rule elimination optimization will be used.</td>
</tr>
<tr>
<td>Ok</td>
<td>Button to confirm options and close dialog.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Button to cancel operation and close dialog.</td>
</tr>
</tbody>
</table>

### 3 Parsing Calculations and Code Generation

#### 3.1 Regular Expression Parsers

Regular expression parsers implement automatons that recognize languages generated by regular expressions which are equivalent to those languages generated by right linear grammars. The automatons are characterized as the finite state machine automatons.

#### 3.1.1 Language Grammar

The grammar for regular expression parsers is based on special character rules. A regular expression grammar is formed as a concatenation of characters according to the special character rules.

##### 3.1.1.1 Special Characters

The following characters have special interpretation in regular expressions and must be preceded by a backslash '\' character to have literal interpretation:

1) space character: \`

2) ( start of subexpression grouping

3) ) end of subexpression grouping

4) * zero or more instances of preceding expression

5) + one or more instances of preceding expression

6) ? zero or one instance of preceding expression

7) | or-operator; match one of two expressions

8) \ escape character

9) " quotation of literal character string expressions

10) [ start of character class specification
11) ]  end of character class specification
12) –  separates two characters to form a character range
13) ^  complement of character class specification
14) .  match all characters except newline
15) {  start of repeat range
16) }  end of repeat range
17) ,  delimiter character
18) $  reserved character
19) /  start of trailing context
20) <  start of activation/assignment list
21) >  end of activation/assignment list

3.1.1.2 Non-special Characters
The following characters are interpreted literally:
1) upper and lower case alphabetic characters: A–Za–z
2) digit characters: 0–9
3) other non-special characters: !#%&:';=@_`~

3.1.1.3 Subexpression Grouping
The parentheses characters ( ) are used to form subexpressions. Repeat and or-operator characters, and repeat ranges which immediately follow a parenthetical subexpression act on the entire subexpression.

3.1.1.4 Repeat Characters
The repeat characters are:
1) *  zero or more instances of the preceding expression
2) +  one or more instances of the preceding expression
3) ?  zero or one instance of the preceding expression
Repeat characters act on the least subexpression which immediately precedes the repeat character.

3.1.1.5 Or-Operator Character
The vertical bar character | is used to form alternate match expressions. The expression expr1|expr2 matches either expr1 or expr2. The vertical bar operator is left associative.

3.1.1.6 Escape Character
The backslash character \ is used to escape special characters. A backslash followed by any
character matches the literal character with the following exceptions:
1) \a matches 0x07, alert bell
2) \b matches 0x08, backspace
3) \t matches 0x09, horizontal tab
4) \n matches 0x0a, newline
5) \v matches 0x0b, vertical tab
6) \f matches 0x0c, form feed
7) \r matches 0x0d, carriage return

3.1.1.7 Double Quote Character
All characters between two quotes are matched as literal characters in sequence with two exceptions:
1) " must be preceded by a \ escape to match a literal "
2) \ must be preceded by a \ escape to match a literal \n
3.1.1.8 Character Class Specifications
A character class specification has two forms:
1) \[item_list]\n2) \[^item_list]\n
where item_list is a list of character class items.
In the first form the expression matches any character in the set specified by item_list. In the second form the expression matches any character not in the set specified by item_list.

A character class item is one of two forms:
1) A single character which can be a non-special character, a special character other than " \ [ ] , or an escape character.
2) A character range of the form item-item, where each item is a single character item as described above.

A special class specification syntax is recognized for arbitrary character values. This syntax is one of the following forms:
1) \[numeric_escape]\n2) \[numeric_escape;numeric_escape]\n
The first form specifies a single character value. The second form specifies a range of character values. The syntax of numeric_escape is one of the following, where digit_list is a list of decimal digits 0–9 and hexdigit_list is a list of hexadecimal digits 0–9a–fA–F:
1) \\digit_list if list begins with a 0 the conversion is octal, otherwise it is decimal
2) `\\hexdigit_list`
3) `\\Xhexdigit_list`

### 3.1.1.9 Match All Character

The period character `.` matches all characters except the newline character `\n`.

### 3.1.1.10 Repeat Ranges

A repeat range has one of the following forms where `digit_list1` and `digit_list2` are lists of decimal digits 0–9:

1) `{digit_list1,digit_list2}` - `digit_list1` to `digit_list2` instances of the preceding expression
2) `{digit_list1,}` - `digit_list1` or more instances of the preceding expression
3) `{,digit_list2}` - zero to `digit_list2` instances of the preceding expression

Repeat ranges act on the least subexpression which immediately precedes the repeat range.

### 3.1.1.11 Trailing Context

Trailing context is specified with a forward slash character `/`. When a match occurs for the expression `prefix/suffix` the token value is `prefix` and the characters matching `suffix` are returned to the input stream and processed again.

### 3.1.1.12 Activation and Assignment Contexts

A state machine mechanism for controlling the recognition of terminals is available using the following syntax:

```
<string_list> - activation or assignment context, where string_list is a list of character strings separated by commas; each character in each string is a non-special character or a special character escaped by a backslash \\.
```

The strings in the `string_list` specify boolean context variables. When `<string_list>` appears on the left of a regular expression it specifies an activation context. When `<string_list>` appears on the right of a regular expression it specifies an assignment context. When the regular expression automaton parser begins processing an input stream the context variables are set according to the value of the `start context` control which uses the same assignment context syntax. If a string occurs in the start context the corresponding variable is set to true. If it does not occur the variable is set to false unless the start context is empty. If the start context is empty then all context variables are set to true.

The next terminal recognized will be a terminal which has an activation context that contains a string whose corresponding variable is true or a terminal which does not have an activation context. When a terminal is recognized that has an assignment context all variables for the strings in the assignment context are set to true and all other variables are set to false. When a terminal is recognized that does not have an assignment context then all context variables are unchanged. As terminals are recognized the true variable subset will change to produce a transition from one terminal recognition subset to another. Within the current active terminal
subset the terminal that is recognized is the one whose regular expression matches the longest string with the least terminal entry index.

The following example demonstrates the activation and assignment contexts along with a trailing context expression.

token 1 = <STATE1>[A-Za-z]
token 2 = <STATE1>[0-9]
token 3 = <STATE1>##<STATE2>
token 4 = <STATE2>[A-Za-z]{1,2}
token 5 = <STATE2>[0-9]{1,2}
token 6 = <STATE2>##<STATE1>
token 7 = <STATE3>[0-9][a-z][0-9]<STATE1>
token 8 = abc/def<STATE2,STATE3>

The input stream

#abc012##ABC012abcdef0a11abc##012

is parsed into the tokens

# a b c 0 1 2 ## AB C 01 2 abc de f 0a l 1 a b c ## 01 2

3.1.2 Calculations

3.1.2.1 Single Character Type

The single character type regular expression automaton parser maps individual characters to token codes. It is implemented as a lookup table on character codes. Input characters are mapped as named individual characters and as named character classes to the symbol values that appear in the terminal subview entries. The following table shows the named characters and named character classes that are supported and the equivalent regular expressions matching the same characters or character classes. The mapping supported by the parser driver is designed specifically for the implementation of the regular expression syntax parser built into the CPG application. Note how the partitioning of the character mapping corresponds to the special and non-special character functions of the regular expression grammar syntax.

<table>
<thead>
<tr>
<th>name</th>
<th>regexp</th>
<th>name</th>
<th>regexp</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>[A-Za-z]</td>
<td>[A-Za-z]</td>
<td>[</td>
</tr>
<tr>
<td>digit</td>
<td>[0-9]</td>
<td>[0-9]</td>
<td>]</td>
</tr>
<tr>
<td>other</td>
<td>[!#$%&amp;'();@_`~]</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>space</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>,</td>
<td>,</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>
When implementing a single character type parser the name should be entered into the name column and regexp into the regexp column of a terminal specification entry. The Lexical Type selection in the Calculate dialog should be set to SingleChar. The Input Range selection is ignored for a single character type parser.

### 3.1.2.2 Regular Expression Type

The regular expression type automaton is calculated from the regular expressions for the terminal entries. The regular expression for each terminal is parsed to create a nondeterministic finite state automaton. The automata are combined by an or-operator into a single nondeterministic finite state automaton. From the representation of the automaton which includes all transitions, final states, and trailing context prefix final states a recursive calculation produces an equivalent deterministic finite state automaton. The Lexical Type selection in the Calculate dialog should be set to Regexp. The Input Range selection determines the expected input character values which determines the number of columns in the finite state machine tables. A range selection of 128 is used for character values 0–127 and a selection of 256 is used for character values 0–255.

### 3.1.3 Parser Driver Interface

A regular expression automaton parser driver is built from base class, derived class, and parser driver table source files. The following table shows the source files for both the single character type and the regular expression type parsers. Generated files have names beginning with the prefix lbase which is set in the Lexical Base Name control of the Generate Code dialog. The lbase prefix is also used in the creation of derived class name identifiers.

The class derivation relations are:

CLexicalSingleCharDriver --> CLexicalDriver

CLbaseLexicalDriver --> CLexicalRegexpDriver --> CLexicalDriver

<table>
<thead>
<tr>
<th>Files common to both types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base class files</strong></td>
</tr>
<tr>
<td><code>lexicalDriver.cpp</code> - source file containing base class definitions for CLexicalDriver, CLexicalSingleCharDriver, and CLexicalRegexpDriver.</td>
</tr>
<tr>
<td><code>lexicalDriver.h</code> - include for base classes.</td>
</tr>
<tr>
<td><code>tvaluelist.h</code> - linked list template include file.</td>
</tr>
<tr>
<td><code>stackElement.h</code> - parser stack element class definitions used in multiple parser stage interfacing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single Character Type Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generated character mapping table file</strong></td>
</tr>
<tr>
<td><code>lbaseSingleCharData.h</code> - contains string array which is supplied as a constructor argument for defining character and character class mapping to parser input token codes.</td>
</tr>
</tbody>
</table>

| Regular Expression Type Parser |
### 3.1.3.1 Base Class Interface

**CLexicalDriver** public members:

```cpp
enum InputError // parser input operation error codes
{
    NoError = 0,  // return code for no error on input operation
    OpenError = 1,  // error on file or stream open operation
    CloseError = 2,  // error on file or stream close operation
    ReadError = 4  // error on file or stream read operation
};
```

// The functions for setting the parser input can be called at any time
// to set the input to a new source. A previously open input source is
// automatically closed. These functions return an InputError value
// indicating the operation result status.

```cpp
// Set the parser input to an array buffer of specified size.

InputError SetInputBuffer(const unsigned char *i_buffer, int i_size);
```

// return - operation result status
// arguments
// i_buffer - pointer to buffer containing input character data
// i_size   - size of input buffer

```cpp
// Set the parser input to a standard library string.

InputError SetInputString(const string &i_string);
```

// return - operation result status
// arguments
// i_string - reference to standard library string

```cpp
// Set the parser input to a standard library stream I/O istream.

InputError SetInputIoStream(       istream *i_stream,
int b_size = default_block_size);
```
// return      - operation result status
// arguments
// i_stream     - pointer to standard library stream I/O istream
// b_size       - requested number of bytes read during buffer
//             fill operation

// Set the parser input to a standard library stream I/O fstream by
// file name.

InputError SetInputFileIoStream( const string &f_name,
                                 int b_size = default_block_size);

// return      - operation result status
// arguments
// f_name       - input data file name
// b_size       - requested number of bytes read during buffer
//             fill operation

// Set the parser input to a standard library FILE stream.

InputError SetInputStdStream( FILE *f,
                              int b_size = default_block_size);

// return      - operation result status
// arguments
// f           - pointer to FILE stream
// b_size       - requested number of bytes read during buffer
//             fill operation

// Set the parser input to a standard library FILE stream by file name.

InputError SetInputFileStdStream( const string &f_name,
                                   int b_size = default_block_size);

// return      - operation result status
// arguments
// f_name      - input data file name
// b_size       - requested number of bytes read during buffer
//             fill operation

// Set the parser input to a standard library file descriptor.

InputError SetInputDescriptor( int i_d,
                               int b_size = default_block_size);
Compiler – a Parser Generator Program

// return - operation result status
// arguments
// i_d - input data file descriptor
// b_size - requested number of bytes read during buffer fill operation

// Set the parser input to a standard library file descriptor by file name.
InputError SetInputFileDescriptor( const string &f_name,
                                   int b_size = default_block_size);

// return - operation result status
// arguments
// f_name - input data file name
// b_size - requested number of bytes read during buffer fill operation

// Initialize the parser. Call once before parser processes input.
virtual void Initialize(int t_size = 100, bool ptf = true) = 0;

// arguments
// t_size - initial size of internal terminal character array buffer
// ptf - process terminal flag, true indicates token and filter code section member functions are called, false otherwise

// Get next token terminal. Trap filter terminals and execute filter code section member functions if the process terminal flag is true.
virtual int GetToken() = 0;

// return - token code

// Get next token terminal. Trap filter terminals and execute filter code section member functions if the process terminal flag is true. Arguments are set to the token character array and size.
virtual int GetToken(unsigned char *token_ptr, int &token_size) = 0;

// return - token code
// arguments
// token_ptr - reference to pointer to character array set to the token character array
// token_size - reference to int set to the size of the token character array
virtual void     LoadStackElement(CStackElement *se) = 0;

// arguments
// se - pointer to CStackElement class, members are set to token or derived data by token code section member function

virtual unsigned char    *GetTerminalString() = 0;

// return       - pointer to terminal character array data

virtual int              GetTerminalLength() = 0;

// return       - size of the current terminal string data not including
//                a null termination character

void     PushCurrentTokenBack();

// Sets the process terminal flag to false which halts execution of
// terminal code section member functions. Calls HaltProcessFilter().

void     HaltProcessTerminal();

virtual int      GetCurrentCharNumber();
// return - current input character number

// Returns the current line number as determined by newline characters \n relative to a starting line number of zero.
virtual int GetCurrentLineNumber();

// return - current line number

// Returns the current column number as determined by the number of characters appearing after the previous newline character \n with column numbers starting at zero.
virtual int GetCurrentColumnNumber();

// return - current column number

// Generate status string containing current character, line, and column information. The result is returned both as a function return value and through an argument reference. The flag argument selects either one or zero based numbers.
virtual string &GetLexicalStatusString(string &s,
                                         bool one_based = true);

// return - reference to string that is assigned status string arguments
// s - reference to string that is assigned status string
// one_based - true generates values starting from one, false generates values starting from zero

CLexicalDriver protected members:

// Function is called by HaltProcessTerminal() function. Handles any additional halt recovery processing related to filter terminals.
virtual void HaltProcessFilter() = 0;
### 3.1.3.2 Single Character Type Interface

A single character type parser is implemented using the `CLexicalSingleCharDriver` class directly without requiring any further derived objects.

`CLexicalSingleCharDriver` public members:

```cpp
CLexicalSingleCharDriver( const char * const lex_char_token_names[],
                        int est = 0);
```

// arguments
// lex_char_token_names  - character string array that establishes
//                          character and character class mapping to
//                          token codes. The codes are assigned the
//                          values of the array indexes of the strings
// est                    - empty symbol token code synonymous with
//                          the end of file symbol token code

// Destructor function for the parser driver class.
~CLexicalSingleCharDriver();
```

### 3.1.3.3 Regular Expression Type Interface

A regular expression type parser is derived from `CLexicalRegexpDriver`.

`CLbaseLexicalDriver` derived class public members:

```cpp
CLbaseLexicalDriver();
```

// Constructor function for derived parser driver class.

```cpp
~CLbaseLexicalDriver();
```

// Destructor function for derived parser driver class.
3.1.3.4 Application Interface

The organization of a code sequence to implement a stand alone regular expression type parser follows a general form:

```c
#include "lbaseLexicalDriver.h"

extern istream cin;

void function1()
{
    int token_code;
    int token_size;
    unsigned char *token_ptr;
    CStackElement stack_element;
    CLbaseLexicalDriver *lexical = new CLbaseLexicalDriver();

    lexical->SetInputIoStream(&cin);
    lexical->Initialize();

    while (1)
    {
        token_code = lexical->GetToken(token_ptr, token_size);

        // token_code, token_ptr, and token_size are now set to values for
        // for the current token. Check token code to see if end of file
        // has been reached.

        If (token_code == Lbase_Empty_Symbol_Token)
            break;

        // The LoadStackElement function will invoke the token section code
        // member function for the current token. This function is
        // automatically called by an integrated context free parser.

        LoadStackElement(&stack_element);
    }

    delete lexical;
}
```

3.1.4 Code Generation

The code generation for regular expression type derived parser class objects supports code sections that are executed upon object construction, object destruction, token and filter recognition, plus code for arbitrary class data and function members. The following schematic of generated files shows the location of the different code sections.
Compiler – a Parser Generator Program 65

ibaseLexicalDriver.h

#ifndef BASELEXICALDRIVER_H
#define BASELEXICALDRIVER_H

#include "lexicalDriver.h"
#include "lbaseRegexpData.h"

linky

Lexical Include code location
linky

class CLbaseLexicalDriver: public CLexicalRegexpDriver
{
  public:
    CLbaseLexicalDriver();
    virtual ~CLbaseLexicalDriver();

    virtual void SetupFunctionTables();
    virtual void LoadStackElement(CStackElement *se);
    virtual void ProcessFilter();

linky

Lexical Class code location
linky

linky

// terminal function declarations for CLbaseLexicalDriver

void (CLbaseLexicalDriver::* *token_functions)(CStackElement *se);
void (CLbaseLexicalDriver::* *filter_functions)();

void Token_00(CStackElement *se);
linky
linky
linky

void Token_count_minus_1(CStackElement *se);

void Filter_0();
linky
linky
linky

void Filter_count_minus_1();
};

#endif
#include "lbaseLexicalDriver.h"

CLbaseLexicalDriver::CLbaseLexicalDriver()
    : CLexicalRegexpDriver(
        Lbase_Token_Count,
        Lbase_Filter_Count,
        Lbase_Terminal_Count,
        Lbase_Empty_Symbol_Token,
        Lbase_Input_Range,
        Lbase_FSM_Count,
        Lbase_FSM_Table,
        Lbase_Regexp_Match,
        Lbase_Prefix_Match,
        Lbase_Terminal_Match,
        Lbase_Start_Assignment,
        Lbase_Trailing_Context,
        Lbase_Token_Map,
        Lbase_Filter_Map,
        Lbase_Literal_Terminal,
        Lbase_Single_Column_Match)
{
    token_functions =
        new (void (CLbaseLexicalDriver::*[token_count])(CStackElement *));

    filter_functions =
        new (void (CLbaseLexicalDriver::*[filter_count]()));

    SetupFunctionTables();

    delete [] token_functions;
    delete [] filter_functions;
}

CLbaseLexicalDriver::~CLbaseLexicalDriver()
{
    delete [] token_functions;
    delete [] filter_functions;
}
Compiler – a Parser Generator Program  67

//
//   ..............................................
//

// terminal function definitions for C1baseLexicalDriver

void C1baseLexicalDriver::Filter_0()
{
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// terminal code location for filter 0
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
}

//
//   ..............................................
//

void C1baseLexicalDriver::Filter_count_minus_1()
{
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// terminal code location for filter (filter_count - 1)
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
}

void C1baseLexicalDriver::Token_00(CStackElement *se)
{
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// terminal code location for token 0
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
}

//
//   ..............................................
//

void C1baseLexicalDriver::Token_count_minus_1(CStackElement *se)
{
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// terminal code location for token (token_count - 1)
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
}

3.1.4.1 Terminal Code

Source code for terminal code section member functions is entered using the terminal select and the terminal code subviews. The terminal select subview is used to select the terminal whose code is displayed and edited in the terminal code subview. The code entered will form the body of a derived parser class member function whose signature is dependent on whether the terminal type is FILTER or TOKEN. A terminal code section member
function is invoked only if the process terminal flag is true and the terminal driver specification is set to REGEXP.

### 3.1.4.1.1 Token Code

Token terminal code section member functions have the signature:

```cpp
void CbaseLexicalDriver::Token_symbol(CStackElement *se);
```

The function name has a suffix of `symbol` which is the digit string of the `symbol` column value for the token terminal entry. The `symbol` value is the token code that is used by an integrated context free automaton parser. The `CStackElement` pointer argument points to a class instance that is used to store the token and any derived data for interfacing to a context free parser. If the regular expression parser is operated as a stand alone parser stage then the `LoadStackElement(CStackElement *se)` function is called to invoke the token terminal code section member functions.

Writing token code involves these general steps:

1. Use the `GetTerminalString()` and `GetTerminalLength()` base class functions to access and process the token string data.

2. Store the token or derived data into the proper class data members of the `CStackElement` argument; this operation will typically take the form:
   ```cpp
   se->data_member1 = convert_function(GetTerminalString());
   ```

### 3.1.4.1.2 Filter Code

Filter terminal code section member functions have the signature:

```cpp
void CbaseLexicalDriver::Filter_symbol();
```

The function name has a suffix of `symbol` which is the digit string of an integer value that is the position in a sequential numbering from zero of the filter terminals. The base class functions `GetTerminalString()` and `GetTerminalLength()` are used to process the filter data.

### 3.1.4.2 Class Code

The class code sections are displayed and edited in the `class code` subview. The subview has a control for selecting the `Lexical Include`, `Lexical Class`, `Lexical Source`, `Lexical Constructor`, and `Lexical Destructor` code sections. The sections can be used to create any include statements, class declarations and definitions, and global declarations and definitions that are required.

### 3.1.5 Parser Tables

The following illustrations show the data generated for regular expression type parsers. All constants and tables are illustrated and are described in terms used by the illustrations of the parser driver logic. The information in both sets of illustrations is a complete specification of the operation of the parser driver that is directly dependent on the automaton calculations and the generated constants and tables.
Generated Regular Expression Type Parser Data

(prefix) - generated code identifier prefix
(prefix)_Token_Count = number of token terminals
(prefix)_Terminal_Count = number of terminals
(prefix)_Empty_Symbol-Token = end of file (EOF) symbol
(prefix)_Input_Range = input character value range
(prefix)_FSM_Count = finite state machine count
(prefix)_Single_Column_Match = 1 indicates terminal match table has one column, 0 indicates terminal match table has
(prefix)_FSM_Table = finite state machine automaton table,
   size of table = (prefix)_FSM_Count * (prefix)_Input_Range
(prefix)_Regexp_Match = regular expression match table,
   size of table = (prefix)_FSM_Count * (prefix)_Terminal_Count
(prefix)_Prefix_Match = trailing context prefix match table,
   size of table = (prefix)_FSM_Count * (prefix)_Terminal_Count
(prefix)_Terminal_Match = terminal match table,
   if single_column_match = 0
   size of table = (prefix)_FSM_Count * ((prefix)_Terminal_Count+1)
   if single_column_match = 1
   size of table = (prefix) FSM Count
(prefix)_Start_Assignment = start assignment flag table,
   size of table = (prefix)_Terminal_Count,
   flag = 1 indicates terminal has a start assignment and the
   corresponding column of the terminal match table becomes active when
   the terminal is recognized. flag = 0 indicates otherwise
(prefix)_Trailing_Context = trailing context flag table,
   size of table = (prefix)_Terminal_Count,
   flag = 1 indicates terminal has a trailing context regular expression
   and once the full regular expression for the terminal is recognized
   the trailing context prefix match table is used to locate the prefix
   portion that is returned as the final matched terminal,
   flag = 0 indicates otherwise
(prefix)_Token_Map = terminal to token map table,
   size of table = (prefix)_Terminal_Count,
   maps terminal index to token index, filter terminals are mapped to -2,
   token terminals are mapped to indexes that are consecutive from 0 to
   token count-1 corresponding to the order of the tokens
(prefix)_Filter_Map = terminal to filter map table,
   size of table = (prefix)_Terminal_Count,
   maps terminal index to filter index, token terminals are mapped to -1,
   filter terminals are mapped to indexes that are consecutive from 0 to
   filter count-1 corresponding to the order of the filters
(prefix)_Literal_Terminal = literal terminal flag table,
   size of table = (prefix)_Terminal_Count,
   flag = 1 indicates terminal is intended to be processed for data
   content, flag = 0 indicates otherwise

Illustration 24  Generated Regular Expression Type Parser Data
Structure and Numbering Scheme of Regular Expression Parser Tables – 1

Finite State Machine Table

s = number of machine states
r = character input range
f = finite state machine entry

Transitions for entry \( f = (i, j) \)
where \( i \) is a machine state and
\( j \) is an input character:

1) \( f = -1 \)
error
2) \( 0 \leq f \leq s-1 \)
from machine state \( i \) goto

<table>
<thead>
<tr>
<th>initial state</th>
<th>machine state 0</th>
<th>machine state 1</th>
<th>machine state 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( f \ f f )</td>
<td>( f f f )</td>
<td>( f f f )</td>
</tr>
<tr>
<td></td>
<td>( f f f )</td>
<td>( f f f )</td>
<td>( f f f )</td>
</tr>
<tr>
<td></td>
<td>( f f f )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Regular Expression Match Table

s = number of machine states
m = number of terminals
b = regular expression match flag entry

For entry \( b = (i, j) \) where \( i \) is
a machine state and \( j \) is a
terminal index:

1) \( b = 1 \) indicates machine
state \( i \) is a final state
for terminal \( j \)

<table>
<thead>
<tr>
<th>terminal</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>( H )</th>
<th>( H )</th>
<th>( H )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>machine state 0</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>machine state 1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>machine state 2</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>machine state s-3</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>machine state s-2</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>machine state s-1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Illustration 25 Structure and Numbering Scheme of Regular Expression Parser Tables - 1
Structure and Numbering Scheme of Regular Expression Parser Tables – 2

Trailing Context Prefix Match Table

s = number of machine states
m = number of terminals
b = trailing context prefix match flag entry

For entry \( b = (i, j) \) where \( i \) is a machine state and \( j \) is a terminal index:
1) \( b = 1 \) indicates machine state \( i \) is a prefix state for terminal \( j \)

<table>
<thead>
<tr>
<th>Terminal</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>m-3</th>
<th>m-2</th>
<th>m-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine state 0</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>machine state 1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>machine state 2</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>machine state s-3</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>machine state s-2</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>machine state s-1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>*</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

Terminal Match Table

s = number of machine states
m = number of terminals
h = terminal match entry

<table>
<thead>
<tr>
<th>Terminal</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>m-3</th>
<th>m-2</th>
<th>m-1</th>
<th>m-1 context</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine state 0</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>*</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>machine state 1</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>*</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>machine state 2</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>*</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>machine state s-3</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>*</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>machine state s-2</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>*</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>machine state s-1</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>*</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
</tbody>
</table>

For entry \( h = (i, j) \), where \( i \) is a machine state and \( j \) is a terminal index or the start context:
1) \( 0 \leq h \leq m-1 \) indicates machine state \( i \) is a final state for terminal \( h \) where terminal \( h \) is the first terminal with machine state \( i \) as a final state and a start activation included in the start assignment of terminal \( j \) or the start context.
2) \( h = -1 \) indicates machine state \( i \) is not a final state for any terminal with a start activation included in

Illustration 26 Structure and Numbering Scheme of Regular Expression Parser Tables - 2
### 3.1.6 Parser Logic

The following illustrations show the parsing table dependent logic of the parser driver.

#### Regular Expression Parser – Table Based Logic – 1

**Viable String Determination**

<table>
<thead>
<tr>
<th>initial machine state = 0</th>
<th>machine state</th>
</tr>
</thead>
<tbody>
<tr>
<td>list element index 0</td>
<td>character</td>
</tr>
<tr>
<td></td>
<td>machine state</td>
</tr>
<tr>
<td></td>
<td>match terminal</td>
</tr>
</tbody>
</table>

**Automaton Configuration**

\[ c = \text{viable string list count} \]

<table>
<thead>
<tr>
<th>character input stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite State Machine Table</td>
</tr>
</tbody>
</table>

**Terminal Match Table**

\[ \text{current start condition} \]

1) \( \text{next\_machine\_state} = \) 
   \( \text{fsm\_table[current\_machine\_state][next\_character]} \)

2) if single column match = 0 
   \( \text{match\_terminal} = \) 
   \( \text{terminal\_match\_table[next\_machine\_state][current\_start\_condition]} \)

   else if single column match = 1 
   \( \text{match\_terminal} = \) 
   \( \text{terminal\_match\_table[next\_machine\_state]} \)

*Illustration 27  Regular Expression Parser - Table Based Logic - 1*
Regular Expression Parser – Table Based Logic – 2

Viable String Determination

1. Initialize machine state to 0.
2. Get next character from character input stream.
3. Compute next machine state by equation 1.
4. If next machine state is not equal to -1 (true), push new character list element and set character to next character, machine state to next machine state, and match terminal to the computed match terminal value.
5. If next machine state is equal to -1 (false), compute match terminal by equation 2.

Viable string consists of characters from list elements in order.

Illustration 28  Regular Expression Parser - Table Based Logic - 2
Regular Expression Parser – Table Based Logic – 3

Terminal and Prefix Match String Determination

<table>
<thead>
<tr>
<th>list element index 0</th>
<th>character</th>
<th>machine state</th>
<th>match terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>list element index p-1</td>
<td>character</td>
<td>machine state</td>
<td>match terminal</td>
</tr>
<tr>
<td>list element index p</td>
<td>character</td>
<td>machine state</td>
<td>match terminal</td>
</tr>
<tr>
<td>list element index k-1</td>
<td>character</td>
<td>machine state</td>
<td>match terminal</td>
</tr>
<tr>
<td>list element index k</td>
<td>character</td>
<td>machine state</td>
<td>match terminal</td>
</tr>
<tr>
<td>final machine state</td>
<td>character</td>
<td>machine state</td>
<td>match terminal</td>
</tr>
</tbody>
</table>

Automaton Configuration

\[ c = \text{viable string list count} \]
\[ k = \text{terminal match string list count} \]
\[ p = \text{prefix match string list count} \]

1) \(\text{start_assignment_flag} = \text{start_assignment_table}[\text{match_terminal}]\)
2) \(\text{trailing_context_flag} = \text{trailing_context_table}[\text{match_terminal}]\)
3) \(\text{prefix_match_flag} = \text{prefix_match_table}[\text{machine_state}][\text{match_terminal}]\)
Regular Expression Parser – Table Based Logic – 4

Terminal and Prefix Match String Determination

1. **Match String Determination**
   - Initialize list index to c-1.

2. Get match terminal from list element with current list index.

3. **Match terminal = -1**
   - true
     - Set current start condition to match terminal and set the corresponding column of the terminal match table active.
   - false
     - Compute trailing context flag by equation 2.

4. **Start assignment flag = 1**
   - true
     - Compute trailing context flag by equation 2.
   - false
     - Match string consists of characters from list elements with indexes 0 through k-1.

5. **List index != -1**
   - true
     - Compute start assignment flag by equation 1.
   - false
     - Process lexical error.

6. **Determine prefix match string.**

Illustration 30  Regular Expression Parser - Table Based Logic - 4
Regular Expression Parser – Table Based Logic – 5

Terminal and Prefix Match String Determination

- Determine prefix match string.
- Initialize list index to $k-1$.
- Get machine state from list element with current list index and compute prefix match flag by equation 3.

- If prefix match flag = 1
  - False: Decrement list index.
  - True: Match string consists of characters from list elements with indexes 0 through $p-1$.

*Illustration 31  Regular Expression Parser - Table Based Logic - 5*
3.2 Deterministic Context Free Parsers

Deterministic context free parsers implement one way pushdown automatons that are operated deterministically. The languages recognized by these parsers are generated by context free grammars.

3.2.1 Language Grammar

The grammar for a deterministic context free parser is based on token terminal symbols, error token symbols, nonterminal symbols, and context free rules. The token terminals are those entries in the terminal subview with a terminal type of TOKEN. The error tokens and nonterminals are the entries in the error token and nonterminal subviews respectively. The context free rules are the entries in the rule subview.

Context free rules are of the form:
nonterminal -> symbol_list where symbol_list is a list of zero or more terminals, error tokens, and nonterminals.

The grammar input to the application is implicitly augmented by one token terminal, one nonterminal, and one rule. The augmented grammar elements are:

1) EOF – end of file symbol appended to the end of a character input stream.
2) ASTART – augmented start symbol.
3) ASTART -> (start symbol) EOF - augmented start rule, where start symbol is the unique nonterminal symbol that has its start symbol flag set in the nonterminal subview.

The augmented grammar elements are added to a grammar specification in order to introduce the end of file symbol and to simplify the determination of an accept action for a deterministic context free parser. The implicit subview shows the augmented grammar elements and the element numbering within the parser representation.

3.2.2 Calculations

A context free grammar must satisfy two initial conditions before an automaton calculation can proceed. These conditions are automatically checked by the application when a calculation is started.

1) Derives Terminal – a nonterminal symbol must derive a terminal symbol string consisting of zero or more token terminals and error tokens.
2) Accessible Symbol – every token terminal symbol, error token symbol, and nonterminal symbol must be appear in a derivation beginning with the grammar start symbol.

If both conditions are true the grammar is said to be a regular grammar and will then satisfy the hypotheses of automaton calculation theorems used by the application.
3.2.2.1 LR(k) Automaton

Three different LR(k) based automatons can be calculated. All calculations result in parsing table representations that use the same one token lookahead parsing driver. The automaton calculation type is selected in the Calculate dialog.

1) SLR(1) – the LR(0) machine is calculated and the parsing actions are determined by calculating the first and follow sets for the grammar.

2) LALR(1) – the LR(0) machine is calculated and the parsing actions are determined by calculating the LALR(1) lookaheads.

3) LR(1) – the LR(1) machine is calculated.

The action and goto parsing tables computed are a direct representation of the machine states, token actions, and symbol transitions of the automaton. All action table error entries have a default value of −1. Default conflict resolution is applied to all parsing conflict entries after the automaton calculation is complete. Any non-default conflict resolutions stored in the parser specification file are then applied after being checked for consistency with the parsing tables.

3.2.2.2 Conflicts and Precedences

Parsing conflicts within the action table are displayed and resolved by the conflict, action, and status subviews. The default resolution of the conflicts is determined by a set of default rules and by the precedence subview specification entries. The precedence entries are ordered by their index so that the entries with the highest index have the highest precedence. The default conflict resolution rules are:

1) The precedence level assigned to a rule is the same as the last token terminal or error token appearing in the rule which has a precedence level, or the level assigned to the rule as an entry in a precedence specification element, or none.

2) If conflict is a shift, 1 reduce conflict and the rule and lookahead terminal token have an assigned precedence then if the precedences differ the conflict is resolved to the action with the highest precedence, which is reduce for a rule and shift for a terminal token, and if the precedences are the same then associativity is used.

3) If conflict is a shift, 1 reduce conflict and the rule or the lookahead terminal token do not have assigned precedences then the conflict is resolved to shift.

4) If conflict is a multiple reduce conflict then if shift is a valid action the conflict is resolved to a shift, and if shift is not a valid action the conflict is resolved to the valid rule reduce action with the lowest index.

3.2.2.3 Essential Error Entries

The essential error entry calculation determines which parsing action table error entries are accessible and how the entries are used to invoke error code section member functions. The Error Entries dialog controls and configures the calculation of essential error entries, the numbering of error entries, and the mapping and aggregation of error code section member functions. Three parameters select the configuration of the calculation and numbering:
1) **Essential Error Entry Calculation** – selects the type of calculation; No Calculation is used for simple error entry numbering where all error entries are considered essential, Approximate for an efficient determination of a superset of the essential entries, Exact for exact determination of the essential entries.

2) **Error Function Numbering** – determines the numbering of parser table error entries; One Number sets all essential error entries to a single number, Number Per State sets essential error entries to the values assigned to the automaton machine states in which the errors occur, decreasing negative integers assigned to successive machine states, Number Per Entry sets essential error entries to consecutive decreasing negative integers.

3) **Error Function Aggregation** – determines how error entry indexes are mapped to error code section member functions; One Function maps all essential error index values to one error function, Function Per State maps all essential error index values for an automaton machine state to a single function with a separate function for each machine state, Function Per Entry maps each essential error index value to an individual error function.

The essential error entry calculation is performed after the LR(k) automaton calculation is complete and any parsing conflicts have been resolved to the desired actions.

### 3.2.2.4 Unit Rule Eliminations

The unit rule elimination calculation creates action and goto parsing tables with additional states that eliminate unnecessary rule reduction actions by unit rules. A unit rule can be marked for potential elimination using the **Unit Rule Elimination** dialog if no code generation is associated with the rule. The unit rule elimination calculation is performed after the essential error entry calculation is complete.

### 3.2.3 Parser Driver Interface

A deterministic context free automaton parser driver is built from base class, derived class, and parser driver table source files. The following table shows the source files. Generated files have names beginning with the prefix `pbase` which is set in the **Parser Base Name** control of the **Generate Code** dialog. The `pbase` prefix is also used in the creation of derived class name identifiers.

The class derivation relation is:

```
CPbaseParserDriver --> CParserDriver
```

<table>
<thead>
<tr>
<th>Base class files</th>
<th>parserDriver.cpp - source file containing base class definitions for CParserDriver.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>parserDriver.h - include for base class.</td>
</tr>
<tr>
<td></td>
<td>lexicalDriver.h - include for regular expression parser base class.</td>
</tr>
<tr>
<td></td>
<td>tvaluelist.h - linked list template include file.</td>
</tr>
<tr>
<td></td>
<td>stackElement.h - parser stack element class definitions used in multiple parser stage interfacing.</td>
</tr>
</tbody>
</table>
**3.2.3.1 Base Class Interface**

CParseDriver public members:

```cpp
defined derived class files
pbaseParserDriver.cpp - source file containing derived class definitions including constructor, destructor, and rule, conflict, and error code member functions.
pbaseParserDriver.h - include for derived class.
pbaseParserData.h - one way pushdown automaton state machine parser driver tables.
```

```
enum StatusCode       // parser operation error codes
{
    NoError               = 0,   // no error status
    LexicalError          = 1,   // lexical stage error occurred
    ParserError           = 2,   // syntax error occurred
    CodeGenerationError   = 4    // rule reduction code generation error
    ParserHalted          = 8    // halt parser flag set
    EndOfFile             = 16   // input stream accepted by parser
};
```

```
// Initialize the parser. Call once before parser processes input.

void   Initialize( int iss = 100, bool icgf = true,
                   bool dlef = false, bool erf = false,
                   bool spm = false);
```

// arguments
// iss - initial parsing stack size
// icgf - initial code generation flag, if true the parser will initially call rule reduce functions for code generation,
// if false then no rule reduce functions are called
// dlef - discard lexical error flag, if true the parser will halt code generation when a lexical error occurs but continue to parse input, if false the parser will halt code generation when a lexical error occurs and then quit
// erf - error reporting flag, if true the parser will output an error message to the error output stream when a lexical error occurs or when the parser is halted by setting the halt parser flag to true, if false then the messages are not generated
// spm - syntax parsing mode flag, if true the parser operates in syntax parsing mode, if false the parser operates in default mode
```
/-----------------------------------------------------------------------------
// Initialize the parser for syntax mode operation. Call once before
// parser processes input.

void InitializeSyntaxMode(int iss = 100, bool icgf = true,
                           bool dlef = false, bool erf = false);

// arguments
// iss     - initial parsing stack size
// icgf    - initial code generation flag, if true the parser will
//           initially call rule reduce functions for code generation,
//           if false then no rule reduce functions are called
// dlef    - discard lexical error flag, if true the parser will halt
//           code generation when a lexical error occurs but continue
//           to parse input, if false the parser will halt code
//           generation when a lexical error occurs and then quit
// erf     - error reporting flag, if true the parser will output an
//           error message to the error output stream when a lexical
//           error occurs or when the parser is halted by setting the
//           halt parser flag to true, if false then the messages are
//           not generated

//-- -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - --
// Set the initial code generation flag.

void SetInitialCodeGenerationFlag(bool f);

// arguments
// f       - initial code generation flag value

//-- -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - --
// Set the discard lexical error flag.

void SetDiscardLexicalErrorFlag(bool f);

// arguments
// f       - discard lexical error flag value

//-- -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - --
// Set the error reporting flag.

void SetErrorReportingFlag(bool f);

// arguments
// f       - error reporting flag value

//-- -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - -- -- -- - -- - --- - -- - - --- -- - - --
// Set the error output stream.

void SetErrorOutputStream(ostream *o_stream);
// arguments
// o_stream - pointer to standard library stream I/O ostream

// Set the halt parser flag to true. Halts code generation and causes
// parser to quit.
void HaltParser();

// Call this function in error code section member functions to insert
// the error token specified by the function argument into the input
// token stream.
void SetErrorToken(int et);

// arguments
// et        - error token code

// Call this function in error code section member functions to set the
// synchronization length of an error token that has been inserted into
// the input token stream by the SetErrorToken(int et) function.
void SetSynchronizationLength(int sl);

// arguments
// sl        - error token synchronization length

// Set the input lexical stage for the parser. The parser will call
// member functions of the lexical object to get tokens from the
// character input stream.
void SetLexical(CLexicalDriver *l);

// arguments
// l        - pointer to a CLexicalDriver derived object

// The current lexical stage is pushed on a stack and the lexical object
// specified by the function argument becomes the new input lexical
// stage for the parser. When the token stream from an input lexical
// stage object is exhausted and the lexical object stack is not empty
// then the object on the top of the lexical object stack is removed and
// becomes the current input stage for the parser. The previous input
// stage object is deleted.
void PushLexical(CLexicalDriver *l);
// arguments
// l         - pointer to a CLexicalDriver derived object

//@------------------------------------------
// The current input lexical stage object is deleted. The object on the
// top of the lexical object stack is removed and becomes the current
// input stage for the parser.

void     PopLexical();

//@------------------------------------------
// Run the parser. The return code is a bitwise-or of StatusCode values.

StatusCode     ParseStream();

//@------------------------------------------
// return     - bitwise-or of StatusCode values

//@------------------------------------------
// Returns the stack element class object that contains the code
// generation object created by the parser when the parser accepts the
// character input stream.

CStackElement    *CompiledObject();

//@------------------------------------------
// return     - pointer to CStackElement that contains code generation
//               object created by the parser

3.2.3.2 Derived Class Interface

A deterministic context free parser is derived from CParseDriver.

CPbaseParserDriver derived class public members:
//@------------------------------------------
// Constructor function for derived parser driver class.

CPbaseParserDriver();

//@------------------------------------------
// Destructor function for derived parser driver class.

~CPbaseParserDriver();

//@------------------------------------------
3.2.3.3 Application Interface

The organization of a code sequence to implement a deterministic context free parser integrated with a regular expression type parser follows a general form:

```cpp
#include "lbaseLexicalDriver.h"
#include "pbaseParserDriver.h"

extern istream cin;

void function1()
{
    CStackElement *stack_element;
    CLbaseLexicalDriver *lexical = new CLbaseLexicalDriver();
    CPbaseParserDriver *parser = new CPbaseParserDriver();

    lexical->SetInputIoStream(&cin);
    lexical->Initialize();
    parser->Initialize();
    parser->SetLexical(lexical);

    CParseDriver::StatusCode sc = parser->ParseStream();

    int lexical_error = sc & CParseDriver::LexicalError;
    int parser_error = sc & CParseDriver::ParserError;

    if (lexical_error)
        cout << "LEXICAL ERROR" << endl;

    if (parser_error)
        cout << "PARSER ERROR" << endl;

    if (lexical_error || parser_error)
    {
        cout << "current line = "
             << lexical->GetCurrentLineNumber() + 1 << endl;
        cout << "current column = "
             << lexical->GetCurrentColumnNumber() + 1 << endl;
    }

    if (sc == CParseDriver::EndOfFile)
    {
        stack_element = parser->CompiledObject();
        // stack_element points to the CStackElement class object that
        // contains code generation objects created by the parser
    }

    delete parser;
    delete lexical;
}```
### 3.2.4 Code Generation

The code generation for deterministic context free parser class objects supports code sections that are executed upon object construction, object destruction, rule reduction, error and conflict entry occurrence, plus code for arbitrary class data and function members. The following schematic of generated files shows the location of the different code sections.

```
#include "parserDriver.h"
#include "pbaseParserData.h"

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  
// Parser Include code location  
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  

class CPbaseParserDriver: public CParseDriver {
public:
    CPbaseParserDriver();
    virtual ~CPbaseParserDriver();

    virtual void SetupFunctionTables();
    virtual int CallReduceFunction(int action, void *data);
    virtual void CallConfli ctFunction(int index);
    virtual void CallErrorFunction(int index);
    virtual void DeleteGeneratedCode(CStackElement *s, int c);

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  
// Parser Class code location  
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  

// rule reduce function declarations for CPbaseParserDriver

    int (CPbaseParserDriver::* *reduce_functions)(void *data);

    int Reduc e_000(void *data);

    //
    // .................................
    //
    int Reduc e_count_minus_1(void *data);

// conflict function declarations for CPbaseParserDriver

    void (CPbaseParserDriver::* *conf lict_functions)();

    void Conflict_0();
```
void Conflict_count_minus_1();

void (CPbaseParserDriver::* error_functions)();

void Error_001();

void Error_count();

#include "pbaseParserDriver.h"

CPbaseParserDriver::CPbaseParserDriver()
: CParserDriver(
    Pbase_Terminal_Count_E,
    Pbase_Symbol_Count_A,
    Pbase_Rule_Count_A,
    Pbase_Parser_Empty_Symbol_Table,
    Pbase_Shift_Action,
    Pbase_Accept_Action,
    Pbase_Conflict_Entry,
    Pbase_State_Count,
    Pbase_Number_Of_Conflict_Entries,
    Pbase_Number_Of_Error_Indexes,
    Pbase_Action_Table,
    Pbase_Goto_Table,
    Pbase_Rule_Left_Symbol_Table,
    Pbase_Rule_Right_Length_Table)
{
    reduce_functions = new (int (CPbaseParserDriver::*[Pbase_Rule_Count_A - 1])(void *));

    conflict_functions = new (void (CPbaseParserDriver::*[Pbase_Number_Of_Conflict_Entries])());
error_functions = new (void (CPbaseParserDriver::*[Pbase_Number_Of_Error_Indexes + 1]))();

SetupFunctionTables();

//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Parser Constructor code location
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

CPbaseParserDriver::~CPbaseParserDriver()
{
    delete [] reduce_functions;
    delete [] conflict_functions;
    delete [] error_functions;

//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Parser Destructor code location
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

//

// rule reduce function definitions for CPbaseParserDriver

int     CPbaseParserDriver::Reduce_000(void *data)
{
    //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    // rule code location for rule 0
    //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    //

    //...................................................................................

    int     CPbaseParserDriver::Reduce_count_minus_1(void *data)
    {
        //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        // rule code location for rule (rule_count - 1)
        //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        
    // conflict function definitions for CPbaseParserDriver

    void     CPbaseParserDriver::Conflict_0()
    {

void CPbaseParserDriver::Conflict_count_minus_1()
{
    // conflict code location for conflict (conflict_count - 1)
    //+++++++++++++++++++++++++++++++++++++++++++++++++
}

void CPbaseParserDriver::Error_001()
{
    // error code location for error 1
    //+++++++++++++++++++++++++++++++++++++++++++++++++
}

void CPbaseParserDriver::Error_count()
{
    // error code location for error (error_count)
    //+++++++++++++++++++++++++++++++++++++++++++++++++
}

### 3.2.4.1 Rule Code

A rule code section member function is invoked by the parser driver when a reduce action is performed for the rule. Source code for rule code section member functions is entered using the *rule select* and the *rule code* subviews. The *rule select* subview is used to select the rule whose code is displayed and edited in the *rule code* subview. The code entered will form the body of a derived parser class member function that has the following signature:

```cpp
int CPbaseParserDriver::Reduce_rule(void *data);
```

The function name has a suffix of *rule* which is the digit string of the *index* column value for the rule entry. The argument `void *data` is used when the parser is initialized for syntax parsing mode and unused otherwise. The function should return −1 for a code generation error.
Rule section code accesses the parsing stack elements that comprise the rule right side through a base class pointer variable:

CStackElement *stack_ptr

Successive stack elements that correspond to the right side symbols of the context free rule are

stack_ptr[0], stack_ptr[1], ..., stack_ptr[m-1]

where m is the rule right side length. The code generation data or objects created by the rule section code should be stored in the stack element

stack_ptr[0]

which then becomes the rule left side symbol resulting from the rule reduction action.

3.2.4.2 Error Code

An error code section member function is invoked by the parser driver when a syntax error entry which is mapped to the function occurs. Source code for error code section member functions is entered using the error and the error code subviews. The error subview is used to select the error function whose code is displayed and edited in the error code subview. The code entered will form the body of a derived parser class member function that has the following signature:

void CPbaseParserDriver::Error_entry();

The function name has a suffix of entry which is the digit string of the index column value for the error function entry. In order for the parser driver to attempt a recovery from a syntax error the error function that handles the error must set two variables:

1) error_token – the parser symbol value for the error token to be inserted into the lookahead token input stream. The value is the integer value in the symbol column of the error token subview entry for the error token.

2) synchronization_length – the number of lookahead tokens the parser driver must accept after the error token in order for the parser driver to be resynchronized with the input stream. The value is the integer value in the sync column of the error token subview entry for the error token.

In order to facilitate the required source code several generated constants and parser driver base class functions are available.

1) Error_Token_name – constant equal to parser symbol value of the error token name.

2) Error_Synchronization_name – synchronization length of error token name.

3) void SetErrorToken(int et) – function which sets error_token variable.

4) void SetSynchronizationLength(int sl) – function which sets synchronization_length variable.

Thus, an error function which attempts to resynchronize the input stream by inserting error token name into the lookahead token stream should include the following code:

SetErrorToken (Error_Token_name);
SetSynchronizationLength(Error_Synchronization_name);

3.2.4.3 Conflict Code

A conflict code section member function is invoked by the parser driver only if the conflict is resolved to the special value Conflict Entry. In this case the conflict code section member function operates as an error code section member function and all the same code development steps apply.

Source code for conflict code section member functions is entered using the conflict and the conflict code subviews. The conflict subview is used to select the conflict entry whose code is displayed and edited in the conflict code subview. The code entered will form the body of a derived parser class member function that has the following signature:

```cpp
void CPbaseParserDriver::Conflict_entry();
```

The function name has a suffix of entry which is the digit string of the index column value for the conflict entry.

3.2.4.4 Class Code

The class code sections are displayed and edited in the class code subview. The subview has a control for selecting the Parser Include, Parser Class, Parser Source, Parser Constructor, and Parser Destructor code sections, plus the Stack Element class declaration. The sections can be used to create any include statements, class declarations and definitions, and global declarations and definitions that are required.
3.2.5 Parser Tables

The following illustrations show the data generated for deterministic context free parsers. All constants and tables are illustrated and are described in terms used by the illustrations of the parser driver logic. The information in both sets of illustrations is a complete specification of the operation of the parser driver that is directly dependent on the automaton calculations and the generated constants and tables.

Generated Context Free Parser Data

(prefix) – generated code identifier prefix
(prefix)_Terminal_Count_E = number of tokens including the empty symbol token
(prefix)_Symbol_Count_A = number of token and nonterminal symbols including the empty symbol token and the augmented start symbol
(prefix)_Rule_Count_A = number of rules including the augmented rule
(prefix)_Parser_Empty_Symbol_Token = end of file (EOF) symbol
(prefix)_Shift_Action = shift lookahead token action
(prefix)_Accept_Action = input token stream accept action, occurs on shift of empty symbol token
(prefix)_Conflict_Entry = base conflict entry value
(prefix)_State_Count = number of one way pushdown automaton machine states
(prefix)_Number_Of_Conflict_Entries = number of parsing conflicts
(prefix)_Number_Of_Error_Indices = number of distinct action table error index values

(prefix)_Action_Table = one way pushdown automaton action table, size of table = (prefix)_State_Count * (prefix)_Terminal_Count_E
(prefix)_Goto_Table = one way pushdown automaton goto table, size of table = (prefix)_State_Count * (prefix)_Symbol_Count_A
(prefix)_Rule_Left_Symbol_Table = table of rule left side symbols, size of table = (prefix)_Rule_Count_A
(prefix)_Rule_Right_Length_Table = table of rule right side lengths, size of table = (prefix)_Rule_Count_A

Illustration 32 Generated Context Free Parser Data
### Grammar Symbol Numbering Scheme

<table>
<thead>
<tr>
<th>index</th>
<th>terminal symbol</th>
<th>symbol</th>
<th>parser symbol type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>token</td>
<td>0</td>
<td>token</td>
</tr>
<tr>
<td>1</td>
<td>token</td>
<td>1</td>
<td>token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>token</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>t-3</td>
<td>token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t-2</td>
<td>token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t-1</td>
<td>token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t</td>
<td>error token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t+e-1</td>
<td>error token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t+e</td>
<td>end of file symbol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t+e+1</td>
<td>nonterminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t+e+2</td>
<td>nonterminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t+e+n-1</td>
<td>nonterminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t+e+n</td>
<td>nonterminal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t+e+n+1</td>
<td>augmented start symbol</td>
</tr>
</tbody>
</table>

m = number of terminals  
t = number of token terminals  
f = number of filter terminals  
e = number of error tokens  
n = number of nonterminals  
m = t + f

---

*Illustration 33  Grammar Symbol Numbering Scheme*
Action Table Structure and Numbering Scheme

- \( s \) = number of machine states
- \( r \) = number of rules including augmented rule
- \( c \) = number of parsing conflicts
- \( a \) = action table entry

Actions:
1) \( 0 \leq a \leq r - 1 \)
   reduce by rule \( a \)
2) \( a = r \)
   shift
3) \( a = r + 1 \)
   accept
4) \( r + 2 \leq a \leq r + c + 1 \)
   conflict entry \( a - r - 2 \)
5) \( a \leq -1 \)
   error entry

**Illustration 34** Action Table Structure and Numbering Scheme
Goto Table Structure and Numbering Scheme

\[ \text{symbol} \quad \text{parser symbol type} \quad \text{token} \quad \text{token} \quad \text{token} \quad \text{token} \quad \text{token} \quad \text{error token} \quad \text{error token} \quad \text{end of file symbol} \quad \text{nonterminal} \quad \text{nonterminal} \quad \text{nonterminal} \quad \text{nonterminal} \quad \text{augmented start symbol} \]

\[
\begin{array}{ccccccccccccc}
\text{symbol} & 0 & 1 & 2 & t-3 & t-2 & t-1 & t & t+e-1 & t+e & t+e+n-1 & t+e+n & t+e+n+1 \\
\hline
\text{machine state 0} & g & g & g & . & g & g & g & . & g & g & g & . & g & g & g \\
\text{machine state 1} & g & g & g & . & g & g & g & . & g & g & g & . & g & g & g \\
\text{machine state 2} & g & g & g & . & g & g & g & . & g & g & g & . & g & g & g \\
\text{machine state s-3} & g & g & g & . & g & g & g & . & g & g & g & . & g & g & g \\
\text{machine state s-2} & g & g & g & . & g & g & g & . & g & g & g & . & g & g & g \\
\text{machine state s-1} & g & g & g & . & g & g & g & . & g & g & g & . & g & g & g \\
\end{array}
\]

Illustration 35  Goto Table Structure and Numbering Scheme
Error Entry Numbering

\( a_e = \) action table error entry \( \leq -1 \)

\( x(i,1) \) = first default error entry in action table row machine state \( i \)

\( x(i,2) \) = last default error entry in action table row machine state \( i \)

\( y(i,1) \) = first essential error entry in action table row machine state \( i \)

\( y(i,2) \) = last essential error entry in action table row machine state \( i \)

<table>
<thead>
<tr>
<th>Essential Error Entry Calculation</th>
<th>Error Entry Numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>One Number</td>
</tr>
<tr>
<td>machine state 0</td>
<td>( a_e = -1 )</td>
</tr>
<tr>
<td>machine state 1</td>
<td>( a_e = -1 )</td>
</tr>
<tr>
<td>machine state s-2</td>
<td>( a_e = -1 )</td>
</tr>
<tr>
<td>machine state s-1</td>
<td>( a_e = -1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Essential Error Entry Calculation</th>
<th>Error Entry Numbering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>One Number</td>
</tr>
<tr>
<td>machine state 0</td>
<td>( a_e = -1,-2 )</td>
</tr>
<tr>
<td>machine state 1</td>
<td>( a_e = -1,-2 )</td>
</tr>
<tr>
<td>machine state s-2</td>
<td>( a_e = -1,-2 )</td>
</tr>
<tr>
<td>machine state s-1</td>
<td>( a_e = -1,-2 )</td>
</tr>
</tbody>
</table>

Illustration 36  Error Entry Numbering
### Error Function Aggregation – Default Error Entries

<table>
<thead>
<tr>
<th>Essential Error Entry Calculation</th>
<th>Error Function Aggregation – Function Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>= No Calculation</td>
<td>One Function</td>
</tr>
<tr>
<td>One Number</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>-3</td>
<td>1</td>
</tr>
<tr>
<td>( -(s-1) )</td>
<td>1</td>
</tr>
<tr>
<td>-s</td>
<td>1</td>
</tr>
<tr>
<td>( x(0,1) )</td>
<td>1</td>
</tr>
<tr>
<td>( x(0,1) - 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(0,2) + 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(0,2) )</td>
<td>1</td>
</tr>
<tr>
<td>( x(1,1) )</td>
<td>1</td>
</tr>
<tr>
<td>( x(1,1) - 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(1,2) + 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(1,2) )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-2,1) )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-2,1) - 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-2,2) + 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-2,2) )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-1,1) )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-1,1) - 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-1,2) + 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( x(s-1,2) )</td>
<td>1</td>
</tr>
</tbody>
</table>

*Illustration 37  Error Function Aggregation - Default Error Entries*
## Error Function Aggregation - Essential Error Entries

<table>
<thead>
<tr>
<th>Essential Error Entry Calculation</th>
<th>Error Function Aggregation - Function Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>= Approximate, Exact</td>
<td>One Function</td>
</tr>
<tr>
<td>One Number</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>-2</td>
</tr>
<tr>
<td>Number Per State</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>-2</td>
</tr>
<tr>
<td></td>
<td>-3</td>
</tr>
<tr>
<td>Number Per Entry</td>
<td>-s</td>
</tr>
<tr>
<td></td>
<td>-(s+1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error Entry Numbering</th>
<th>One Function</th>
<th>Function Per State</th>
<th>Function Per Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>y(0,1)</td>
<td>2</td>
<td>2</td>
<td>-y(0,1)</td>
</tr>
<tr>
<td>y(0,1)-1</td>
<td>2</td>
<td>2</td>
<td>-y(0,1)+1</td>
</tr>
<tr>
<td>y(0,2)+1</td>
<td>2</td>
<td>2</td>
<td>-y(0,2)-1</td>
</tr>
<tr>
<td>y(0,2)</td>
<td>2</td>
<td>2</td>
<td>-y(0,2)</td>
</tr>
<tr>
<td>y(1,1)</td>
<td>2</td>
<td>3</td>
<td>-y(1,1)</td>
</tr>
<tr>
<td>y(1,1)-1</td>
<td>2</td>
<td>3</td>
<td>-y(1,1)+1</td>
</tr>
<tr>
<td>y(1,2)+1</td>
<td>2</td>
<td>3</td>
<td>-y(1,2)</td>
</tr>
<tr>
<td>y(1,2)</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>y(s-2,1)</td>
<td>2</td>
<td>s</td>
<td>-y(s-2,1)</td>
</tr>
<tr>
<td>y(s-2,1)-1</td>
<td>2</td>
<td>s</td>
<td>-y(s-2,1)+1</td>
</tr>
<tr>
<td>y(s-2,2)+1</td>
<td>2</td>
<td>s</td>
<td>-y(s-2,2)-1</td>
</tr>
<tr>
<td>y(s-2,2)</td>
<td>2</td>
<td>s</td>
<td>-y(s-2,2)</td>
</tr>
<tr>
<td>y(s-1,1)</td>
<td>2</td>
<td>s+1</td>
<td>-y(s-1,1)</td>
</tr>
<tr>
<td>y(s-1,1)-1</td>
<td>2</td>
<td>s+1</td>
<td>-y(s-1,1)+1</td>
</tr>
<tr>
<td>y(s-1,2)+1</td>
<td>2</td>
<td>s+1</td>
<td>-y(s-1,2)</td>
</tr>
<tr>
<td>y(s-1,2)</td>
<td>2</td>
<td>s+1</td>
<td>-y(s-1,2)</td>
</tr>
</tbody>
</table>

*Illustration 38 Error Function Aggregation - Essential Error Entries*
### 3.2.6 Parser Logic

The following illustrations show the parsing table dependent logic of the parser driver.

#### Deterministic Context Free Parser – Table Based Logic – 1

**Parsing Action Determination**

<table>
<thead>
<tr>
<th>Stack Element Index 0</th>
<th>Machine State Stack Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Element Index 1</td>
<td>Machine State Stack Data</td>
</tr>
<tr>
<td>. . . . . . . . . . . .</td>
<td></td>
</tr>
<tr>
<td>Stack Element Index c-2</td>
<td>Machine State Stack Data</td>
</tr>
<tr>
<td>Stack Element Index c-1</td>
<td>Machine State Stack Data</td>
</tr>
</tbody>
</table>

1) Parsing action =

\[
\text{action_table}[\text{current\_machine\_state}][\text{lookahead\_token\_number}]
\]

**Automaton Configuration**

\[ c = \text{stack\_count} \]

- Get current machine state from stack element with index c-1 and get current lookahead token number.
- Compute parsing action by equation 1.

*Illustration 39  Deterministic Context Free Parser - Table Based Logic - 1*
Deterministic Context Free Parser – Table Based Logic – 2

**Shift Action**

<table>
<thead>
<tr>
<th>Stack Element Index c-1</th>
<th>Stack Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index c</td>
<td>Stack Data</td>
</tr>
</tbody>
</table>

**Automaton Configuration**

- Initial Stack Count = c
- Final Stack Count = c+1
- Current Machine State
- Goto Table

1) new\_machine\_state = 
goto\_table[current\_machine\_state][lookahead\_token\_number]

**Shift Action Logic Flow**

Get current machine state from stack element with index c-1 and get current lookahead token number.

Compute new machine state by equation 1.

Push new stack element with index c on stack and increment stack count to c+1.

For stack element with index c set machine state to new machine state and set stack data to lookahead token data.

**Illustration 40 Deterministic Context Free Parser - Table Based Logic - 2**
Deterministic Context Free Parser – Table Based Logic – 3

Reduce Action

rule reduce action

Automaton Configuration

1) \( rrl = \text{rule}_\text{right}_\text{length}_\text{table}[\text{rule}_\text{reduce}_\text{action}] \)
2) \( rls = \text{rule}_\text{left}_\text{symbol}_\text{table}[\text{rule}_\text{reduce}_\text{action}] \)
3) \( \text{new}_\text{machine}_\text{state} = \text{goto}_\text{table}[\text{goto}_\text{machine}_\text{state}][rls] \)

Reduce Action Logic Flow

Compute \( rrl \) and \( rls \) by equations 1 and 2.

Get goto machine state from stack element with index \( c-rrl-1 \)

Compute new machine state by equation 3.

For stack element with index \( c-rrl \) set machine state to new machine state and set stack count to \( c-rrl+1 \)

Execute code generation to create reduce data from stack data of stack elements with indexes \( c-rrl \) to \( c-1 \) and set stack data of stack element with index \( c-rrl \) to reduce data.
Deterministic Context Free Parser – Table Based Logic – 4

Parser Driver Logic

Parser Driver Start
Initialize parsing stack by
clearing stack then pushing one
stack element with machine
state 0 on stack. Set stack
count to 1.

Get next lookahead token
number and data and determine
next parsing action.

Action = Shift
true
Execute shift action logic.
false

Action = Reduce
true
Execute reduce action logic.
false

Action = Accept
true
Input token stream accepted.
Return stack data in stack
element with index 1 as parse
object.
false

Action = Error
true
Execute error processing with
error function index = -action.
false

Execute conflict processing
with conflict function index =
action-base_conflict_entry.
3.3 Syntax Parsing Mode

A deterministic context free parser driver can be initialized to run in syntax parsing mode. This mode enables features of the parser driver to support direct stack element storage of allocated code generation objects and automatic deletion in response to syntax or code generation errors. The mode can simplify development of parsers that have the property of syntactically valid input implying successful code generation. Such parsers will accept input or incur an error determined only by the syntax of the input. The syntax parsing mode uses a default CStackElement class declaration that is incorporated into the build of the CPG application itself. The default class can be modified to accommodate any additional stack storage data members required. The default class declaration has the form:

class CStackElement
{
  public:
    enum ElementType
    {
      LiteralToken,
      Error,
      SingleChar,
      Scalar,
      CharArray,
      NonTerminal,
      Data
    };

    CStackElement() : type(LiteralToken), char_array(0), data(0) { }

    int state;
    ElementType type;
    int token;
    UScalar scalar;
    unsigned char *char_array;
    void *data;
    int reduce_action;
};

For syntax parsing mode the rule section code should have the following form:

    if (!data)
    {
      // When a rule reduction action is being processed the data
      // argument is zero. Code generation for the rule reduction
      // is performed in this block. For assigning allocated data or
      // objects using the syntax parsing mode members:
      //
      // for character array allocation
      stack_ptr[0].char_array = new unsigned char[n];
      stack_ptr[0].type       = CharArray;
      // for object allocation
When an error condition arises in syntax parsing mode allocated
character arrays or objects using syntax parsing mode members
are automatically deleted. The parser driver directly deletes
a character array allocation stored in the char_array member.
For an object allocation the parser driver calls the rule code
member function that allocated the object with an argument
that is the pointer to the object. This block should delete
the allocated object using the correct object type destructor.

delete (CObject1 *)data;

4 Parser Specification File Format

The CPG application parser specification file stores the grammars, code sections, modifiers, and
configuration parameters that completely specify automaton parsers and the code generation
procedures to create them. The command line program also uses the parser specification file
format. The format has the following grammar rule specification:

specification_file  ->  main_specifier
specification_file  ->  specification_file  main_specifier

main_specifier  ->  start_context
                 ->  terminal_entry_list
                 ->  error_token_entry_list
                 ->  nonterminal_entry_list
                 ->  rule_entry_list
                 ->  precedence_entry_list
                 ->  lexical_code_section
                 ->  terminal_code_section_list
                 ->  parser_code_section
                 ->  rule_code_section_list
                 ->  conflict_code_section
                 ->  error_code_section
                 ->  unit_rule_order_list
                 ->  configuration_list

start_context  ->  #CONTEXT\n   string_literal \n
terminal_entry_list  ->  #TERMINAL\n
terminal_entry  ->  terminal_entry_list  terminal_entry

terminal_entry  ->  index  type_spec  driver_spec  name  regexp \n
index -> integer

type_spec -> FILTER
type_spec -> TOKEN

driver_spec -> REGEXP
driver_spec -> LITERAL

name -> string_literal

regexp -> string_literal

error_token_entry_list -> #ERROR_TOKEN\nerror_token_entry_list -> error_token_entry_list error_token_entry

error_token_entry -> index name synchronization \n
synchronization -> integer

nonterminal_entry_list -> #NONTERMINAL\nnonterminal_entry_list -> nonterminal_entry_list nonterminal_entry

nonterminal_entry -> index name \nnonterminal_entry -> index name START \n
rule_entry_list -> #RULE\nrule_entry_list -> rule_entry_list rule_entry

rule_entry -> index left -> right \nrule_entry -> index * left -> right \n
left -> string_literal

right -> string_literal
right -> right string_literal

precedence_entry_list -> #PRECEDENCE\nprecedence_entry_list -> precedence_entry_list precedence_entry

precedence_entry -> index associativity_spec precedence_spec \n
associativity_spec -> LEFT
associativity_spec -> RIGHT
associativity_spec -> NONASSOCIATIVE

precedence_spec -> token
precedence_spec -> RULE:index

token -> string_literal
lexical_code_section -> #LEXICAL_CODE
lexical_code_section -> lexical_code_section #INCLUDE
lexical_code_section -> lexical_code_section #CLASS
lexical_code_section -> lexical_code_section #SOURCE
lexical_code_section -> lexical_code_section #CONSTRUCTOR
lexical_code_section -> lexical_code_section #DESTRUCTOR
line_list ->
line_list -> string_literal \
line_list -> line_list string_literal \
terminal_code_section_list -> #TERMINAL_CODE
terminal_code_section_list -> terminal_code_section_list
terminal_code_section -> TERMINAL:index
parser_code_section -> #PARSER_CODE
parser_code_section -> parser_code_section #INCLUDE
parser_code_section -> parser_code_section #CLASS
parser_code_section -> parser_code_section #SOURCE
parser_code_section -> parser_code_section #CONSTRUCTOR
parser_code_section -> parser_code_section #DESTRUCTOR
rule_code_section_list -> #REDUCE_CODE
rule_code_section_list -> rule_code_section_list rule_code_section
rule_code_section -> RULE:index
conflict_code_section -> #CONFLICT
conflict_code_section -> index state symbol
conflict_code_section -> action \
state -> integer
symbol -> integer
action -> integer
error_code_section -> #ERROR_CODE
error_code_section -> #ERROR_CODE
error_code_section -> #ERROR_CODE
error_code_section -> error_token_index \
entry -> integer
error_token_index -> integer
unit_rule_order_list ->
unit_rule_order_list -> integer
unit_rule_order_list -> unit_rule_order_list integer
configuration_list -> #CONFIGURATION
configuration_list -> configuration_list LEXICAL_AUTOMATON: bool 
configuration_list -> configuration_list LEXICAL_BASE_NAME: name 
configuration_list -> configuration_list LEXICALFINITE_STATE_MACHINE: 
  lexical_machine 
configuration_list -> configuration_list LEXICAL_INPUT_RANGE: 
  input_range 
configuration_list -> configuration_list PARSER_AUTOMATON: bool 
configuration_list -> configuration_list PARSER_BASE_NAME: name 
configuration_list -> configuration_list PARSER_TYPE: parser_type 
configuration_list -> configuration_list PARSER_ERROR_CALCULATION: 
  error_calculation 
configuration_list -> configuration_list PARSER_ERROR_NUMBERING: 
  error_numbering 
configuration_list -> configuration_list PARSER_AGGREGATION: 
  optimize 
configuration_list -> configuration_list GENERATE_TTY_CODE: bool 
configuration_list -> configuration_list REGEXP_MATCH_TABLE: bool 

bool -> TRUE
bool -> FALSE

lexical_machine -> REGEXP
lexical_machine -> SINGLECHAR

input_range -> 128
input_range -> 256

parser_type -> LR(1)
parser_type -> LALR(1)
parser_type -> SLR(1)

error_calculation -> NONE
error_calculation -> APPROXIMATE
error_calculation -> EXACT

error_numbering -> ONE
error_numbering -> STATE
error_numbering -> ENTRY

error_aggregation -> ONE
error_aggregation -> STATE
error_aggregation -> ENTRY

optimize ->
optimize -> UNIT_RULE_REDUCTION
5 Command Line Program

A command line version of the CPG application is available for incorporating CPG parser generation into application build and configuration control systems. The program is invoked by the following terminal command line:

cpgcmd [ -d directory_path ] [ -p ] [ file_name ]

- `-d directory_path` - specifies target directory for generated source files, default is current directory
- `-p` - directs program to output LR(k) state machine information
- `file_name` - name of parser specification file for input, default is standard input

6 References

The Art of Computer Programming
Donald E. Knuth
Addison-Wesley-Longman 1998

Volume 1: Fundamental Algorithms
ISBN 0-201-03809-9

Volume 2: Seminumerical Algorithms
ISBN 0-201-03822-6

Volume 3: Sorting and Searching
ISBN 0-201-89685-0

Effective C++: 50 specific ways to improve your programs and designs
Scott Meyers
ISBN 0-201-56364-9
Addison-Wesley 1992

More Effective C++: 35 new ways to improve your programs and designs
Scott Meyers
Addison-Wesley 1996

The Design and Analysis of Computer Algorithms
Alfred V. Aho
John E. Hopcroft
Jeffrey D. Ullman
ISBN 0-201-00029-6
Addison-Wesley 1974

Parsing Theory Volume I
Languages and Parsing
Seppo Sippu
Eljas Soisalon-Soininen
ISBN 3-540-13720-3
Springer-Verlag 1988

Parsing Theory Volume II
LR(k) and LL(k) Parsing
Seppo Sippu
Eljas Soisalon-Soininen
ISBN 3-540-51732-4
ISBN 0-387-51732-4
Springer-Verlag 1990

The Theory of Parsing, Translation, and Compiling
Volume I: Parsing
Alfred V. Aho
Jeffrey. D. Ullman
ISBN 0139145567
Prentice-Hall 1972

The Theory of Parsing, Translation, and Compiling
Volume II: Compiling
Alfred V. Aho
Jeffrey D. Ullman
ISBN 0139145648
Prentice-Hall 1973

Algebraic Theory of Automata
Abraham Ginzburg
Library of Congress Catalog Card Number 68-23492
Academic Press 1968

Formal Languages
Arto Salomaa
Library of Congress Catalog Card Number 72-88356
Academic Press 1973

Semirings, Automata, Languages
Werner Kuich
Arto Salomaa
ISBN 3-540-13716-5
Springer-Verlag 1986

Regulated Rewriting in Formal Language Theory
Jurgen Dassow
Gheorghe Paun
ISBN 3-540-51414-7
ISBN 0-387-51414-7
Springer-Verlag 1989

Classical Recursion Theory
Piergiorgio Odifreddi
ISBN 0-444-89483-7
Elsevier Science B.V. 1992

Computability and Unsolvability
Martin Davis
Dover Publications 1982

Introduction to Formal Languages
Gyorgy E. Revesz
Dover Publications 1991

The Annotated C++ Reference Manual
Margaret A. Ellis
Bjarne Stroustrup
ISBN 0-201-51459-1
Addison-Wesley 1995

The C++ Programming Language
Bjarne Stroustrup
ISBN 0-201-88954-4
Addison-Wesley 1997

Standard Template Library Programmer's Guide
Silicon Graphics 1999