INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

UMI®

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

A Syntax-Directed Compiler/Compiler Emitting LR(1) Object-Oriented Code

David Bone

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September 1998

ABSTRACT

A Syntax-Directed Compiler/Compiler
Emitting LR(1) Object-Oriented Code

David Bone

This is a study into applying Object-Oriented principles to a LR(1) compiler/compiler, its grammar language, and the emission of output tables containing class objects. The grammar's entities — rules, subrules, and terminals — are viewed as class structures defined in the C++ language. Woven into the grammar's language are constructs supporting syntax-directed C++ code. Object-Oriented design decisions are discussed in supporting the grammar's symbols and various bottom-up behaviors, the definition of the grammar language documented, and executed grammar examples illustrated with commentary.

Contents

List of Figures

Chapter 1

Preface

Here we are in 1998 approximately 30 years after the development of parsing techniques and a new programming paradigm is object-oriented languages — some examples being Eiffel, C++, and Smalltalk. But what about the emitted tables produced from compiler/compilers used to translate these languages? These are the tools that are slaying the language dragon but there is nothing object-oriented in their outputs. Using these tools leads to a loose co-ordinate set of procedures to achieve parsing. The following is a brief overview of my motives to investigate an object-oriented approach to table-driven parsing — in particular shift-reduce parsing with syntax-directed translation.

The basic benefit from grammars and their compiler/compilers is the fast manufacture of recognizers with ambiguity detection. With special tools like YACC [7] and LEX [10], compiler writers deliver products faster and with confidence knowing their grammar describes the language to be translated; at least that is the idea behind the use of grammars with a compiler/compiler. Unfortunately due to a grammar's coarseness in defining the new language, these tools still lead to difficulties. In

Stroustrup's book "The Design and Evolution of C++" published in 1994 [14], he quietly explains his frustration with YACC and the bottom-up approach to compiling the C++ language:

> In 1982 when I first planned Cfront, I wanted to use a recursive descent parser because I had experience writing and maintaining such a beast, because I liked such parsers' ability to produce good error messages, and because I liked the idea of having the full power of a general-purpose programming language available when decisions had to be made in the parser. However, being a conscientious young computer scientist I asked the experts. Al Aho and Steve Johnson were in the Computer Science Research Center and they, primarily Steve, convinced me that writing a parser by hand was most old-fashioned, would be an inefficient use of my time, would almost certainly result in a hard-to-understand and hard-to-maintain parser, and would be prone to unsystematic and therefore unreliable error recovery. The right way was to use an LALR(1) parser generator, so I used Al and Steve's YACC[Aho,1986].
>
> For most projects, it would have been the right choice. For almost every project writing an experimental language from scratch, it would have been the right choice. For most people, it would have been the right choice. In retrospect, for me and C++ it was a bad mistake. C++ was not a new experimental language, it was an almost compatible superset of C — and at the time nobody had been able to write an LALR(1) grammar for C. The LALR(1) grammar used by ANSI C was constructed by Tom Pennello about a year and a half later — far too late to benefit me and C++ .

In January 1997, ACM SIGPLAN Notice's article "Programming Languages: Past, Present, and Future" [17] where Peter Trott interviews a variety of computer scientists intimate in language processing, Dr. Stroustrup again restates his frustration with bottom-up processing and its over-exaggerated expressive powers (I apologize if this is an overstatement).

So back in 1993 before the above observations were read, my preliminary thoughts to bottom-up parsing were the following. LR(1) parsers should be the norm. With LR(1) being the most powerful recognizer of deterministic grammars [8], so why

choose a lesser recognizer to compile languages? Having very little experience in applying bottom-up parsers to real projects, I thought LR(1) resolution along with object-oriented techniques would ease the limitations expressed. By making symbols of a grammar act as classes as defined in the C++ language, and adding other object-oriented ideas to the emitted tables, this would convince people that table-driven parsers should be the off-the-shelf way to go in compiling languages.

In this thesis we explore an objected-oriented framework applied to a compiler/compiler's emitted tables representing the compiled grammar and its entities — terminals and nonterminals with their associated syntax-directed code. It evolves the class definitions from their abstract base classes through to the derived classes required in supporting a table-driven compiler. As the object-oriented paradigm separates definition from implementation, the thesis only develops the class definitions.

Throughout the development, snippets of the grammar language are used to highlight the developing section. The local development explains the relationship expressed in the language's structure being exposed. The reader is not expected to be completely familiar with the grammar language but for the curious, the appendix defines the grammar language. Principally, keywords tie together the grammar's entities with their C++ class names. Various code directives are used to inject C++ code into the defining class structure and its implementation companion.

The thesis consists of the following chapters in its development. Chapter two reviews the theories of both a context-free grammar and the C++'s objected-oriented paradigm. The terminology for each discipline is presented along with their components. Within the object-oriented paradigm, other class facilities are brought out which augment the LR(1) bottom-up parsing environment. Chapter three, the mar-

riage of Object-oriented paradigm with Context-free Grammar is the essence of the thesis. Object-oriented principles are applied in defining a grammar's symbols for use in a finite-state-machine, or FSM. Chapter four presents the finite-state-machine with its ready-made C++ class definitions. A grammar example drawn from the appendix is used to illustrate the generated LR(1) table. Chapter five discusses the pushdown automata with its class definitions. The support classes are presented leading into an experimental parser. The "firing off" mechanism for a symbol's syntax-directed services is commented-on tying together "how the services" are carried by the grammar's symbols, and "how the actions" are actual executed by the parser. Chapter six talks about future research drawn from my experiences using the object-oriented output of the compiler/compiler. The final chapter summarizes the thesis's development. Appendix A documents the grammar's language, appendix B details the IDOW pattern, and finally, appendix C gives examples of lexical and syntactic grammars, the emitted terminals' classes with its FSM class, and a grammar showing the parser's tracing capability for some input.

To aid the reader, the following typesetting convention is used throughout the thesis:

- *italics type* indicates (abstract) base classes, and a class's internal data variables and procedures
- **bold type** indicates concrete symbol classes usually derived from base classes
- examples of C++ code use a fixed slightly smaller font with the exampled code indented from the left margin.

For the reader who is familiar with the object-oriented discipline, a fast look at the class diagrams within the thesis will quickly orient one to the relationships being expressed throughout — within the grammar language, within the emitted parser's

FSM table, and associated syntax-directed methods. To paraphrase an old adage — a picture is worth a thousand curses or is it a thousand curses is worth one class? To spice up a rather dry subject, bits of humour and personal comments on my trek through the project are intertwined.

Chapter 2

Theory

The basic objective to the chapter is a review of terms used throughout the thesis within the context of context-free grammars(CFG) as expressed in [2]. A grammar's structural make-up is defined without going into details of language generation. I assume the reader is familiar with the capabilities of a compiler/compiler and has some knowledge of the algorithms used to compile a context-free grammar into a FSM [3] [12]. The grammar language, an extended Backus-Naur Form (EBNF) [11], is completely documented in the appendix. Deviations from the Backus-Naur Form (BNF) will be addressed within this chapter. These slight variations are done for the economy of expression.

As to the Object-Oriented paradigm, also referred to as OO, the terms developed are what's used in its literature; their definitions are as supported within the C++ language. C++ examples will illustrate these support features which are heavily used in the class definitions generated from the compiler/compiler.

2.1 Context-free Grammar

A Context-free grammar G is composed of a 4-tupple (N, Σ, P, S). The symbols Σ, N, P, and S are, respectfully, the *terminal* alphabet, the *nonterminal* alphabet, *productions*, and the *start symbol* which is also known as the *goal symbol*. N, Σ, and P are finite sets; N and Σ are disjoint: there are no common elements. The symbol V stands for the vocabulary of $\Sigma \cup$ N. An element of productions P has the form:

$$x \rightarrow y$$

where $x \in N$ and y is a string of symbols in V, and the $\rightarrow$ expresses the mapping between the string of x into the string of y. S is the start symbol in N. The following is CFG's restrictions place on x and y: x has only one element from N making up its string of symbols, and y is a string of symbols from V including the empty string ϵ. To facilitate the description of a production of form $x \rightarrow y$, the following terms describe the production's structural parts:

- x will be known as the *left-hand-side* of the production(*lhs*). It also takes on other synonyms of a *production rule*, or just *rule*.
- y will be known as the *right-hand-side*(*rhs*) of the production. It will also be addressed as a *production's subrule*, or plain *subrule*.

Throughout the balance of the thesis, these terms are used extensively in the development of classes from a grammar's structural perspective.

Below is an example of a CFG:

$$G = (\{S, E, T, F\}, \{+, *, -, /, 1, 2, 3, (,)\}, P, S)$$

$N = \{S, E, T, F\}$
$\Sigma = \{+, *, -, /, 1, 2, 3, (,)\}$

where the productions P are:

1. $S \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow E - T$
4. $E \rightarrow T$
5. $T \rightarrow T * F$
6. $T \rightarrow T / F$
7. $T \rightarrow F$
8. $F \rightarrow (E)$
9. $F \rightarrow 1$
10. $F \rightarrow 2$
11. $F \rightarrow 3$

This grammar represents arithmetic expressions using numbers 1, 2, and 3. Though this is not a requirement of P, so that the productions can be individually referenced, each production is numbered in the example above and productions with the same left-hand-side are grouped on the same line for discussion sake.

From this formal definition of a CFG, two modifications are made to the grammar in the name of economy. Productions having a common left-hand-side are grouped together with the left-hand-side being expressed once followed by the multiple $\rightarrow$ y forms. From the grammar above, the enumerated productions 8. – 11. having F as the left-hand-side would be written:

$$
\begin{aligned}
F &\rightarrow (E) \\
&\rightarrow 1 \\
&\rightarrow 2 \\
&\rightarrow 3
\end{aligned}
$$

This variation describes alternatives. The second savings is obtained by using productions to define N, and S. The left-hand-side of the first production in P is considered the start symbol S. All left-hand-sides in P make up N. From this, the EBNF language tupple is (Σ, P) from which N and S can be inferred.

2.2 Object-Oriented Paradigm

This section provides a thumbnail sketch of OO's principles and components used in the C++ language. The aim is to provide the user with the OO mechanisms used and not to be a compendium of C++ language features. I assume the reader has a working knowledge of the C++ language and its constructs as documented in [4] [15]. The thesis reviews these OO features of C++ as applied to the symbols of a grammar emitted as classes. The examples are simple and highlight the C++ mechanisms used in the emitted class definitions. To note, for the syntax-directed code, all of C++ language features can be used — classes, access control, friends, multiple inheritance, exceptions, and templates. For well authored articles and discussions using C++ and OO, these magazines are recommended reading:

- C/C++ Users Journal
- Dr. Dobb's Journal
- C++ Report
- JOOP - The Journal of Object-Oriented Programming

Within the thesis, the following are OO terms used with their benefits:

- *Classes* — packaging agents. They act like a cell's enclosing membrane containing these components and benefits:
  - atomic and derived data types. Atomic data types are the computer's native data structures — character, integer, floating point number, etc. A derived data type is a class. Using the "hasa" relationship, a class can be built containing other classes.
  - procedures or methods contained inside the class definition. This eliminates loosely assembled definitions of data types and their procedures trying to act as one unit.

- *encapsulation* lessens global name space pollution. A class reduces and can hide its components from the program that uses it. This should lead to better program maintenance.

- *Inheritance* — the extension of a class definition. Inheritance acts as a stretching force using the "isa" relationship. For example, a dog "isa" an animal. It is also known as a class *deriving* from another class. This newly formed class inherits the data structures and the procedures of the attached to class.

- *Multiple inheritance* — a bridging force. It combines or joins two or more unrelated classes into an enclosing class acting as one unit. Each inherited from class gives its controlled benefits to the newly formed class for its own use.

- *Access control* — who is allowed to access what within a class. Control covers both atomic and derived data types, and use of inheritance. C++ has *public*, *protected*, and *private* access control.

- *Polymorphism* — dynamic run-time procedures. Procedures that are resolved at run-time depending on the class object giving variation in run behavior. This works in hand with inheritance. Depending on the language used, abstract base classes can enforce a uniform set of behaviours. They are also known as virtual behaviours.

- *Strong data typing* — compile time detection. Errors are immediately reported at compile time instead run time. Strong data typing is not illustrated as this is inherently built into the C++ language.

2.2.1 Class Packaging

This example shows how the class defines its name and contents. Further into the thesis, the mapping of the grammar's symbol into its C++ class name is discussed. The compiler writer provides the class name within the EBNF.

```
class Canimal{// class definition for Canimal
 public:                              // access control - allows anyone access
                                      // within its domain
  string*              id();
  virtual              ~Canimal();  // destructor
  virtual const char* sound() = 0; // an abstract polymorphic procedure that
                                      // must be defined by the inheriting class
```

```
 proctected:                          // access control on constructor forces
                                      // the class to be derived from
  Canimal(string* Name);              // contructor

 private:                             // access control - forces data hiding
  string*     id_;
};
```

From the C++ comments within the class definition, as one can see, the class defining capabilities covers efficiently access control to hid data or procedures, polymorphic procedures that can be made abstract, and control over how the class is to be used. This is an elegant way to package data and procedures, and to regulate use. Choice of a class name by a class designer can be as cryptic or as meaningful as one deems fit. *class* packaging is succinct which is what I am trying to achieve with a grammar's entities emitted as classes for compilation.

2.2.2 Inheritance

Used extensively in the compiler/compiler's emitted classes, inheritance provides the "isa" relationship to all the grammar's symbols translated into C++ classes. It does two things: the class inherits the use of the class derived from, and the new class provides its own spin on functionality. In view of the grammar's symbols, the *base class* — derived from class — enforces common symbol behaviours and polymorphism provides dynamic behaviours in various contexts. The simple example below shows the class **Cdog** deriving from the abstract base class **Canimal** through the use of C++'s *inheritance list.* It also shows defaulted parameters in its constructor: in this case, the default dog is a Poodle having a ferocious sound "grrrr". The polymorphic procedure **sound()** is defined.

```
class Cdog
  :  // indicator starting an inheritance list
  public Canimal // a component in the inheritance list
                 // with explicit access control
                 // where Cdog inherits from Canimal
{
 public:
  ~Cdog();
  Cdog(const char* Name = "Poodle",const char* Sound = "grrrr");
  const char*     sound(); // defined polymorphic procedure
 private:
  const char*      sound_;
};
```

If there is more then one class in the inheritance list, the class is said to be using *multiple inheritance.* This is an over simplification of inheritance but illustrates how the emitted symbol classes will be built. Left out of this explanation is shared base classes with its access control available to the C++ programmer. Please see reference [15] for a complete development of C++'s inheritance.

2.2.3 Polymorphism

Base classes are the blocks from which the grammar symbols, action classes, and patterns will get built. Polymorphism creates abstract base classes which allow the class designer to enforce regular and dynamic behaviours. It is this abstractness that gives enriched facilities to the class designer. Creatively, the designer has complete command of enforcement: from default behaviours, optional behaviours, to no choice must-do behaviours. It is polymorphism that prisms-out the software spectrum of variation. All of this is done at run-time. From the example above, because **Canimal** uses abstract polymorphism, the deriving class **Cdog** is forced to define and implement the abstract procedure **sound()**. This abstract procedure from **Canimal**

spawns the menagerie of sounds from all its derived animals.

2.2.4 Life Cycle of a Class

When a class is used in a program, an object of that class is created. It stays in existence until its dies. The C++ framework for the birth/death cycle of a class's object is its *contructor(s)* and *destructor* constructs. C++ language allows explicit and implicit execution of a class's contructor and any derived-from classes with their constructors. It always builds the class's object from base-to-derived order. In death, the destructors are run in derived-to-base class order, that is in opposite order to the creation of the constructors. These are discrete execution points of the class's object. Knowing this, why not allow the compiler writer use of these discrete logic points for his own purposes? Within these logic points, all other class procedures can be run. Looked at this way, the constructor/destructor encapsulates all other activities of the class.

2.2.5 Events

Like any developer of software, I prudently try to add functionality with minimum cost. Within the life/death cycle of a class, there is a lot of room to impose functionality. Now lets step back and reflect a little on what additional behaviours a compiler writer might need with the grammar's symbols. My first thought is tracing of the symbol while parsing takes place or receiving the parsing environment for local semantic processing. Well, use of constructors as a container to receive injected functionality leads to the problem of distribution. Because a class can have many constructors, how does one centralize a common execution point? My solution is to

not distribute the commonality throughout each constructor but to add a procedure *op* parameterized with the parse environment to each class and to ensure that this common procedure gets executed right after the constructor is executed. As there is only one destructor to a class, this becomes a common stop for additional functionality at the tail end of a class's life. And so, code directives of the grammar language can be used to inject the compiler writer's code into these end points.

A more deeper question arises when dealing with a grammar's symbols. Modern languages like C++ and Java have grammars with hundreds of symbols. What options does a compiler writer have in corralling these symbols and associating code to be executed in these contexts? Is there a nice way to corral these symbols such that the following can be done?:

1. to traverse the symbols within a context. For example, the traversal of an abstract syntax tree emitting compiled code.
2. during the traversal of the symbols, polymorphic code associated per symbol gets executed. Each traversal context should support different functionality.
3. the code to be executed per symbol should be encapsulated and should be imported into the to-be-traversed context.

Yes there is an elegant way to do this within the OO paradigm: use of Patterns. Such a pattern is developed in this thesis — Iterator-Dispatch of Objects and their Works(IDOW). It is a variation on some of the patterns described in "Design Patterns Elements of Reusable Object-Oriented Software"[5]. Basically the support procedure is part of the symbol base class emitted. This procedure is *accept_workers*. All symbols of a grammar — terminals, nonterminals, and productions — are built with this functionality. Throughout the thesis, different contexts needing the IDOW support are developed. The abstract mechanisms driving the IDOW pattern is fully documented in the appendix.

So, inherently built into a class will be its variations on birth, improvisations during life, and its death throes — *op*, *accept_workers*, and use of the destructor.

2.3 Summary

To recapitulate, using all of these features of C++ provides a very powerful defining capability to encapsulate the grammar's components. Throughout the following chapters, these three points are illustrated:

1. the birth-life-death cycle of a class
2. inheritance — abstract base classes enforcing behaviours
3. polymorphism — dynamic variations in behaviors

The life cycle originates all the class designs emitted. Code directives of the grammar language use this pattern to inject code into the constructors, destructor, and ancillary methods of the class. Base classes build real classes through inheritance, with the syntax-directed code being put into these newly formed classes. During the compiler/compiler implementation, Microsoft Foundation Class Library (MFC) was used; some of MFC's classes used are CObject, CString, and various container classes [1]. These classes show up in the examples developed later. In the future, as the Standard Template Library(STL) has stabilized, the emitted class definitions can be converted to use STL.

Chapter 3

The Marriage of OO with CFG

As an introduction to this chapter, I am going to ask four related questions surrounding different stages of syntax-directed parsing that a compiler writer reflects on:

1. How does one intelligently certify a grammar — both in the reading of the source definition and at run time?
2. How does the compiler writer relate to the emitted code?
3. How does the emitted LR(1) table relate to the grammar definition?
4. How does the syntax-directed code use the defining grammar's objects: rules and terminals?

At whatever stage the compiler writer is in, each stage should relate back to the grammar's definition. All terms used in the definition should be embedded in the other stages. In the world of program debuggers, the running program is mapped back to the source code along with its run time values. In grammar debugging, one should relate back to the definition and if inclined, use the compiler's debugger to examine the emitted code at run time.

From examining past bottom-up implementations, one principle issue surfaced which is rules and terminals have parallel definitions: definitions within the grammar, and definitions supplied for the emitted programs. The accompanying syntax-directed code, where this code is injected into the execution flow and how it is implemented, became the expertise of the bottom-up parsing guru. None of the definitions were integrated; they were spread amongst global variables and constants. The syntax-directed code was difficult to assess when debugging. Due to the computer language used and space efficiency, the implementation process obscured the grammar's vocabulary and meaning. Table fiddling, due to grammar inadequacies, paid dearly in understanding how the grammar was implemented.

So we can conclude that the vocabulary boundaries between the rules and their terminals, and the syntax-directed code must be integrated into a framework solving the above observations with no distortion. Clarity and grammar certification are number one in importance. The objects used must be synthesized across all compiling boundaries so that the compiler's code is readable. Now the fundamental issue becomes "how to define and translate a grammar into emitted entities with attributes and related personalities having a volubility?". Clarity of definition throughout all phases must be consistent. The names in the grammar must be identically expressed in the emitted code. Their reference in the syntax-directed code, again, must be identical. Their definitions should be centrally defined: they should not spread across various global variables and constants. Multiple grammars used together must work in harmony and if possible, each term within each grammar should be a self-contained definition. Modularity should be supported in the syntax-directed code — global awareness should be minimized: its local run environment should be imported via

passed parameters.

My solution to this problem is use an object-oriented discipline by applying class definitions to the compiling objects at hand. This means that terminals, rules and their subrules should all be defined by class definitions built from abstract base classes. Class definition addresses the coherence of items used in all compiling parts along with syntax-directed code's association with and integration into these classes. To lessen code shock, the syntax-directed code and the class definitions produced by the grammar all use the C++ language. The grammar language itself models the C++ class construct. The grammar's supporting constructs are designed to fit the C++ class's life cycle which accompanies each entry in the grammar's vocabulary. A class name is chosen by the compiler writer for each rule and terminal. Naming control by the compiler writer gives the C++ code its freedom; it is not held to implementation restrictions (unless one feels C++'s naming convention restrictive).

Because the grammar is the source, I wanted the compiler writer to stay within the grammar definition when debugging his code. To do this, the grammar's rules and terminals each have a symbolic key; this key is incorporated into the emitted object's class definition; in the case of the rule, it is also the class name. The symbolic key becomes the translation between the running object and the grammar's defining vocabulary which aids the debug process with its automatic action service facilities. Its value is a constant literal equated in the class's contructor id parameter. The symbolic key is the handle used in the LR(1) table lookup of its objects.

Molded into this framework is the syntax-directed code which seamlessly injects into the objects being compiled: in the grammar, code directives achieve the integration. For multiple parse staging using different grammars: one parsing stage's output

becomes the following parsing stage's input. The lexical and syntactical stages use a centrally defined terminal vocabulary whereby tokens can be manufactured over time: terminals built in previous grammars can become assembly units for other more sophisticated meta-terminals being built in future parses.

But what about using these emitted objects in different contexts? Tracing of tokens, copying, or archiving of tokens are examples of such needs. To assist the compiler writer, a skeleton IDOW pattern is emitted. The abstract class definition becomes the building block in defining other specific actions.

A taste of parsing with objects is presented using multiple parse passes:

```
#include "clr_pass1_lexer.h"
#include "clr_pass2_lexer.h"
Get_chr gt;                    // raw character dispenser
Token_container pass1_tokens;
Clr_pass1_lexer pass1_table;  // parse table
Parser pass1_parser(&pass1_table,&gt,&pass1_tokens);
pass1_parser.parse();
// Example: IDOW pattern for tracing of tokens
    clog << "start trace of pass1 tokens\n";
    Cprint_terminals_dv print_terminals(clog);
    Cterminal_token_container_iterator
    pass1_tokens_iter(&pass1_tokens,&print_terminals);
    pass1_tokens_iter.access();
// Example: pass1 tokens piping into pass2 parsing
Token_container pass2_tokens;
Clr_pass2_lexer pass2_table;  // parse table
Parser pass2_lex_parser(&pass2_table,&pass1_tokens,&pass2_tokens);
pass2_lex_parser.parse();
// re-use of IDOW pattern on token dump
    clog << "start trace of pass2 tokens\n";
    Cterminal_token_container_iterator
    lr_pass2_tokens_iter(&pass2_tokens,&print_terminals);
    pass2_tokens_iter.access();
```

This example shows two grammar staging using the IDOW pattern to print-dump their tokens. The C++ comments annotate the example's sections.

The following subsections define my object-oriented approach to the grammar's vocabulary: rules and terminals. The use of these symbols as translated into C++ classes is developed along with their predefined actions. The syntax-directed operations using these symbols, the mechanism to access these symbols through objects, and extensions available to the compiler writer to enhance these symbols are also evolved. The development of the subject uses a top-down approach starting with the abstract base classes that are universal to all symbols followed with the concrete class definitions building on these abstract classes. Throughout the development, the skeletal dynamics of these symbols are explained in answer to the four questions posed at the beginning of this section.

3.1 Symbols

As a reminder from CFG defined in chapter 2, productions are rules. Structurally, the right-hand-side of a rule is its subrule. The make-up of productions are symbols drawn from the terminal and nonterminal alphabets. Nonterminals and the start symbol can be deduced from the productions. In the development of symbols, the terms used will be drawn from the productions and terminals with the implicit understanding that the nonterminal symbols are inferred. Interchangeably, rules and nonterminals will be used as nonterminal symbols. When the structure of the production is important, its context will be explicitly expressed.

In my class design of a grammar's symbols, there is one abstract class spawning all the terminal and nonterminal symbols , and two companion abstract classes providing syntax-directed services for these symbols. As previously developed, a base class acts as a common repository of data and functionality. These classes are:

- *CAbs_lr1_sym* base class for all grammar symbols: rules and terminals
- *CAbs_lr1_sym_workers* support class for the IDOW pattern service
- *CAbs_s_r_rtn* action class for stacking services

The companion classes are part of the *CAbs_lr1_sym* class using the "hasa" relationship. These classes are now explained.

3.1.1 *CAbs_lr1_sym* Abstract Base Class: Mother Nature to All Symbols

CAbs_lr1_sym class raison d'être is:

- to provide a symbolic identity used in tracing and as a key in the LR(1) table lookup of objects.
- to contain action classes supporting the parsing activities of shift and reduce stacking operations, and lookahead symbol used in the reducing operation.

Its C++ class definition is:

```
class CAbs_parser;
class CAbs_lr1_sym: public CObject{
 public:
  CString*       id();
  const char*    sym_id();
  virtual void   op(CAbs_parser* Parser_env)=0;
  void           look_ahead_rtn(CAbs_s_r_rtn* Look_ahead_rtn);
  CAbs_s_r_rtn*  look_ahead_rtn();
  void           shift_rtn(CAbs_s_r_rtn* Shift_rtn);
  CAbs_s_r_rtn*  shift_rtn();
  void           reduce_rtn(CAbs_s_r_rtn* Reduce_rtn);
  CAbs_s_r_rtn*  reduce_rtn();
  virtual        ~CAbs_lr1_sym();
  virtual void   accept_workers(CAbs_lr1_sym_workers* Worker);
 protected:
  CAbs_lr1_sym
    (CAbs_s_r_rtn* Look_ahead_rtn
    ,CAbs_s_r_rtn* Shift_rtn
    ,CAbs_s_r_rtn* Reduce_rtn
    ,const char*   Id);
```

```
 private:
  CString*       id_;
  CAbs_s_r_rtn*  look_ahead_rtn_;
  CAbs_s_r_rtn*  shift_rtn_;
  CAbs_s_r_rtn*  reduce_rtn_;
};
```

Two routines are special and were described in OO's Events of the previous chapter: *op* and *accept_workers*. The *op* routine always gets executed right after the constructor ends — injected code is placed inside it by the "op" language directive. It is virtually declared without an implementation which forces the deriving class to define and to implement. It is a constant execution point between the constructor, and the symbol's destructor. Passed to it is a *CAbs_parser* object defining the running parse environment. The compiler writer's code for *op* uses the cast-operator to turn the abstract parser into a specific class — usually the LR(1) parse table class that is emitted from the language's "fsm" construct which is described in a future chapter. Between the *op* and destructor events, user code routines execute along with the parser's stacking routines associated with the symbol. *op* also has a stack tracing macro included in its implementation. This execution point is applied to all emitted classes where there is an "op" language directive.

The *accept_workers* routine supports the IDOW pattern. It is the gateway for the symbol to execute code-specific routines imported into it. Due to the base class passed parameter, only one general action can occur. Because the grammar defines specific rules, the parameter must be abstract specific to the grammar and not to an abstract singularity. The workers are local to each defining grammar and hence each grammar must have their distinct abstract middlemen classes. Because I found this pattern very useful, as a courtesy to other compiler writers using this tool I have

included it in all the classes emitted. It is there for the using.

Included into the base symbol class are bottom-up action objects which take on the names of *shift_rtn*, *reduce_rtn*, and *look_ahead_rtn*. These action objects are the dietary supplements for bottom-up behaviors used in various contexts — they are execution points to hang compiler writer's code.

Figure 3.1 shows the class hierarchy of the grammar symbols: the base class with its deriving abstract middlemen classes through to the actual real symbol class definitions. The middleman provides a layer between generality and specifics. This line divides the run behavior from everyone acts the same to everyone having an individual personality. Their abstract C++ class names are built from the "fsm-class" language construct.

The *id_* variable is the symbolic value used in tracing of a grammar at execution time and in the LR(1) table lookup. Its access function has the same name. The other variables are event driven classes described later — *look_ahead_rtn*, *shift_rtn*, and *reduce_rtn*.

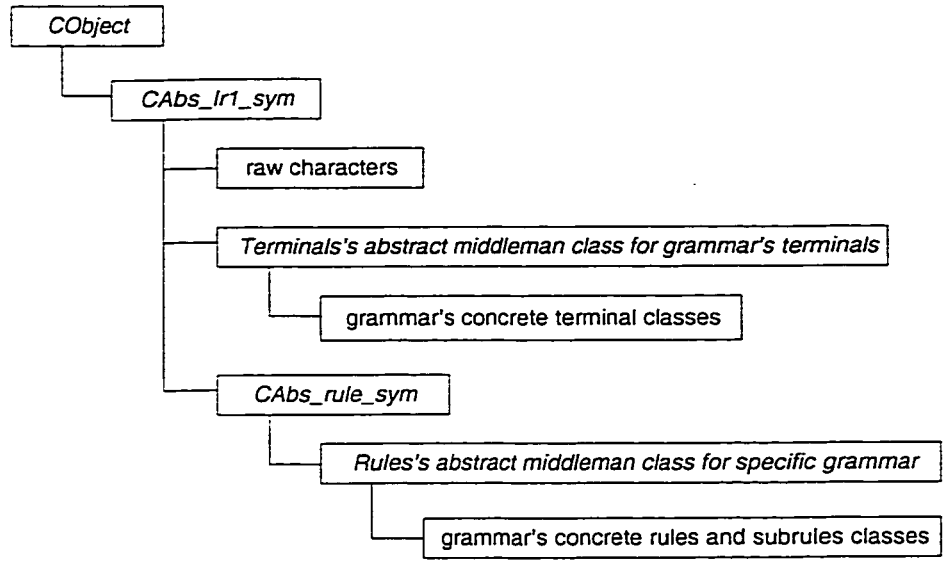

Figure 3.1: Class hierarchy of symbols

3.1.2 *CAbs_lr1_sym_workers* Base Class for IDOW Pattern Support

The *CAbs_lr1_sym_workers* class supports the IDOW pattern having this definition:

```
class CAbs_lr1_sym_workers: virtual public CObject{
 public:
  CAbs_lr1_sym_workers();
  virtual ~CAbs_lr1_sym_workers();
  virtual void general_contracting(CAbs_lr1_sym* Object);
};
```

The "virtual public CObject" phrase marries the dispatch part (using double dispatch is defined in the IDOW Pattern appendix) of the IDOW pattern with the pool-of-

workers defined here. Multiple inheritance is the "isa" relationship joining these two entities together. This is C++'s way of doing it.

Its *general_contracting* routine is the does-all facility for the IDOW pattern. The passed object *CAbs_lr1_sym* is a nondescript entity requiring no specialization. An example of a general IDOW pattern is the disposal of characters whom are general class objects building up specialized tokens. These general character objects call *general_contracting* which just deletes itself. For specialization, each grammar's middlemen classes are derived from this class. Throughout the following sections, specialization within various contexts is talked about using Figure 3.2 class hierarchy. The IDOW appendix explains fully this pattern with all its linked parts.

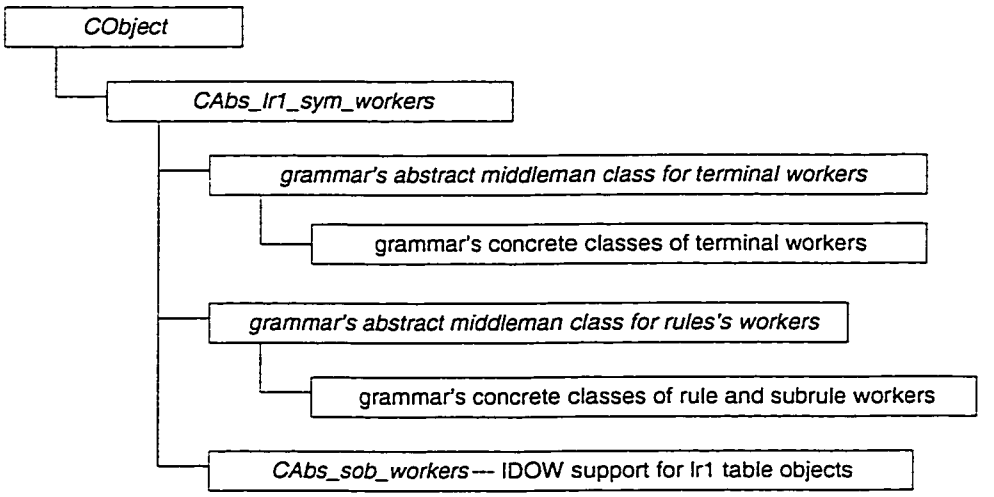

Figure 3.2: IDOW's workers class hierarchy

3.1.3 *CAbs_s_r_rtn* Abstract Base Class for Shift, Reduce, Lookahead Action Services

This abstract class provides syntax-directed services for the symbol when the parser does a specific operation: two of the actions, shift and reduce, are activated when the symbol is put on or taken off the stack; the lookahead activity executes when the terminal symbol is the boundary symbol used by the parser's reduce operation when collapsing the right-hand-side of a production into its left-hand-side rule. The "s_r_" in its name are mnemonics for the shift and reduce actions. Its definition is:

```
class CAbs_s_r_rtn{
 public:
  CString*    id();
  const char* sym_id();
  virtual void op(CAbs_lr1_sym* Sym, CAbs_parser*  Parser_env)=0;
  virtual ~CAbs_s_r_rtn();
 protected:
  CAbs_s_r_rtn(const char* Id);
 private:
  CString* id_;
};
```

CAbs_s_r_rtn has the same defining template as the *CAbs_lr1_sym* class described previously. A symbolic key identifies itself for tracing purposes. Its *op* routine receives two parameters:

1. the symbol associated with this action class
2. the parser's run environment. The top symbol on the parse stack is the same symbol passed in the first parameter. It is passed as a convenience to the compiler writer.

All real action classes derive from this base class which is shown in Figure 3.3. They are associated with their rule or terminal class definition by inclusion.

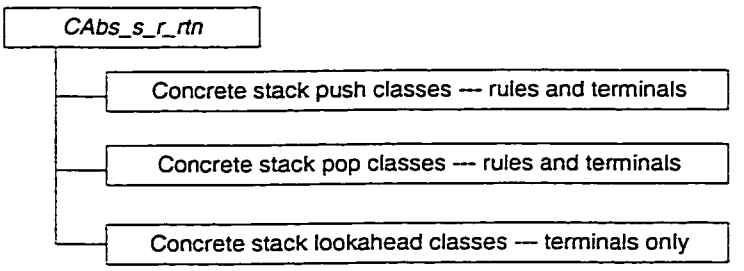

Figure 3.3: Class hierarchy of stack services

In the appendix A — The Grammar Language, these language action facilities are fully documented under "Terminals — Individual Components" and "Rules Section". How useful this base class is will be judged over time. Whether the derived services are too general within the context being run — like the lookahead service which only sees the parsing stack before the reducing of the subrule into its rule, I felt automatic "firing off" mechanisms for the compiler writer should be designed into the language. This is just one such setting.

3.2 Terminals

To discuss terminals, the following grammar excerpt shows a simple terminal with all its syntax-directed routines defined:

```
c
 (sym-class          Csymbol_c
 ,sym-look-ahead-rtn CSymbol_c_look_ahead_rtn
 ,stack-pushed-rtn   Csymbol_c_pushed_rtn
 ,stack-popped-rtn   Csymbol_c_popped_rtn);
```

The terminal symbol is c and to its right is the packaging that relates the terminal to its C++ class name and syntax-directed activity classes. The C++ class name assignment uses a keyword identifying the relationship expressed with its C++ class name partner. In this example, all the C++ class names start with the capital letter "C" and are to the right of their keywords: for the **c** terminal, its C++ class name is **Csymbol_c** supplied by the keyword *sym-class*. As part of each syntax-directed keyword, the stack activity indicates when the activity will execute — for example, *stack-pushed-rtn* keyword indicates this class will be executed when the **c** terminal is pushed onto the parse stack. The only exception is the *sym-look-ahead-rtn* keyword that provides an activity when it is used as the lookahead symbol in a reduce operation. These class names are compiler writer chosen and so should provide clarity and familiarity to the emitted code. From the example, the C++ name for the lookahead action class is **Csymbol_c_look_ahead_rtn**. This name is a little overkill but it brings home the point that what-you-want-is-what-you-get. For brevity, the C++ syntax-directed code has been omitted from this example. Appendix A "Terminals Section: Example" fully details these language contructs.

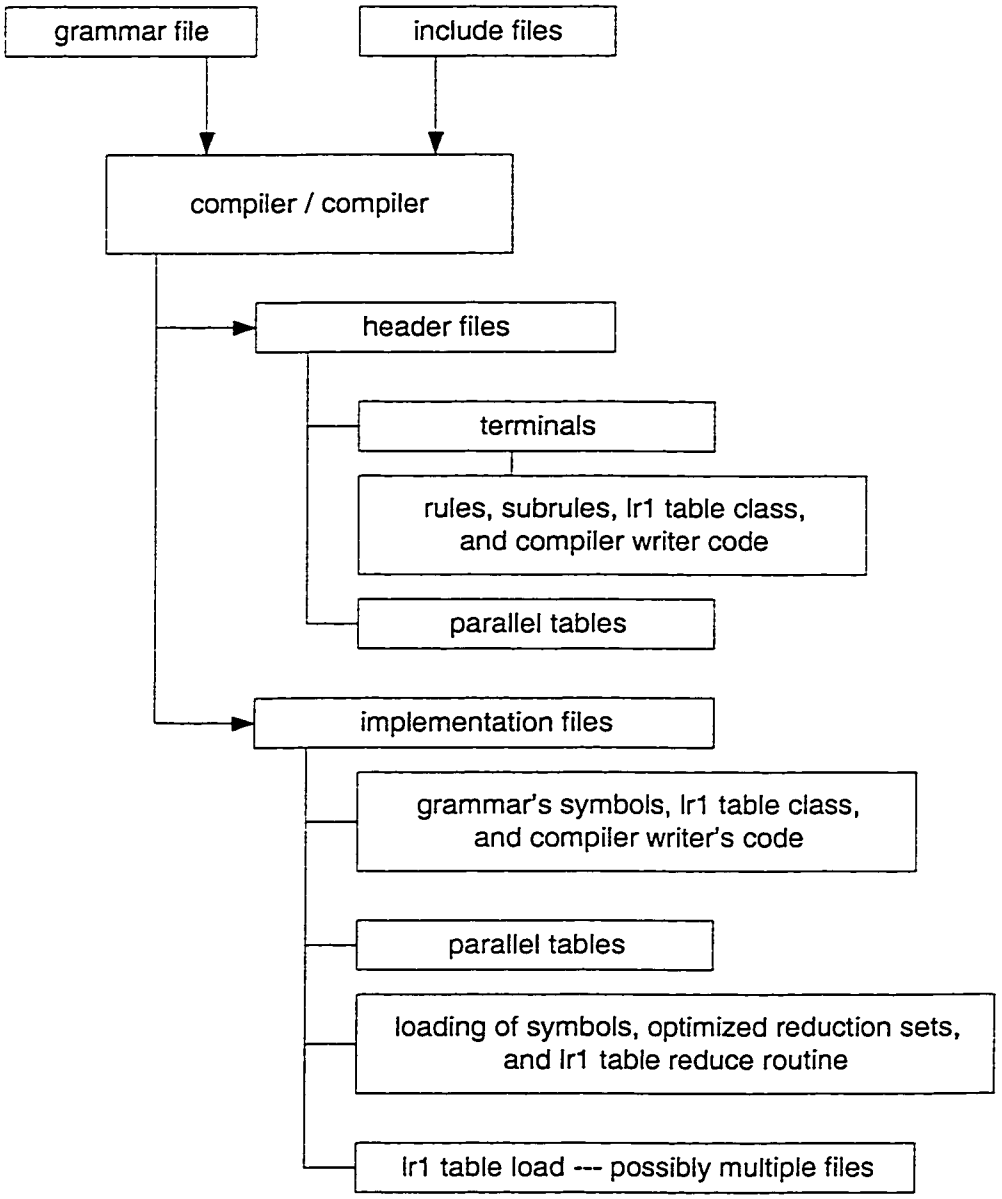

Figure 3.4: Emitted source files

When the grammar is compiled, various C++ source files are produced as depicted by Figure 3.4. The boxes with arrows leading into or out of the compiler/compiler box are files identified by their boxed comments. None arrowed lines emanating from the file boxes indicate the file's contents. The following subsection is an extract from the "Ctsuite" grammar: Appendix C "Lr1 Grammar Test Suite".

3.2.1 Specialization for the Terminals by Abstract Middlemen Classes

Middlemen classes define the boundaries of specialization for the compiling grammar and its vocabulary. They provide the segregation required in a polyglot of grammars each requiring their own name space to prevent naming conflicts of classes. The same comments apply to rules' names. For the "Ctsuite" grammar example, these middlemen are:

- *CAbs_Ctsuite_terminal_objects_workers* defines the specific workers for the grammar
- *CAbs_Ctsuite_terminal_sym* defines the base terminal which uses the specific workers

Their excerpted C++ definitions are:

```
class CAbs_Ctsuite_terminal_objects_workers;
class CAbs_Ctsuite_terminal_sym: public CAbs_lr1_sym{
  public:
    virtual      ~CAbs_Ctsuite_terminal_sym();
    virtual void // accept_workers rtn supporting IDOW pattern
    accept_workers(CAbs_Ctsuite_terminal_objects_workers* Workers)=0;
  protected:
    CAbs_Ctsuite_terminal_sym
        (CAbs_s_r_rtn*       Look_ahead_rtn
        ,CAbs_s_r_rtn*       Shift_rtn
        ,CAbs_s_r_rtn*       Reduce_rtn
        ,const char*         Id);
};
class CAbs_Ctsuite_terminal_objects_workers: public CAbs_lr1_sym_workers{
  public:
    CAbs_Ctsuite_terminal_objects_workers();
    ~CAbs_Ctsuite_terminal_objects_workers();
 virtual void worker_Csymbol_c(Csymbol_c* Object);
            . . . . . .
 virtual void worker_bs$$_5(bs$$_5* Object);
 virtual void worker_Csymbol_d(Csymbol_d* Object);
 virtual void worker_Csymbol_eolr(Csymbol_eolr* Object);
};
```

Again, the example shows the use of inheritance in constructing extended base classes: *CAbs_Ctsuite_terminal_sym* derived from *CAbs_lr1_sym* the abstract symbol class and *CAbs_Ctsuite_terminal_objects_workers* derived from *CAbs_lr1_sym_workers* the base class supporting the IDOW pattern. The middlemen manufactured base class names are built from individual components. For the terminals base class name, its middleman is *CAbs_Ctsuite_terminal_sym*. The compound name is prefixed by "CAbs_" indicating an abstract class, the grammar's class name which in this case is "Ctsuite", and a suffix "_terminal_sym". For the terminal object workers, its uses the same formula with its suffix being "_terminal_objects_workers". What is important is *CAbs_Ctsuite_terminal_sym*'s *accept_workers* routine passed parameter *CAbs_Ctsuite_terminal_objects_workers*. These are the specialized terminal workers for this specific grammar. Use of inheritance from an abstract base, allows the routine to receive an open-ended number of real worker objects derived from this base class. In the example above, the other workers defined are part of "Ctsuite" terminals extracted from the lexical grammar: Appendix C "Lr1 Grammar Test Suite".

3.2.2 Concrete Terminal Classes

Once the abstract classes are defined, the concrete terminal definitions follow deriving from their middlemen. All the concrete class names are controlled by the compiler writer. These are his own reference points to the terminals in his code. It is his vocabulary defining the objects. Due to forward referencing in the header file, the action classes are defined first before the terminal class definition. The C++ code snippet defines the terminal **c** given above:

```
class Csymbol_c_look_ahead_rtn: public CAbs_s_r_rtn{
  public:
    Csymbol_c_look_ahead_rtn(const char* Id = "Csymbol_c_look_ahead_rtn");
    ~Csymbol_c_look_ahead_rtn();
    void op(CAbs_lr1_sym* Terminal,CAbs_parser* Parser_env);
};

class Csymbol_c_pushed_rtn: public CAbs_s_r_rtn{
  public:
    Csymbol_c_pushed_rtn(const char* Id = "Csymbol_c_pushed_rtn");
    ~Csymbol_c_pushed_rtn();
    void op(CAbs_lr1_sym* Terminal,CAbs_parser* Parser_env);
};

class Csymbol_c_popped_rtn: public CAbs_s_r_rtn{
  public:
    Csymbol_c_popped_rtn(const char* Id = "Csymbol_c_popped_rtn");
    ~Csymbol_c_popped_rtn();
    void op(CAbs_lr1_sym* Terminal,CAbs_parser* Parser_env);
};

class Csymbol_c: public CAbs_Ctsuite_terminal_sym{
  public:
    ~Csymbol_c();
    void accept_workers(CAbs_Ctsuite_terminal_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);
    Csymbol_c(const char*   Id = "c");
    static class Csymbol_c_look_ahead_rtn look_ahead_rtn;
    static class Csymbol_c_pushed_rtn shift_rtn;
    static class Csymbol_c_popped_rtn reduce_rtn;
};
```

Even though the terminal c is simple and contains no compiler writer's injected code, all terminals are emitted with the same mold — *accept_workers* to support the IDOW pattern and *op* as an execution point, containment classes supporting its services, and having the capacity to absorb the compiler writer's own functionality associated with this terminal symbol. As described later in the rule-subrule subsection, stack references to this terminal are strongly typed to the **Csymbol_c** class.

This is quite an achievement using C++'s inheritance capability to define the grammar's symbol:

- a meaningful name for the inheriting class
- encapsulation of data and functionality by the class definition
- automatic support of services carried inside the class

Is this a forced marriage for the grammar's advantage? Yes it is, and what an offspring! OO certainly enriches the grammar's terminal.

3.3 Relationships: Rules and Subrules Conventions

Continuing with OO's class definitions of a grammar, rules will be represented by classes but what about their subrules? Technically, a subrule is a string of symbols but can it be looked at as an encapsulating entity? Yes with its contents being the string of symbols represented by classes which are strongly typed. Subrules have personalities; even though they are subordinates to the rule, they also exhibit syntax-directed code needs. In measuring the interplay between the rule and its subrules, I felt subrules should be class defined, their relationship to its rule should be controlled somehow, awareness was important amongst the rules and their subrules, and their generated class names should exhibit their ancestry. From these observations, four control mechanisms evolved:

1. a naming convention to associate the subrules classes with their production rule
2. enumerated types to uniquely identify each subrule and rule within a grammar. The rule's birth, when the subrule collapses into its rule, should be an event that is discrete with syntax directed code capabilities that distinguish its reducing subrule.

3. strongly typed accessing of the run stack's components. All items on the run stack are class objects and they are accessed by their real class name and not by an underlying class abstraction.

4. C++ friend usage allowing cross accessing of class parts between subrules, rules, and action service routines

These four requirements are individually explored in the following subsections.

3.3.1 Class Naming Convention for Rules and Subrules

The rule's symbolic key is also its class name. To easily distinguish a rule from a terminal, the rule name must begin with a "$": this is a liberty taken by the author. Like any hierarchy, the subrule names should clue the reader into its relationship with its parent: and so, using the rule's class name, a "Rhs#_" prefix is affixed creating the sub-rule's class name. In the example below, "Rhs1_$E" is a subrule of $E. The "#" in the prefix indicates the subrule's position, relative to one, in the defining production. This naming scheme is as bad as George Foreman's boys named George Junior 1 and a two and a .... but it works. The following example, broken into two pieces, shows the defining grammar's rule **$E** followed by the emitted classes for the subrules and its production.

```
      // grammar's rule
$E    // rule's symbolic key and class name
(sym-class        Csymbol_$E // action objects class container
,lhs
,stack-pushed-rtn Csymbol_$E_pushed_rtn
,stack-popped-rtn Csymbol_$E_popped_rtn)
{
  -> $A a    // subrule 1
  -> $B b    // subrule 2
  -> u $A b  // subrule 3
  -> u $B a  // subrule 4
};
```

$E has four subrules and stack action routines defined. The action routines act exactly as described in the "action services" at page 26. The example is an extract from the "Clr1_br1_rul_fsm" grammar defined in "Test Suite Grammars" of Appendix C. The derived from base class for the rule and subrule will be described later — see "Abstract Middlemen Classes" at page 43.

```
      // emitted subrule classes for $E rule
class Rhs1_$E: public CAbs_Clr1_br1_rul_fsm_rule_sym{
  public:
    Rhs1_$E(int P1,int P2
           ,CAbs_parser*      Parser_env
           ,int Sub_rule_id = Clr1_br1_rul_fsm::rhs1_$E
           ,const char* Id = "Rhs1_$E");
    ~Rhs1_$E();
    void accept_workers(CAbs_Clr1_br1_rul_fsm_rule_objects_workers* W);
    void op(CAbs_parser* Parser_env);
    $A* get_p1();
    Csymbol_a* get_p2();
  private: friend $E;
  friend Csymbol_$E_pushed_rtn;friend Csymbol_$E_popped_rtn;
   // subrule's string of symbols: $A a
   $A* p1_;
  Csymbol_a* p2_;
  CAbs_parser*  parser_env_;
};
        .... subrules 2 and 3 omitted ....
class Rhs4_$E: public CAbs_Clr1_br1_rul_fsm_rule_sym{
  public:
    Rhs4_$E(int P1,int P2,int P3
           ,CAbs_parser*      Parser_env
           ,int Sub_rule_id = Clr1_br1_rul_fsm::rhs4_$E
           ,const char* Id = "Rhs4_$E");
    ~Rhs4_$E();
    void accept_workers(CAbs_Clr1_br1_rul_fsm_rule_objects_workers* W);
    void op(CAbs_parser* Parser_env);
    Csymbol_u* get_p1();              // access rtn: 1st object
    $B* get_p2();                     // access rtn: 2nd object
    Csymbol_a* get_p3();              // access rtn: 3rd object
  private: friend $E;
  friend Csymbol_$E_pushed_rtn;friend Csymbol_$E_popped_rtn;
  // subrule's string of symbols: u $B a
  Csymbol_u* p1_;
  $B*        p2_;
  Csymbol_a* p3_;
  CAbs_parser*  parser_env_;
```

```
};
      // $E Rule
class $E: public CAbs_Clr1_br1_rul_fsm_rule_sym{
  public:
    ~$E();
    void accept_workers(CAbs_Clr1_br1_rul_fsm_rule_objects_workers* W);
    void op(CAbs_parser* Parser_env);
    $E(Csymbol_$E* Rule_sym_class,Rhs1_$E* Sub_rule_1
      ,CAbs_parser* Parser_env,const char* Id = "$E");
    $E(Csymbol_$E* Rule_sym_class,Rhs2_$E* Sub_rule_2
      ,CAbs_parser* Parser_env,const char* Id = "$E");
    $E(Csymbol_$E* Rule_sym_class,Rhs3_$E* Sub_rule_3
      ,CAbs_parser* Parser_env,const char* Id = "$E");
    $E(Csymbol_$E* Rule_sym_class,Rhs4_$E* Sub_rule_4
      ,CAbs_parser* Parser_env,const char* Id = "$E");
  private:
  friend Csymbol_$E_pushed_rtn;friend Csymbol_$E_popped_rtn;
  CAbs_parser* parser_env_;
};
```

From this C++ code example, not only is the naming conventions demonstrated but so is OO's inheritance. It also pre-maturely shows that a subrule uses the same middleman base class as its rule. This will be explained later.

3.3.2 Enumerated Types for Rules and Subrules

For the rule to know what subrule is recognized, two techniques are used:

1. enumeration of the grammar's rules and their subrules in the grammar's LR(1) table class. The subrule's constructor has its enumerated identity via a default parameter value. This value can be queried by the *sub_rule_id* routine described later.

2. use of contructors in the rule's class to import the specific subrule object just recognized

The previous example shows multiple constructor definitions for **$E** with its recognized subrules. Each subrule has its enumerated identity defined in the *enum* constant

sub_rules. Each rule has its enumerated identity defined in the enum constant "rules". The rule's constructor keeps track of the subrule's identity in case the compiler writer needs to dynamically cast the abstract subrule back into its concrete self. The following abbreviated example shows the enumerated types defined in the grammar's fsm class — the LR(1) parse table:

```
class Clr1_br1_rul_fsm: public CAbs_fsm{
 public:
  enum sub_rules{start_of_sub_rule_list = 0
       ,rhs1_$S
       ,rhs1_$E,rhs2_$E,rhs3_$E,rhs4_$E
       ,rhs1_$A,rhs2_$A
       ,rhs1_$B,rhs2_$B};
  enum rules{start_of_rule_list = 0,rule_$S,rule_$E,rule_$A,rule_$B};
  enum states{reduce = 0,R = reduce,no_of_states = 24};
  Clr1_br1_rul_fsm
    (CMapStringToOb** Gbl_sym_tbl=NULL
      ,const char*    Debug="yes"
      ,const char*    Comments="test out lr1 compiler/compiler"
      ,const char*    Id="lr1_br1.rul"
      ,const char*    Version="1.0"
      ,const char*    Date="8-oct-96");
  ~Clr1_br1_rul_fsm();
  void op(CAbs_parser* Parser_env);
              . . . . . .
};
```

The enum type *sub_rules* enumerates all the subrules using the following naming convention:

"rhs#_<rule name>" — subrule with its position within its rule class name.

The enum type "rules" enumerates all the rules within the grammar. It uses the following naming convention:

"rule_" affixed with the rule's class name

Normally these enum types are not needed by the syntax code, but they are open to use. A typical use for the subrule enum is to turn its subrule's abstractness back

into a concrete subrule, or to act as a filter value: for example, specific work is only done in the rule's *op* routine — the compiler writer does not want to be specific per subrule's syntax-directed code. The enumerated rule type is used in the bottom-up table lookup. I have not seen any reason outside this use, but who can foretell the future?

3.3.3 Strongly Typed Accessing of Run Stack Objects

Using the two previous examples, the subrule's access routines to the stack objects are defined in its class: see above **Rhs4_$E** class with its C++ comments. Each rule's constructor receives its subrule class object. For **$E**'s fourth subrule, the class name is **Rhs4_$E**. All the stack objects are strongly typed and class defined for each terminal or rule. The stack's objects are specifically accessed within each context. Each subrule's symbol object contained in its string of symbols is referenced by its relative position taken from the left and having its own access routine with a call name format of "get_p#" where the "#" indicates the position relative to one. For an epsilon rule there are no right-hand-side objects(empty string of symbols). For example, the following **$E** rule's contructor definition

```
$E (Csymbol_$E* Rule_sym_class, Rhs4_$E* Sub_rule_4
,CAbs_parser* Parser_env, const char* Id = "$E");
```

accesses its subrule's third stack object "a" by

```
Csymbol_a* a = Sub_rule_4->get_p3();
```

Each subrule parameter is named "Sub_rule_#" with the "#" indicating its subrule position within the rule. This specific name, "Sub_rule_4" in the constructor's prototype, was deliberately chosen to catch mistakes in the adding or the subtracting

of subrules from the rule which changes its references to the stack objects. Syntax-directed code using these specific references are affected by the strong typing of the stack objects, and so it requires maintenance. Errors will occur when the C++ emitted code is compiled. To ease the maintenance burden of modifying a grammar, place new subrules at the end of the rule's definition; unfortunately, removal of subrules from a rule will cascade errors to the ensuing subrules. This is due to the subrule's relative position within the rule has changed which causes the problem.

3.3.4 C++ Access Facility Using Friends

As a convenience to the compiler writer, the C++ friend access facility is used in the subrule and rule manufactured classes. The friends of the subrule class are its rule and the rule's action services. For the rule, its friends are its action services. The assigning of friends flows from the creation sequence of their objects. The subrule is created first, then the rule who imports its subrule, followed by the rule's action services of the push and pop stack classes. These friends are declared in the above emitted code examples using C++'s keyword "friend".

3.4 Rules

Rules are symbols that build on the base symbol class defined previously in "*CAbs_lr1_sym* Abstract Base Class: Mother Nature to All Symbols" at page 21. They have additional behaviours supporting their own rule–subrule relationships. The following subsections describe these additional environments:

- *an abstract base class CAbs_rule_sym for a rule*

- stack service routines equivalent to terminals
- middlemen providing specialization for the IDOW pattern

3.4.1 *CAbs_rule_sym* Abstract Base Class to All Rules

All rules derive indirectly from the *CAbs_rule_sym* base class. It acts as a container for rules and subrules Its contents are for specific class resurrections and action service routines. The C++ class definition is:

```
class CAbs_rule_sym: public CAbs_lr1_sym{
 public:
  CAbs_lr1_sym*  rule_sym_class();
  CAbs_s_r_rtn*  shift_rtn();
  CAbs_s_r_rtn*  reduce_rtn();
  CAbs_rule_sym* sub_rule();
  int            sub_rule_id();
  int            rhs_no_of_parms();
  virtual       ~CAbs_rule_sym();
 protected:
  CAbs_rule_sym                      // rule
      (CAbs_lr1_sym*  Rule_sym_class
      ,CAbs_rule_sym* Sub_rule
      ,const char*    Id);
  CAbs_rule_sym                      // sub rule
      (int            Rhs_no_of_parms
      ,int            Sub_rule_id
      ,const char*    Id);
 private:
  CAbs_lr1_sym*   rule_sym_class_;
  CAbs_rule_sym*  sub_rule_;
  int             sub_rule_id_;
  int             rhs_no_of_parms_;
};
```

The two constructors support rule and subrule creations. They get run automatically by the emitted code. They are access protected because this class must be derived from. It cannot stand on its own. Each grammar produces its middleman class deriving from *CAbs_rule_sym*.

The "rule_sym_class_" variable holds the container object for the stack action service routines. The name comes from the keyword in the rule's definition. The name is a bit arcane; it should describe it as a container of stack services. At the time of language design, the author thought this wrapper class should be explicitly defined. Current use is considering the compiler/compiler to implicitly generate it class name. The other variables relate the subrule used. "sub_rule_id_" is the *sub_rule*'s enum value. From this value, using the dynamic cast operator in C++, you can recover its concrete self. Each of these variables has an access function with the same name. As a convenience, the shift and reduce service routines are accessed through their own access routines *shift_rtn* and *reduce_rtn* rather than through the "CAbs_rule_sym_" variable.

3.4.2 Rule and Its Emitted Classes

To discuss rules, the following grammar excerpt shows a rule having all its syntax-directed actions routines defined and having only one subrule:

```
$MM(sym-class R$MM
   ,lhs
   ,stack-pushed-rtn C$MM_pushed_rtn
   ,stack-popped-rtn C$MM_popped_rtn)
  {
     -> $S eog // subrule
  };
```

As one can see, the rule is **$MM** and to its right is the packaging that relates the rule to its C++ code and syntax-directed activity classes as similarly and previously described in the "Terminals" section. In the case of the exampled rule, the rule's symbolic key is "$MM" which is also the C++ class name in the emitted code. As

in the terminal definition, the C++ class name assignment uses a keyword identifying the relationship expressed with its C++ class name partner. In this example, all the C++ class names start with the capital letter "C" and "R" and are to the right of their keywords. The only exception not having a C++ class name assignment is the *lhs* keyword that just provides additional C++ code to be injected into the rule's defining class named $MM and its implementation code. These class names are compiler writer chosen; this acquaintance with one's object names makes the syntax-directed code legible (at least this is the intent). For brevity, the C++ syntax-directed code has been omitted from this example. Appendix A "Rules Section: Example" fully details the rule's language contructs.

When the grammar is compiled, various C++ source files are produced. Within the C++ definition header are appropriate abstract classes acting as middlemen. The following subsection is an extract from the "Clr1_sp3_rul_fsm" grammar whose name becomes part of the generated abstract class's name. For the rule, its middleman is *CAbs_Clr1_sp3_rul_fsm_rule_sym* prefixed by "CAbs_" indicating an abstract class with its suffix "_rule_sym" (boy I'm long winded or is it my Germanic thought process?). *CAbs_Clr1_sp3_rul_fsm_rule_objects_workers* passed parameter to *accept_workers* is important. These parameters are the specialized rule workers for this specific grammar supporting the IDOW pattern. Each grammar, in multi-pass parsing, has its own middlemen. So far, I have not had any experience using the rule's IDOW pattern.

3.4.3 Abstract Middlemen Classes For Specialization of Rules and Subrules

Usually there is a common terminal definition file with its IDOW middleman emitted from a previous compilation of the terminal grammar. Middlemen classes define the boundaries of specialization for the compiling grammar and its vocabulary as described in the terminal section. They are the "dry wall" material helping to prevent class naming conflicts amongst grammars. One can still have a naming conflict but this is controlled by the compiler writer. For the "Clr1_sp3_rul" grammar, these middlemen are:

- *CAbs_Clr1_sp3_rul_fsm_rule_objects_workers* defines the specific workers for the grammar
- *CAbs_Clr1_sp3_rul_fsm_rule_sym* defines the base rule which uses the specific workers

Their excerpted C++ definitions are:

```
class CAbs_Clr1_sp3_rul_fsm_rule_objects_workers;
class CAbs_Clr1_sp3_rul_fsm_rule_sym: public CAbs_rule_sym{
  public:
    virtual                ~CAbs_Clr1_sp3_rul_fsm_rule_sym();
    virtual void
    accept_workers(CAbs_Clr1_sp3_rul_fsm_rule_objects_workers* Workers)=0;
  protected:
    CAbs_Clr1_sp3_rul_fsm_rule_sym       // rule
        (CAbs_lr1_sym*       Rule_sym_class
        ,CAbs_rule_sym*      Sub_rule
        ,const char*         Id);
    CAbs_Clr1_sp3_rul_fsm_rule_sym       // subrule
        (int                 Rhs_no_of_parms
        ,int                 Sub_rule_id
        ,const char*         Id);
};

class CAbs_Clr1_sp3_fsm_rule_objects_workers: public CAbs_lr1_sym_workers{
  public:
    CAbs_Clr1_sp3_fsm_rule_objects_workers();
```

```
    ~CAbs_Clr1_sp3_fsm_rule_objects_workers();
  virtual void worker_$MM($MM* Object);
  virtual void worker_R$MM(R$MM* Object);
  virtual void worker_Rhs1_$MM(Rhs1_$MM* Object);
  virtual void worker_$S($S* Object);
  virtual void worker_R$S(R$S* Object);
  virtual void worker_Rhs1_$S(Rhs1_$S* Object);
  virtual void worker_Rhs2_$S(Rhs2_$S* Object);
  virtual void worker_$A($A* Object);
  virtual void worker_R$A(R$A* Object);
  virtual void worker_Rhs1_$A(Rhs1_$A* Object);
  virtual void worker_Rhs2_$A(Rhs2_$A* Object);
  virtual void worker_$B($B* Object);
  virtual void worker_R$B(R$B* Object);
  virtual void worker_Rhs1_$B(Rhs1_$B* Object);
  virtual void worker_Rhs2_$B(Rhs2_$B* Object);
};
```

Each worker is associated with a rule, subrule, or a stack action service. For example, "worker_Rhs2_$B" can be read as worker for subrule 2 of rule $B — the Rhs stands for right-hand-side meaning subrule.

These are base class middlemen that must be derived from if the IDOW pattern is to be used. They are a shell from which the compiler writer embellishes his local activities. Whether the compiler writer needs them or not, for the moment they are emitted.

3.4.4 Concrete Rule Classes for Rule, Subrule, and Stack Action Services

Once the abstract classes are defined, the concrete rule definitions follow deriving from their middlemen. As always, the concrete class names are controlled by the compiler writer. Due to forward referencing in the header file, the action classes are defined first, followed by the subrules before the rule class definition. The C++ code snippet

defines the rule **$MM** given above along with its subrule **Rh1_$MM**, **R$MM** its stack services wrapper class, and its stack service classes **C$MM_pushed_rtn** and **C$MM_popped_rtn**:

```
// stack service routines
class C$MM_pushed_rtn: public CAbs_s_r_rtn{
  public:
    C$MM_pushed_rtn(const char* Id = "C$MM_pushed_rtn");
    ~C$MM_pushed_rtn();
    void          op(CAbs_lr1_sym* Rule,CAbs_parser* Parser_env);};
class C$MM_popped_rtn: public CAbs_s_r_rtn{
  public:
    C$MM_popped_rtn(const char* Id = "C$MM_popped_rtn");
    ~C$MM_popped_rtn();
    void          op(CAbs_lr1_sym* Rule,CAbs_parser* Parser_env);};
class R$MM: public CAbs_lr1_sym{
  public:
    R$MM(const char*   Id = "$MM");
    ~R$MM();
    void accept_workers(CAbs_Clr1_sp3_fsm_rule_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);
    static class C$MM_pushed_rtn shift_rtn;
    static class C$MM_popped_rtn reduce_rtn;};
     // subrules for rule $MM
class Rhs1_$MM: public CAbs_Clr1_sp3_fsm_rule_sym{
  public:
    Rhs1_$MM
    (int P1, int P2
    ,CAbs_parser* Parser_env
    ,int Sub_rule_id = Clr1_sp3_fsm::rhs1_$MM
    ,const char* Id = "Rhs1_$MM");
    ~Rhs1_$MM();
    void accept_workers(CAbs_Clr1_sp3_fsm_rule_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);
    $S* get_p1();
    Csymbol_eog* get_p2();
  private:  friend $MM; friend C$MM_pushed_rtn; friend C$MM_popped_rtn;
  $S* p1_; Csymbol_eog* p2_; CAbs_parser* parser_env_;
};
     // rule $MM
class $MM: public CAbs_Clr1_sp3_fsm_rule_sym{
  public:
    ~$MM();
    void accept_workers(CAbs_Clr1_sp3_fsm_rule_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);
    $MM
      (R$MM* Rule_sym_class
```

```
        ,Rhs1_$MM* Sub_rule_1
        ,CAbs_parser* Parser_env
        ,const char* Id = "$MM");
    private:
    friend C$MM_pushed_rtn; friend C$MM_popped_rtn;
    CAbs_parser*  parser_env_;
};
```

From above, $MM and its subrules classes derive from the base class *CAbs_Clr1_sp3_fsm_rule_sym* defined earlier. They have the same defining template of support routines — *op* and *accept_workers* — and friendships as developed for terminals.

3.5 Symbols' Life Cycle as Class Objects

To put a better perspective on the creating, accessing, and dying of a grammar's symbols, Figure 3.5 with running comments shows their life's spectrum in two parts: life sequence and rule's domain. Part one of the figure details the life sequence of the parser's stacked objects — terminals, subrules, and rules: their existence is described with the accompanying parser's stack action. Because the parser deals with pointers of symbol objects, it is up to the compiler writer to control the delete sequence of the object. To help alleviate when the object should be deleted, a delete attribute not described but documented in the grammar's language appendix is built into the symbol's class definition. It is checked by the parser at the time of object removal from the stack as to whether it should be deleted. If the automatic delete attribute is not turned on, then it is up to the compiler writer to explicitly delete the symbol object.

```
1) Life sequence:
  a terminal object is created
    -- object's pointer pushed onto the stack
    .....
  reduce a rule:
    -- subrule object created
    -- rule object created with imported subrule object pointer
    -- stack reduced of subrule objects and
         adjusted to contain rule's pointer

2) Domain of a production:
  rule
    |
     --> contains pointer to subrule object
    |
     --> subrule
           subrule's objects are fetched from the stack
           and stored as pointers inside the subrule's object.
           From this point, the stack reference is ignored.
         |
          --> contains strongly typed access routines to
              its objects by relative position -- position 1,2...
              starting from the left.
```

Figure 3.5: Creation timings and interplay between the subrule and its rule

The second part of the figure — Domain of a production — describes when and how the rule-subrule interplay of objects work. It comments how the symbol pointers on the stack are matched with the subrule's string of symbols and how the rule then accesses them.

3.6 Summary

This chapter covered all of a grammar's symbols as implemented by OO's components: class packaging and inheritance using base classes. I feel this is a good union of OO to a grammar. The evolution of the abstract base classes through to the middlemen

classes and final applied to the grammar's symbols was explained. This structural framework is just the beginning in exploring how bottom-up parsing can be played with using class objects.

At the beginning of the chapter I asked four questions which will be assessed in light of this chapter's development. Here are the questions:

1. How does one intelligently certify a grammar — both in the reading of the source definition and at run time?
2. How does the compiler writer relate to the emitted code?
3. How does the emitted LR(1) table relate to the grammar definition?
4. How does the syntax-directed code use the defining grammar's objects: rules and terminals?

Firstly, the compiler writer names the grammar's symbols which become the class definitions of these symbols. So, C++ code inspection will find the same grammar names for the productions as defined in the grammar. Tokens or terminals have their symbolic names supplied by the compiler writer. These tokens can be print dumped using the IDOW pattern: the data dumped is the symbolic key of the terminal with no distortion between the grammar and its C++ translation.

Secondly, the *op* procedure built within each symbol and action class has a ready made tracing macro that reports its symbolic class name at runtime. Other facilities in the "fsm" grammar language construct allow the compiler writer to turn on or off this tracing facility. This is particularly useful in parallel parsing with multiple grammars. It is the uniformity of the symbol's name across the grammar and the C++ code equivalent that eases the compiler writer debugging efforts.

Regarding the access of these objects at run time, C++ enforces strong data typing which has been incorporated into subrule classes. Each symbol in the subrule's string

has its own access procedure built into the class which returns a pointer to that specific class definition. All of this comes for free. An example of a grammar's runtime trace is given in the appendix. Question 3, LR(1) table relationship to the grammar, has not been answered. This will be explored in the next chapter.

Chapter 4

Finite State Machine

This chapter reviews the ready-made class definitions used to build the finite state machine of the compiled grammar. It starts with the base class development through to the derived concrete classes. I assume the reader has an understanding of the FSM: see [6] for the appropriate definitions. To complete its development, a grammar example drawn from the appendix will be given in its emitted FSM form.

4.1 *CAbs_fsm* Abstract Base Class

The *CAbs_fsm* class is a holding class filled in by the grammar's "fsm" language construct, the generated LR(1) FSM, and various routines supporting the loading up of the symbol table for rules and terminals. Some of the one time functions are protected due to their local abilities. Access routines to the *CAbs_fsm*'s internal data elements are there along with optimized terminal follow sets used by the parser's accept and reduce operations. Also, global optimization for common terminal symbol table for multi-pass and parallel grammars is supported. Figure 4.1 shows the simple

class hierarchy defining the holding class for the finite state machine.

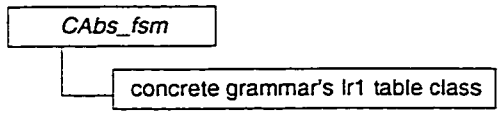

Figure 4.1: Fsm class hierarchy

This base class derives all emitted LR(1) tables. It is the compiler/compiler that manufactures the concrete FSM class inheriting from *CAbs_fsm*. Below is *CAbs_fsm* class definition:

```
class CAbs_fsm{
 public:
  virtual void       op(CAbs_parser* Parser_env)=0;
  virtual CAbs_fsm*  entry()=0;
  virtual CAbs_fsm*  exit()=0;
  CString*           id();
  CString*           version();
  CString*           date();
  CString*           debug();
  CString*           comments();
  void               gened_date(CString* Gened_date);
  CString*           gened_date();
  CPtrArray*         fsm_tbl();
  void               fsm_tbl(CPtrArray* Fsm_tbl);
  CMapStringToOb*    fsm_sym_tbl();
  void               fsm_sym_tbl(CMapStringToOb* Fsm_sym_tbl);
  CMapStringToOb*    fsm_parallel_bndry_sym_tbl();
  void               fsm_parallel_bndry_sym_tbl
                          (CMapStringToOb* Fsm_parallel_bndry_sym_tbl);
  int                add_to_fsm_tbl(int State_no,CAbs_sob* Fsm_tbl_object);
  CMapStringToOb*    state_s_fsm_entry_list(int State_no);
  int                state_s_fsm_entry_list
                          (int State_no
                          ,CMapStringToOb* State_s_fsm_state_entry_list);
  CAbs_sob*          get_state_entry_for_symbol(int State_no,CAbs_lr1_sym* Symbol);
  CAbs_lr1_sym*      get_sym(const char* Id);
  Cstate_s_boundry_set_index*
                     is_la_in_a_bndry_set
                         (Cstate_s_boundry_set_index_list* List_of_arrayed_bndry_set
                         ,const char* Id);
  void               add_sym(CAbs_lr1_sym* Id);
```

```
 void              add_parallel_bndry_sym
                       (int Index_to_arrayed_bndry_set
                       ,CAbs_lr1_sym* Id);
 virtual           ~CAbs_fsm();
 virtual           CAbs_rule_sym* reduce_rhs_of_rule
                       (int Tos,CAbs_parser* Parser_env,int Sub_rule_no)=0;
 virtual CAbs_fsm_oper* s_op()=0;// shift
 virtual CAbs_fsm_oper* r_op()=0;// reduce
 virtual CAbs_fsm_oper* a_op()=0;// accept
 virtual CAbs_fsm_oper* e_op()=0;// error
 void              fsm_gbl_sym_tbl_ind(int Ind);
 int               fsm_gbl_sym_tbl_ind();
protected:
 CAbs_fsm(CString*        Id
         ,CString*        Version
         ,CString*        Date
         ,CString*        Debug
         ,CString*        Comments
         ,CPtrArray*      Fsm_tbl
         ,CMapStringToOb* Fsm_sym_tbl
         ,CMapStringToOb* Fsm_oper_tbl);
 int C(int State_no,CAbs_sob* Fsm_state_entry_object);
 CAbs_lr1_sym*  G(const char* Id);
 CAbs_fsm_oper* F(const char* Id);
 int add_parallel_sob_to_parallel_sob_list
     (Cparallel_sobs_list* Parallel_sobs_list
     ,Cparallel_sob*       Parallel_sob);
 void init_bndry_sets_array(int No_of_indexes);
private:
 CString* id_;    CString* version_;  CString* date_;
 CString* debug_; CString* comments_; CString* gened_date_;
 CPtrArray*       fsm_tbl_;      // state no to CMapStringToOb
 CMapStringToOb*  fsm_sym_tbl_;  // symbol objects
 CMapStringToOb*  fsm_oper_tbl_; // operation objects
 CObArray         fsm_bndry_set_array_;
 int              fsm_gbl_sym_tbl_ind_;
};
```

As the components making up *CAbs_fsm* are large in number, their functionality classifies accordingly:

- source from the "fsm" language construct — id, date, version, etc.
- LR(1) fsm table holder, and its support routines

- run support for the parser from the "fsm-operation" language construct. This is commented-on in the next chapter and documented in the language appendix.

This exposé of *CAbs_fsm* should give the reader a sense of what OO gives to a FSM environment. It is not the author's intent to document each procedure contained in the class: hopefully their names provide their functionality to the reader. The following section now describes the component classes used to construct a LR(1) FSM table.

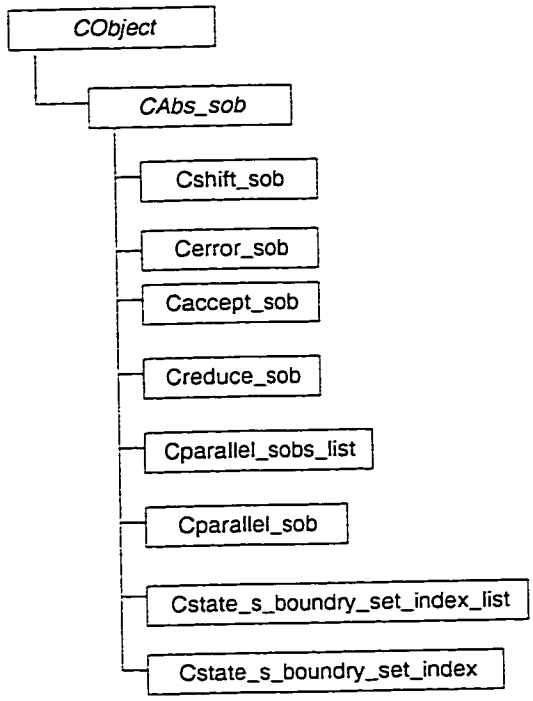

Figure 4.2: Lr1 table's class components hierarchy

4.2 *CAbs_sob* Abstract Base Class and Its Deriving Classes

CAbs_sob is the abstract base class that roots all derived LR(1) table's components making up the LR(1) state's context, symbol being acted on, and the bottom-up parse operation to be performed. The "sob" part of the name stands for "state object" and not what you normally take the acronym for. Again, the *accept_workers* participates in the "IDOW" pattern for the LR1 table objects. The following is *CAbs_sob* class definition:

```
class CAbs_sob: public CObject{
 public:
  ~CAbs_sob();
  int            self_state_no();
  CAbs_lr1_sym*  lr1_sym();
  CAbs_fsm_oper* fsm_operation();
  virtual void   accept_workers(CAbs_sob_workers* Worker)=0;
 protected:
  CAbs_sob(int            Self_state_no
          ,CAbs_lr1_sym*  Lr1_sym
          ,CAbs_fsm_oper* Fsm_operation);
 private:
 int self_state_no_;CAbs_lr1_sym* lr1_sym_;CAbs_fsm_oper* fsm_operation_;
};
```

Figure 4.2 shows all the concrete classes derived from this base class. These are the component classes that are mixed together to produce a LR(1) FSM table. Each class is now explained with its accompanying C++ class definition:

- **Cshift_sob** class. **Cshift_sob** signals to the parser that a shift operation is to be performed. Its contents identifies the state number it is in, the terminal symbol being shifted, an enhancing shift operation detailed in the "fsm-operation" language construct to be executed, and the go to state number for the shift operation.

```
class Cshift_sob: public CAbs_sob{
 public:
  Cshift_sob(int            Self_state_no
            ,CAbs_lr1_sym*  Lr1_sym_to_shift
            ,CAbs_fsm_oper* Fsm_operation
            ,int            Goto_state_no);
  ~Cshift_sob();
  int goto_state_no();
  void accept_workers(CAbs_sob_workers* Worker);
 private: int goto_state_no_;
};
```

- **Caccept_sob** class. **Caccept_sob** signals to the parser that a accept operation is to be preformed where by its contents indicate the subrule being collapsed into its start-rule using the specific follow set. It indicates that the language being recognized has been accepted by the grammar. The parser then signals success to the caller of the parser.

```
class Caccept_sob: public CAbs_sob{
 public:
  Caccept_sob(int            Self_state_no
             ,CAbs_lr1_sym*  Lr1_sym
             ,CAbs_fsm_oper* Fsm_operation
             ,int            Fsm_state_sym_enum);
  ~Caccept_sob();
  int        fsm_state_sym_enum();
  void       accept_workers(CAbs_sob_workers* Worker);
 private:int fsm_state_sym_enum_;
};
```

The variable *fsm_state_sym_enum_* is the enumerated value of subrule being accepted. Its value is defined by *sub_rules* enum constant in the concrete FSM class.

- **Creduce_sob** class. **Creduce_sob** signals to the parser that a reduce operation is to be preformed where by its contents indicate the subrule being collapsed into its rule using the specific follow set.

```
class Creduce_sob: public CAbs_sob{
 public:
  Creduce_sob(int            Self_state_no
             ,CAbs_lr1_sym*  LA_lr1_sym_to_reduce_with
             ,CAbs_fsm_oper* Fsm_operation
```

```
            ,int            Fsm_state_sym_enum);
  ~Creduce_sob();
  int       fsm_state_sym_enum();
  void      accept_workers(CAbs_sob_workers* Worker);
 private:int fsm_state_sym_enum_;
};
```

The variable *fsm_state_sym_enum_* is the enumerated subrule being reduced as defined in the enum variable *sub_rules* of the concrete FSM class.

- **Cerror_sob** class. It is built for the possibility of inserting, manually, error objects into the LR(1) table. I have not fully explored it but here is its definition:

```
class Cerror_sob: public CAbs_sob{
 public:
  Cerror_sob(int            Self_state_no
            ,CAbs_lr1_sym*  Lr1_sym
            ,CAbs_fsm_oper* Fsm_operation);
  ~Cerror_sob();
  void      accept_workers(CAbs_sob_workers* Worker);
};
```

- **Cstate_s_boundry_set_index_list** class. It is a container of optimized follow sets of **Cstate_s_boundry_set_index** objects used in the accepting and reducing operations of the subrule into its rule. It saves space in the LR(1) table by sharing common follow sets — terminal symbols — used across all the accept and reduce operations. Instead of one terminal per reduce or accept operation which can produce a lot of items to be loaded into the FSM table depending on the number of terminals in its follow set, only one entry per reducing or accepting subrule with its follow set pointer is entered into the FSM table.

```
class Cstate_s_boundry_set_index_list: public CAbs_sob{
 public:
  Cstate_s_boundry_set_index_list
            (int           Self_state_no
            ,CObArray*     State_s_boundry_set_index_list
            ,CAbs_lr1_sym* Lr1_sym ); // key |||b
  Cstate_s_boundry_set_index_list
           (int                          Self_state_no
           ,Cstate_s_boundry_set_index* One_only_boundry_set
           ,CAbs_lr1_sym*               Lr1_sym); // key |||b
  ~Cstate_s_boundry_set_index_list();
```

```
 void    accept_workers(CAbs_sob_workers* Worker);
 CObArray* state_s_boundry_set_index_list();
 void    add_boundry_set_index_to_list
            (Cstate_s_boundry_set_index* Boundry_set_index);
 int     no_of_boundry_set_indexes_in_list();
 Cstate_s_boundry_set_index* boundry_set_index(int List_ind);
 Cstate_s_boundry_set_index* one_only_boundry_set();
 void    one_only_boundry_set
            (Cstate_s_boundry_set_index* One_only_boundry_set);
private:
CObArray*          state_s_boundry_set_index_list_;
Cstate_s_boundry_set_index* one_only_boundry_set_;
};
```

- **Cstate_s_boundry_set_index** class. It identifies a specific follow set used in the accepting or reducing of a specific subrule into its rule. Its contents are specific follow set index keyed to **Cstate_s_boundry_set_index_list** and the follow set pointer.

```
class Cstate_s_boundry_set_index: public CAbs_sob{
 public:
  Cstate_s_boundry_set_index
       (int Boundry_set_index, CAbs_lr1_sym* Symbolic_boundry_set_id);
  ~Cstate_s_boundry_set_index();
  int  boundry_set_index();
  void accept_workers(CAbs_sob_workers* Worker);
  CAbs_lr1_sym* symbolic_boundry_set_id();
 private:
 int           boundry_set_index_;
 CAbs_lr1_sym* symbolic_boundry_set_id_;
};
```

These are the ingredients that make up the FSM table. The next section gives a real FSM class.

4.3 A Real FSM Class Definition

Here is the emitted C++ class for the Spector's grammar [13] used to show the run-time parse tracings described in Appendix C. From the example below, inheritance

is used and its constructor uses the defaulted values provided by the "fsm" language construct. It contains the enumerated constants *sub_rules*, *rules*, *states* that were previously described. These are the local sign posts of the compiled grammar.

```
class Clr1_sp3_fsm: public CAbs_fsm{
  public:
    enum sub_rules{start_of_sub_rule_list = 0
       ,rhs1_$MM, rhs1_$S,rhs2_$S, rhs1_$A,rhs2_$A,rhs1_$B,rhs2_$B};
    enum rules{start_of_rule_list = 0,rule_$MM,rule_$S,rule_$A,rule_$B};
    enum states{reduce = 0,R = reduce,no_of_states = 17};
    Clr1_sp3_fsm
      (CMapStringToOb**     Gbl_sym_tbl=NULL
      ,const char*          Debug="yes"
      ,const char*          Comments="test out lr1"
      ,const char*          Id="lr1_sp3.rul"
      ,const char*          Version="1.0"
      ,const char*          Date="8-oct-96");
    ~Clr1_sp3_fsm();
    void              op(CAbs_parser* Parser_env);
    CAbs_fsm*         entry();
    CAbs_fsm*         exit();
    CAbs_rule_sym*    reduce_rhs_of_rule
                          (int          T
                          ,CAbs_parser* Parser_env
                          ,int          Sub_rule_no);
    CAbs_fsm_oper*    s_op();//shift operation
    CAbs_fsm_oper*    r_op();//reduce operation
    CAbs_fsm_oper*    a_op();//accept operation
    CAbs_fsm_oper*    e_op();//error operation
  private:
    void load_symbols_into_fsm_symbol_table();
    void load_symbols_into_parallel_bndry_fsm_symbol_table();
    void load_states_into_fsm_state_table();
  CAbs_fsm_oper* s_;// shift
  CAbs_fsm_oper* r_;// reduce
  CAbs_fsm_oper* a_;// accept
  CAbs_fsm_oper* e_;// error
};
```

Private and public support functions are generated by the code emitter of the compiler/compiler:

- *load_symbols_into_fsm_symbol_table*

- *load_symbols_into_parallel_bndry_fsm_symbol_table*
- *load_states_into_fsm_state_table*
- *reduce_rhs_of_rule*

The names of the above functions indicate their intent: for example, the *reduce_rhs_of_rule* procedure handles the reducing of a subrule into its left-hand-side rule. It creates the subrule object, executes its service routines, creates its corresponding rule, and returns a pointer of the newly created rule object back to the parser.

Two of the local functions contain optimizations which are global terminal symbol table support shared across multiple grammars, and local shared follow sets for the accept/reduce operations. The *op* function is as described in other contexts: it provides a code injection point that is executed after the constructor which is supplied by the "op" language directive in "fsm" language contruct. The *entry* and *exit* functions are presently vestiges of the future or past thoughts: they currently are useless.

4.4 A Real FSM LR(1) Table Implementation

As the thesis deals in class definitions, the implementation code below is given to spice up the definition process. It is the C++ LR(1) table implementation for the Spector's grammar [13] given in appendix with its depicted fsm. The routines *C* and *G* — load the LR(1) component objects into the table, and find the object in the symbol table — have shortened names because the first draft of the emitted source files were too large for the C++ compiler used. The example demonstrates globally defined boundary sets used in the reducing of subrules and in the accepting of the

grammar:

```
#include "stdafx.h"
#include "clr1_sp3_rul.h"
void Clr1_sp3_fsm::
load_states_into_fsm_state_table(){
 Cstate_s_boundry_set_index_list* bsobl;
 Cstate_s_boundry_set_index* bsob;
 CObArray* bsob_array;
 CAbs_lr1_sym* symbolic_boundry_set;
 Sym_parallel_bndry_operator* bs_op =
    (Sym_parallel_bndry_operator*)G("|||b");
// s1 closure
C(1,new Cshift_sob(1,(R$MM*)G("$MM"),s_,1));
C(1,new Cshift_sob(1,(R$S*)G("$S"),s_,11));
C(1,new Cshift_sob(1,(R$A*)G("$A"),s_,7));
C(1,new Cshift_sob(1,(R$B*)G("$B"),s_,9));
C(1,new Cshift_sob(1,(Csymbol_a*)G("a"),s_,2));
// s2 transitive
C(2,new Cshift_sob(2,(R$A*)G("$A"),s_,3));
C(2,new Creduce_sob(2,(bs$$_1*)G("|||1"),r_,5));
C(2,new Cshift_sob(2,(R$B*)G("$B"),s_,5));
C(2,new Creduce_sob(2,(bs$$_2*)G("|||2"),r_,7));
// s2 closure
C(2,new Cshift_sob(2,(Csymbol_a*)G("a"),s_,13));
 bsob_array = new CObArray;
 bsobl = new Cstate_s_boundry_set_index_list(2,bsob_array,bs_op);
 C(2,bsobl);
 symbolic_boundry_set = (bs$$_2*)G("|||2");
 bsob = new Cstate_s_boundry_set_index(2,symbolic_boundry_set);
 bsob_array->Add(bsob);
 symbolic_boundry_set = (bs$$_1*)G("|||1");
 bsob = new Cstate_s_boundry_set_index(1,symbolic_boundry_set);
 bsob_array->Add(bsob);
// s3 transitive
C(3,new Cshift_sob(3,(Csymbol_c*)G("c"),s_,4));
// s4 transitive
C(4,new Creduce_sob(4,(bs$$_1*)G("|||1"),r_,4));
 symbolic_boundry_set = (bs$$_1*)G("|||1");
 bsob = new Cstate_s_boundry_set_index(1,symbolic_boundry_set);
 bsobl = new Cstate_s_boundry_set_index_list(4,bsob,bs_op);
 C(4,bsobl);
// s5 transitive
C(5,new Cshift_sob(5,(Csymbol_b*)G("b"),s_,6));
// s6 transitive
C(6,new Creduce_sob(6,(bs$$_2*)G("|||2"),r_,6));
 symbolic_boundry_set = (bs$$_2*)G("|||2");
 bsob = new Cstate_s_boundry_set_index(2,symbolic_boundry_set);
 bsobl = new Cstate_s_boundry_set_index_list(6,bsob,bs_op);
```

```
 C(6,bsobl);
// s7 transitive
C(7,new Cshift_sob(7,(Csymbol_b*)G("b"),s_,8));
// s8 transitive
C(8,new Creduce_sob(8,(bs$$_3*)G("|||3"),r_,2));
 symbolic_boundry_set = (bs$$_3*)G("|||3");
 bsob = new Cstate_s_boundry_set_index(3,symbolic_boundry_set);
 bsobl = new Cstate_s_boundry_set_index_list(8,bsob,bs_op);
 C(8,bsobl);
// s9 transitive
C(9,new Cshift_sob(9,(Csymbol_c*)G("c"),s_,10));
// s10 transitive
C(10,new Creduce_sob(10,(bs$$_3*)G("|||3"),r_,3));
 symbolic_boundry_set = (bs$$_3*)G("|||3");
 bsob = new Cstate_s_boundry_set_index(3,symbolic_boundry_set);
 bsobl = new Cstate_s_boundry_set_index_list(10,bsob,bs_op);
 C(10,bsobl);
// s11 transitive
C(11,new Cshift_sob(11,(Csymbol_eog*)G("eog"),s_,12));
// s12 transitive
C(12,new Caccept_sob(12,(bs$$_0*)G("|||0"),a_,1));
 symbolic_boundry_set = (bs$$_0*)G("|||0");
 bsob = new Cstate_s_boundry_set_index(0,symbolic_boundry_set);
 bsobl = new Cstate_s_boundry_set_index_list(12,bsob,bs_op);
 C(12,bsobl);
// s13 transitive
C(13,new Cshift_sob(13,(R$A*)G("$A"),s_,14));
C(13,new Creduce_sob(13,(bs$$_2*)G("|||2"),r_,5));
C(13,new Cshift_sob(13,(R$B*)G("$B"),s_,16));
C(13,new Creduce_sob(13,(bs$$_1*)G("|||1"),r_,7));
// s13 closure
C(13,new Cshift_sob(13,(Csymbol_a*)G("a"),s_,13));
 bsob_array = new CObArray;
 bsobl = new Cstate_s_boundry_set_index_list(13,bsob_array,bs_op);
 C(13,bsobl);
 symbolic_boundry_set = (bs$$_1*)G("|||1");
 bsob = new Cstate_s_boundry_set_index(1,symbolic_boundry_set);
 bsob_array->Add(bsob);
 symbolic_boundry_set = (bs$$_2*)G("|||2");
 bsob = new Cstate_s_boundry_set_index(2,symbolic_boundry_set);
 bsob_array->Add(bsob);
// s14 transitive
C(14,new Cshift_sob(14,(Csymbol_c*)G("c"),s_,15));
// s15 transitive
C(15,new Creduce_sob(15,(bs$$_2*)G("|||2"),r_,4));
 symbolic_boundry_set = (bs$$_2*)G("|||2");
 bsob = new Cstate_s_boundry_set_index(2,symbolic_boundry_set);
 bsobl = new Cstate_s_boundry_set_index_list(15,bsob,bs_op);
```

```
 C(15,bsobl);
// s16 transitive
C(16,new Cshift_sob(16,(Csymbol_b*)G("b"),s_,17));
// s17 transitive
C(17,new Creduce_sob(17,(bs$$_1*)G("|||1"),r_,6));
 symbolic_boundry_set = (bs$$_1*)G("|||1");
 bsob = new Cstate_s_boundry_set_index(1,symbolic_boundry_set);
 bsobl = new Cstate_s_boundry_set_index_list(17,bsob,bs_op);
 C(17,bsobl);
};
```

Except for the closure-only state 1, each state can be composed of a transitive part and possibly a closure part. In the code above, the C++ comment identifies the state and its section type being loaded into the LR(1) FSM table. The C++'s cast operator is used in the parser's operation objects Creduce_sob, Cshift_sob, and Caccept_sob which act as a visual aid to the compiler/compiler writer as to whether the emitting code was using the right symbol. The accepting state is 12 seen using the Caccept_sob object.

Chapter 5

Pushdown Automata

This chapter rounds out the ready-made classes defining the pushdown automata, or PDA. The base class is developed leading into a real parser class deriving from it. I assume the reader has an understanding of the PDA: see [6] for the appropriate definitions. To complete the chapter, implemented C++ code of the parser's operations are given so the reader sees "how and when" the stack service routines get executed. Following this, the "Winding down" section provides some experimental reflections as drawn by my experiences with the parser.

5.1 *CAbs_parser* Abstract Parser Class

The abstract parser class *CAbs_parser* is multifold in function:

- it contains the LR(1) table
- it supplies an enum constant *parse_result* for parsing results used by the generic parser. The parsing result is one of the following in the list: error, accepted, shifted, reduced, paralleled.

- it forces the real parse class to supply its bottom-up operations reduce, accept, shift, and error. This was a design decision allowing parser variations to be experimented with.
- it maintains the parse stack of parsing
- it manages the input and output token containers. This supports multi-pass parsing where one pass's token output is the next pass's token input.
- it supports a line marker object for future use in error processing. The idea is to use invisible tokens as event markers. Such thoughts might lead to tracing support tokens for debugging environments.
- it contains the genesis of parallel parsing for future research

The current version of the compiler/compiler emits code using Microsoft's MFC container classes. C++'s Templates were considered but not used due to their instability at the time of implementation. Once templates stabilize, they will be incorporated into the compiler/compiler for future portability. Following is the C++ definition. The method names within the class definition are self-explanatory.

```
class CAbs_parser{
 public:
  enum parse_result{error,accepted,shifted,reduced,paralleled};
  virtual            ~CAbs_parser();
  virtual            parse_result parse()=0;
  virtual            parse_result parallel_parse()=0;
  virtual            parse_result shift()=0;
  virtual            parse_result reduce()=0;
  virtual            parse_result accept()=0;
  virtual            parse_result start_parallel_parsing()=0;
  virtual            parse_result parallel_shift(CAbs_lr1_sym* Token)=0;
  virtual            void op();

  CAbs_fsm*          fsm_parse_tbl();
  void               fsm_parse_tbl(CAbs_fsm* Fsm_parse_tbl);
  CObArray*          parse_stack();
  Token_container*   token_supplier();
  void               token_supplier(Token_container* Token_supplier);
  Token_container*   token_producer();
  void               token_producer(Token_container* Token_producer);
  CAbs_lr1_sym*      get_token();
```

```
 WORD              get_token_pos();
 CAbs_lr1_sym*     get_spec_stack_token(int Pos);
 CAbs_parse_record* get_current_stack_record();
 CAbs_parse_record* get_stack_record(int Pos);
 CAbs_parse_record* current_parse_record();
 void              current_parse_record
                       (CAbs_parse_record* Current_parse_record);
 int               remove_from_stack(int No_to_remove);
 int               remove_current_record_from_stack();
 void              add_to_stack(CAbs_parse_record* Parse_record);
 CAbs_sob*         find_sob_for_token(CAbs_lr1_sym* Token_sym);
 CAbs_sob*         current_sob();
 void              current_sob(CAbs_sob* Current_sob);
 CAbs_lr1_sym*     current_token();
 void              current_token(CAbs_lr1_sym* Current_token);
 WORD              current_token_pos();
 void              current_token_pos(WORD Pos);
 int               current_stack_pos();
 void              current_stack_pos(int Pos);
 void              set_up_parse_stack(CAbs_parse_record* Parse_record);
 void              clear_parse_stack();
 LM*               line_marker();
 void              line_marker(LM* Line_marker_token);

 // parallel parsing
 Cparallel_sobs_list* find_parallel_sob_from_token
                          (Cparallel_parsing_sym* Token_sym);
 CAbs_lr1_sym*     get_parallel_token();
 CAbs_lr1_sym*     get_spec_parallel_token(int Pos);
 Cparallel_sobs_list* parallel_sobl();
 void parallel_sobl(Cparallel_sobs_list* Parallel_sobs_list);
protected:
 CAbs_parser
   (CAbs_fsm*          Fsm_parse_tbl
   ,Token_container*   Token_supplier
   ,Token_container*   Token_producer);
 CAbs_parser
   (CAbs_fsm*          Fsm_parse_tbl
   ,DWORD              Control_monitor_thread_id
   ,DWORD              Get_token_thread_id
   ,DWORD              Thread_id
   ,Ccontrol_monitor_to_parallel_parser_thread_start_parse*
                       CM_pf_start_parse_parm);
private:
CAbs_fsm*              fsm_parse_tbl_;
Token_container*       token_supplier_;// use bridge pattern
Token_container*       token_producer_;
CObArray               parse_stack_;
```

```
CAbs_parse_record*      current_parse_record_;
CAbs_lr1_sym*           current_token_;
WORD                    current_token_pos_;
CAbs_sob*               current_sob_;
int                     current_stack_pos_;
LM*                     line_marker_;
Cparallel_sobs_list*    parallel_sobl_;// is ||| going on?
};
```

From the definition, protectionism on the class constructors forces use of *CAbs_parser* by inheritance. The second contructor is under construction supporting parallel parsing. Currently, it is not supported.

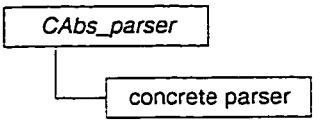

Figure 5.1: Parser class hierarchy

5.2 A Real Parser Class

A real parser, with its parse operations, is derived from the abstract *CAbs_parser* defined above. Figure 5.1 shows the simple class hierarchy of a real parser deriving from its base class *CAbs_parser*. Each basic bottom-up operation is defined and specifically implemented for shift, reduce, and accept. All the parser's operations work in concert with the supplemental code classes explained later. In the code shown, there are references to parallel parsing which is currently being researched: I apologize to the reader for their inclusion but at the time of writing this thesis, this

was the current state of the extracted code. These references can be skipped over; these parallel routines normally ask a question "start_parallel_parsing()?" and if the answer is true then it calls a parallel parsing environment. The **Parser** class defined below is a generic parser with built-in tracing facilities. It is a model from which experiments in parse variations can be played with.

```
class Parser: public CAbs_parser{public:
  virtual ~Parser();
  CAbs_parser::parse_result parse();
  CAbs_parser::parse_result parallel_parse();
  CAbs_parser::parse_result shift();
  CAbs_parser::parse_result reduce();
  CAbs_parser::parse_result accept();
  CAbs_parser::parse_result start_parallel_parsing();
  CAbs_parser::parse_result parallel_shift(CAbs_lr1_sym* Token);
  Parser
    (CAbs_fsm*        Fsm_parse_tbl
    ,Token_container* Token_supplier
    ,Token_container* Token_producer
    ,WORD             Token_supplier_key_pos=0);
  Parser    // currently not supported --- under research
    (CAbs_fsm*        Fsm_parse_tbl
    ,DWORD            Control_monitor_thread_id
    ,DWORD            Get_token_thread_id
    ,DWORD            Thread_id
    ,Ccontrol_monitor_to_parallel_parser_thread_start_parse*
                      CM_pf_start_parse_parm);
};
```

Figure 5.2 describes each operation within its own context. Each parse operation is labeled and framed, within it the functional steps are individually described in a high level way: for example, trace stack configuration. Some steps are generic to the specific operations like "add to stack" and "remove from stack".

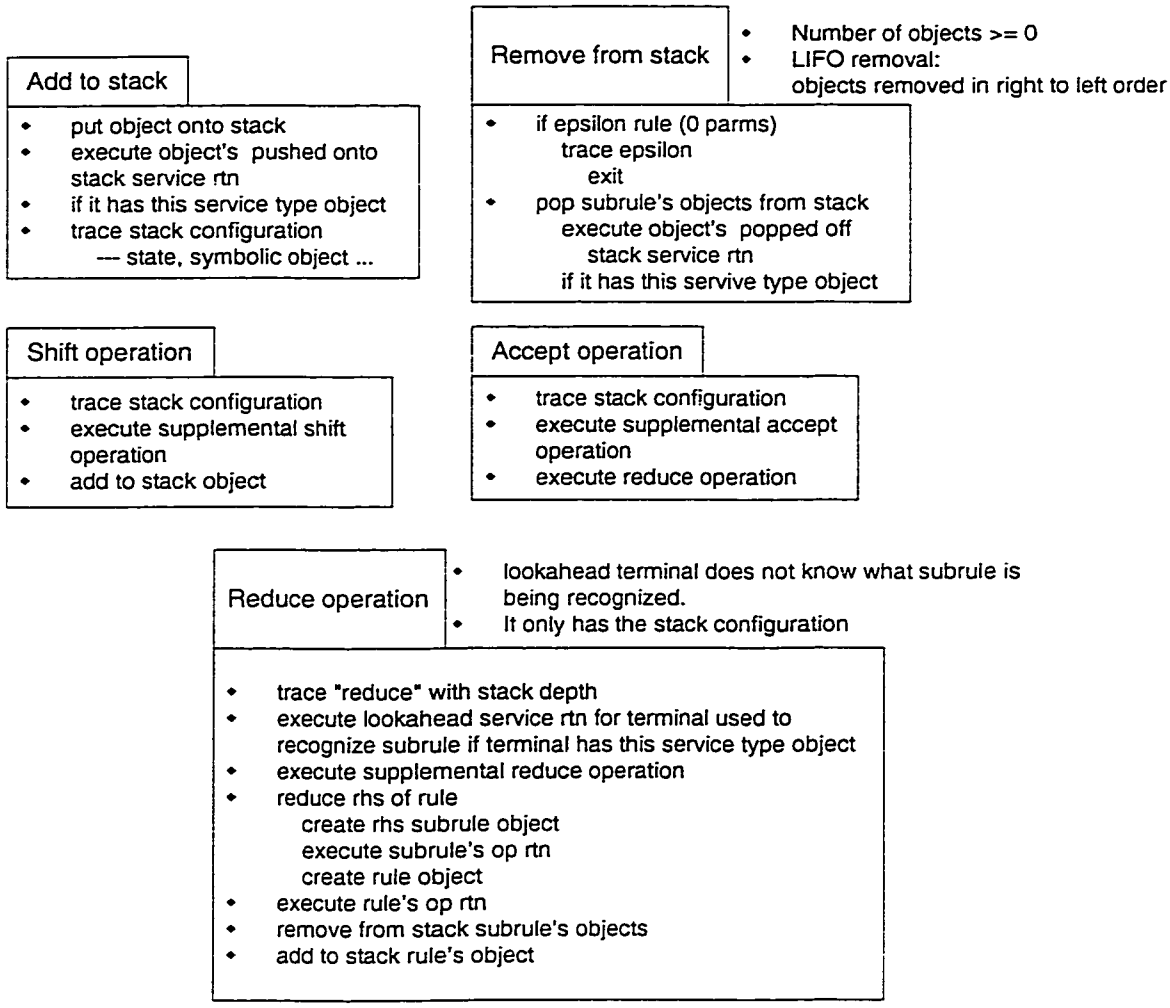

Figure 5.2: Parser operations

5.2.1 A Parser Implementation

The following code is the generic parser used throughout the examples:

```
CAbs_parser::parse_result Parser::parse(){
 op();            // 1st time FIRE OFF parser's op
 Cerror_sob error_sob(0,0,fsm_parse_tbl()->e_op());
 if (current_token() == NULL)
  return CAbs_parser::accepted;// empty language - no tokens
 //
 // 1) lookup token in current parse state entry table
 // 2) do one of the actions
 //   a) shift b) reduce c) accept d) user action e) abort
 //
```

```
  current_parse_record(get_current_stack_record());
  switch (do_parallel_parsing()){
  case 1:{start_parallel_parsing();break;};// ||| parsing
  case 0:{                        // straight parsing
   current_sob(find_sob_for_token(current_token()));
   if (current_sob() == NULL) goto parse_error;
   switch (current_sob()->fsm_operation()->op_type()){
    case CAbs_fsm_oper::reduce_op:{
     if (reduce() == CAbs_parser::accepted) goto end_of_parse;
     break;}
    case CAbs_fsm_oper::shift_op:{shift();break;}
    case CAbs_fsm_oper::accept_op:{accept();goto end_of_parse;}
    default:{        // USER DEFINED
     current_sob()->fsm_operation()->op();break;}
   }
   break;} // end of straight parsing
  } // end of parallel or straight parsing
 }
end_of_parse:
 return CAbs_parser::accepted;
parse_error:
 error_sob.fsm_operation()->op(this);
 return CAbs_parser::error;
}
```

There are three things to note in the above code:

1. there is reference to parallel parsing which is currently being researched — this is a snap shot of code currently being used in my studies. Within the above code, the reader only needs to look at the "straight parsing" logic.

2. user defined supplemental operations are supported but have not been tested.

3. labeled "parse_error:" in the code, the error processing capability is minimal: it executes the **Cerror_sob** object defined at page 56. It is declared locally in the body of the parser. At this point in time, I've have not concluded how to handle errors — be it ad hoc support or canonical driven. Currently the language research into error correction is not as vigorously researched as in the past since errors can be corrected interactively. But if one views the work in the throwing and catching of errors in the C++ language, I believe there is more research to be done.

For the structured-code readers, the use of a goto statement must raise the hair on the nape of your necks; I use it in a disciplined downward-only control flow.

5.2.2 Implementation of Parser Operations

The following parse operations are exposed so that the reader gets a feel of "when and where" the firing-off mechanisms are deployed. These mechanisms are sparsely commented-on in each parse operation. Their accompanying C++ comments should guide the reader.

- *shift* procedure. This is the parser's shift operation. Firstly, it executes a supplemental operation *fsm_operation()->op(this)*. This is explained later in the chapter. The *add_to_stack* procedure pushes the terminal symbol onto the stack. Within *add_to_stack*, the stack service routines are executed for the pushed symbol. Closing off the *shift* operation is the fetching of the next token.

```
CAbs_parser::parse_result Parser::shift(){
 current_sob()->fsm_operation()->op(this);//FIRE OFF fsm's action rtn
 current_parse_record()->symbol(current_token());//state's shift sym
 // shift terminal into new state
 add_to_stack(new CAbs_parse_record
                    (((Cshift_sob*)current_sob())->goto_state_no()
             ,NULL
             ,fsm_parse_tbl()->state_s_fsm_entry_list
                 (((Cshift_sob*)current_sob())->goto_state_no())));
 if (control_monitor_thread_id()) current_token(get_parallel_token());
 else current_token(get_token()); // get new token
 return CAbs_parser::shifted;
}

void CAbs_parser::add_to_stack(CAbs_parse_record* Parse_record){
  CAbs_parse_record* pr = get_current_stack_record();
  parse_stack()->Add(Parse_record);
  current_stack_pos_++;
  if (pr->symbol()->shift_rtn() != NULL)
   pr->symbol()->shift_rtn()
     ->op(pr->symbol(),this);// FIRE OFF pushed on stack rtn
}
```

Also included is the C++ code for *add_to_stack* procedure showing "when and how" the push-onto-stack services get executed for the symbol. Brazenly, the C++ comment "FIRE OFF" tags "how and when" the symbol's shift service routine executes. This type of comment is sprinkled throughout the other pieces of code having the same intent.

- *reduce* procedure. The C++ code with its C++ comments describes the reduce routine logic. The *reduce_rhs_of_rule* routine is the emitted code from the compiler/compiler that finds the appropriate subrule to collapse into its left-hand-side rule using the lookahead terminal within its follow set, the parse stack, and state table of the grammar. The "fire off" events are also C++ commented; these are the service routines described previously in the thesis.

```
CAbs_parser::parse_result Parser::reduce(){
 if (current_token()->look_ahead_rtn() != NULL)
    current_token()->look_ahead_rtn()
      ->op(current_token(),this);// FIRE OFF lookahead action rtn

 current_sob()->fsm_operation()
              ->op(this);// FIRE OFF fsm's action rtn
 CAbs_rule_sym* rule =   // create rhs + FIRE OFF op() + and rule
   fsm_parse_tbl()->
   reduce_rhs_of_rule
       (current_stack_pos(),this
       ,((Creduce_sob*)current_sob())->fsm_state_sym_enum());
 rule->op(this);              // FIRE OFF lhs's op rtn
 remove_from_stack(rule->rhs_no_of_parms());
 current_parse_record()->symbol(rule);//stack state's rule shift sym
 current_sob(fsm_parse_tbl()->
   get_state_entry_for_symbol(current_parse_record()->state_no(),rule));

 add_to_stack// shift rule into new state
    (new CAbs_parse_record(((Cshift_sob*)current_sob())->goto_state_no()
    ,NULL
    ,fsm_parse_tbl()->state_s_fsm_entry_list
        (((Cshift_sob*)current_sob())->goto_state_no())));
 return CAbs_parser::reduced;
}

int CAbs_parser::remove_current_record_from_stack(){
  CAbs_parse_record* Parse_record = get_current_stack_record();
  delete Parse_record;
  return TRUE;
};

int CAbs_parser::remove_from_stack(int No_to_remove){
  if (current_stack_pos_ == -1) return FALSE; // underflow
  if ((No_to_remove < 0) || (No_to_remove > current_stack_pos_ + 1))
    return FALSE;
  CAbs_parse_record* pr;
  if (No_to_remove == 0) return TRUE;// epsilon rule
  //
  //  rhs pop goes the weasel
```

```
  //
  for(;No_to_remove > 0;No_to_remove--){
    pr = get_stack_record(current_stack_pos_);
    delete pr;                          // state causing reduction
    parse_stack()->RemoveAt(current_stack_pos_,1);
    current_stack_pos_--;
    current_parse_record(get_current_stack_record());
    pr = get_stack_record(current_stack_pos_);// exposed symbol that
                                              // goto'ed into
                                              // popped state
    if (pr == NULL) break;              // no more stack
    // find out if auto delete should be done before pop rtn
    // cus pop rtn could delete symbol and you're left with dangling
    // non existent object to test against
    bool del = pr->symbol()->auto_delete();
    if (pr->symbol()->reduce_rtn() != NULL)
      pr->symbol()->reduce_rtn()
        ->op(pr->symbol(),this);        // FIRE OFF popped off stack rtn
    if (del) delete pr->symbol();

  }
  return TRUE;
}
```

Included is the C++ code for *remove_from_stack* procedure showing "when and how" the pop-from-stack services get executed for the symbol. It also shows how the automatic delete attribute is used to delete the popping off symbol objects.

- *accept* procedure. From the C++ comments in the below code, an *accept* operation is really a *reduce* operation with "accept" supplemental code returning a *CAbs_parser::accepted* result.

```
CAbs_parser::parse_result Parser::accept(){
 // let reduce FIRE OFF this operation
 // clear the stack as normal with lhs being pushed
 // onto the parse stack
 // the ~parse will pop off the $start rule
 //
 reduce();
 return CAbs_parser::accepted; // acceptance
};
```

5.3 Supplemental Code to Parsing Operations

This group of classes was designed to support two functions:

- to supply the bottom-up operation to be performed in the real parser. This operation is defined in *operation* an enumerated list within *CAbs_fsm_oper* the base class. This enumerated list can be experimented with to add new functionality to the real parser — parallel parsing support did just this. All the LR(1) table's objects have one of the following supplemental operation objects imported into it.
- to supply an augmenting functionality to the parser's base of operations. There is a one-to-one definition between the real parser's generic operations defined previously and this group of classes.

These classes get injected into the parsing environment using the "fsm-operation" language construct defined in Appendix A — The Grammar Language. At the present time, default user defined operations have not been experimented with. They are executed by default in the real parser — see implementation code of **Parser** with the C++ comment of "//USER DEFINED". The following classes are default support to the generic parser. It is the *op* routine of the derived class that supplies the supplement. Passed as a parameter is the abstract parsing environment which the *op* routine would dynamically cast back into a concrete object. The *CAbs_fsm_oper* base class identifies the parse operation to be performed which is supplied by the derived class. Its C++ class definition follows:

```
class CAbs_fsm_oper{
 public:
  enum operation{reduce_op=0,shift_op,accept_op,error_op,parallel_parse_op};
  CString*     id();
  int          op_type();
  virtual void op(CAbs_parser* Parser_environment);
  virtual      ~CAbs_fsm_oper();
 protected:
  CAbs_fsm_oper(const char* Id,int Op_type);
 private: CString* id_; int op_type_;
```

```
};
```

The generic parser uses the *operation* enum as a verb expressing the parse object's functionality. The following definitions are just model code showing how a compiler writer might write his own enhancing operations for injection into the generic parser's operations. At present, they don't do too much apart from firing off the parse operation, and when in debug mode the outputting of tracing information. As to being really useful, this will be future assessed.

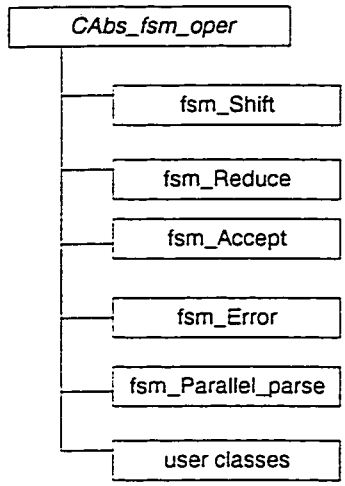

Figure 5.3: Class hierarchy of supplemental parse operations

5.3.1 Supplemental Code Classes

Figure 5.3 shows the supplemental class hierarchy using inheritance. The various supplemental operations with their C++ class definitions are now commented:

- the shift's supplemental operation just traces its operation type. Its implementation has the built-in stack tracings when the "_DEBUG" symbol is defined. This is the same symbol used in Microsoft's Visual C++ compiler for program tracings. The C++ class definition follows:

```
class fsm_Shift: public CAbs_fsm_oper{
 public:
  fsm_Shift((const char* Id, int Op_type = CAbs_fsm_oper::shift_op);
  ~fsm_Shift();
  void op(CAbs_parser* Parser_environment);
};
```

- the reduce's supplemental operation just traces its operation type. Its implementation has the built-in stack tracings when the "_DEBUG" symbol is defined. The C++ class definition follows:

```
class fsm_Reduce: public CAbs_fsm_oper{
 public:
  fsm_Reduce(const char* Id, int Op_type = CAbs_fsm_oper::reduce_op);
  ~fsm_Reduce();
  void op(CAbs_parser* Parser_environment);
};
```

- the accept's supplemental operation just traces its operation type. Its implementation has the built in stack tracings when the "_DEBUG" symbol is defined. The C++ class definition follows:

```
class fsm_Accept:public CAbs_fsm_oper{
 public:
  fsm_Accept(const char* Id, int Op_type = CAbs_fsm_oper::accept_op);
  ~fsm_Accept();
  void op(CAbs_parser* Parser_environment);
};
```

- the error's supplemental operation just traces its operation type. Its implementation has built-in stack tracings when the "_DEBUG" symbol is defined. It is crude in functionality — it just prints out the stack contents and the faulty token. It does not try to do any error recovery. The C++ class definition follows:

```
class fsm_Error: public CAbs_fsm_oper{
 public:
  fsm_Error(const char* Id, int Op_type = CAbs_fsm_oper::error_op);
  ~fsm_Error();
  void op(CAbs_parser* Parser_environment);
};
```

5.4 Winding Down

The following diagrams ear-mark the contexts described in the thesis while experimenting with the parser and the compiler/compiler's tables. I found these various run situations required documentating for context comprehension. These legends hopefully will give the reader the same values. Whilst some diagrams repeat material expressed before with variations, other diagrams provide some glimpses into their functional use with current implementation limitations. Though the implementation issues were not developed in the thesis like LR(1) table layout, accessing the LR(1) table with its objects, efficiency issues at run time, my use of C++ and whether its can be bettered will be looked at in the future. These diagrams break out into the following ideas:

- event driven mechanisms: from the finite state diagram of bottom-up operations to the meta-rules: pure events supporting the compiler writer's needs. How well are the events integrated into the bottom-up running framework? The junction points are currently mapped.
- multi-pass processing using meta-tokens. In the future, parallel parsing will build on the meta-token theme.
- information contexts available to the event processing mechanism as currently developed. With use, these contexts will be refined.

I feel mentioning their functional contexts gives the reader more food for thought into the development direction being taken with this compiler/compiler project.

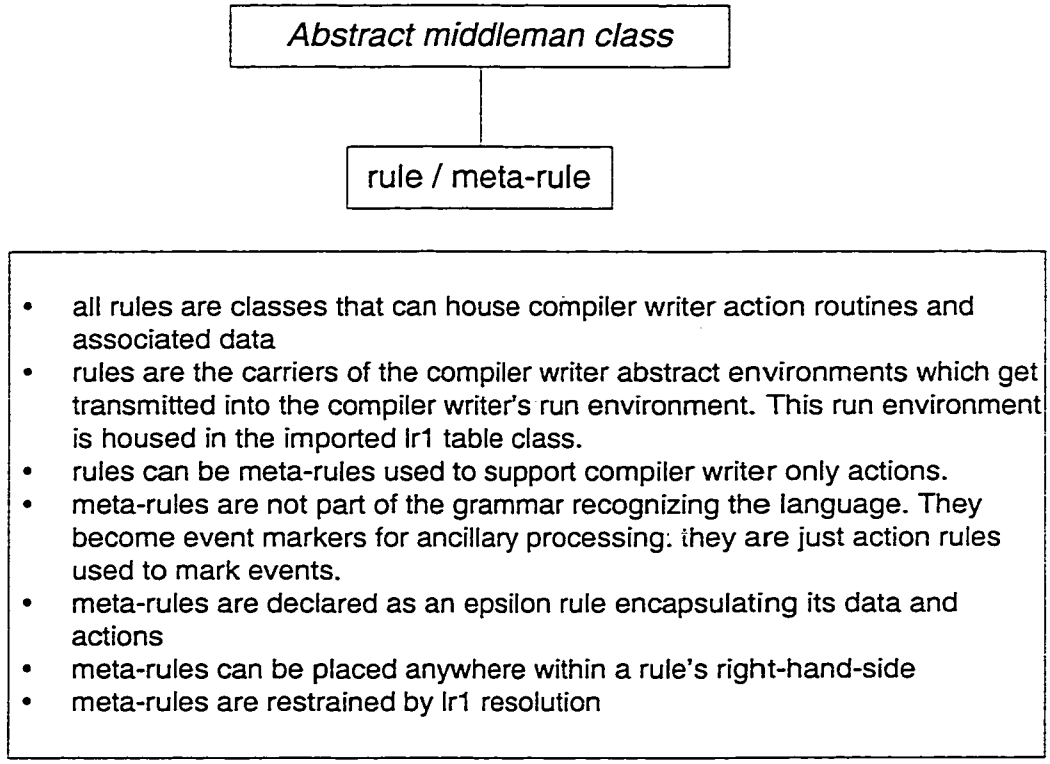

Figure 5.4: Grammar objects with meta-rules

Figure 5.4 contains comments about meta-rules. In general, meta-rules are used as action services. They are epsilon type rules used in specific places to activate a compiler writer's actions. At this point in the development of the compiler/compiler, these rules are governed by the LR(1) constraint.

Figure 5.5 comments on the finite state events fired off by the **Parser**. It identifies, within each parser's operational context, the various procedures executed that are associated with the grammar's symbol. This imported code per grammar symbol gives the flexibility to the compiler writer in fine-tuning his own semantic requirements. The arrows in the diagram direct the execution steps. To differentiate between generic actions by the parser, a larger font is used: for example, "add to stack". Shift, reduce, accept parse actions have labeled boxes with their contents commenting on

their finite state actions. Flowing out of these boxes are the directing arrows leading to other action boxes. Within some of the boxes are the various class procedures with comments that have been discussed throughout the thesis. The language directives built within the grammar inject the compiler writer's code into these areas.

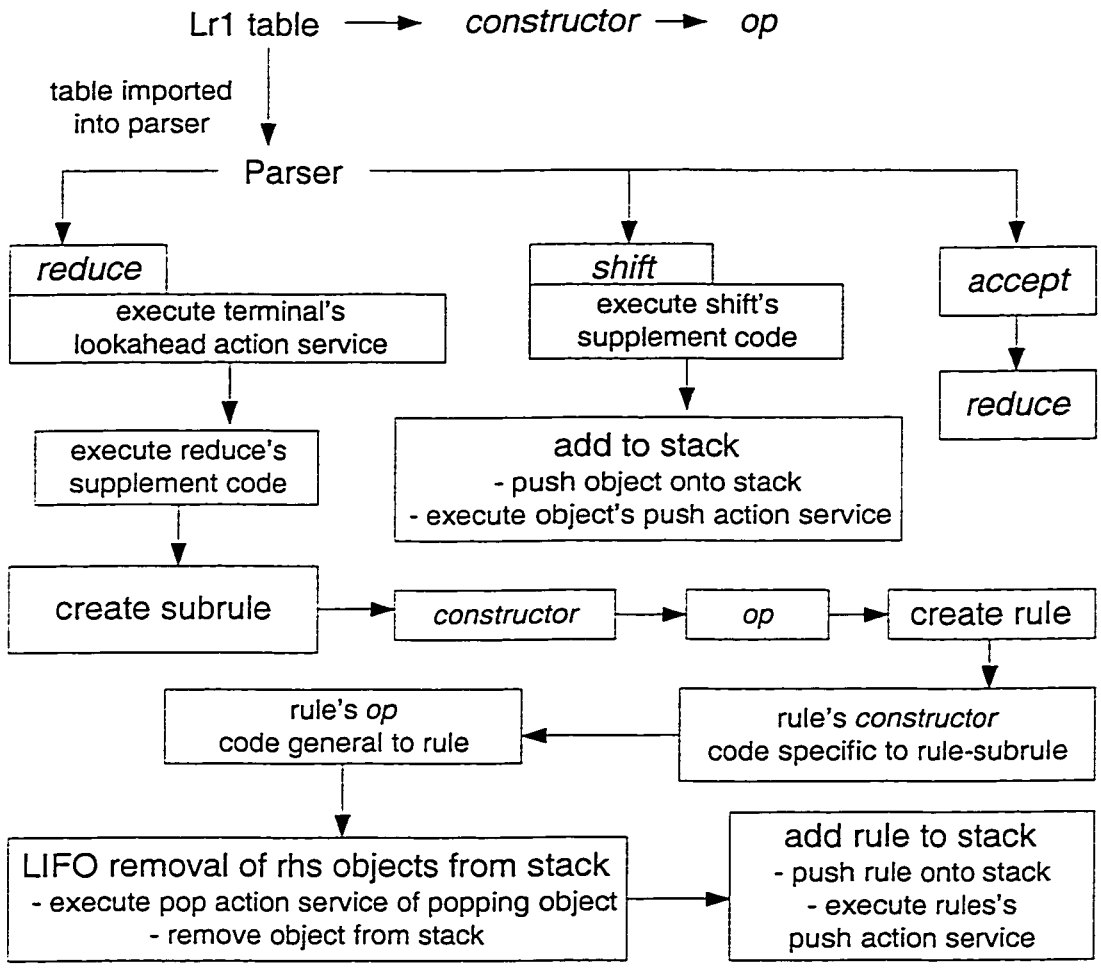

Figure 5.5: Finite state events to parsing objects

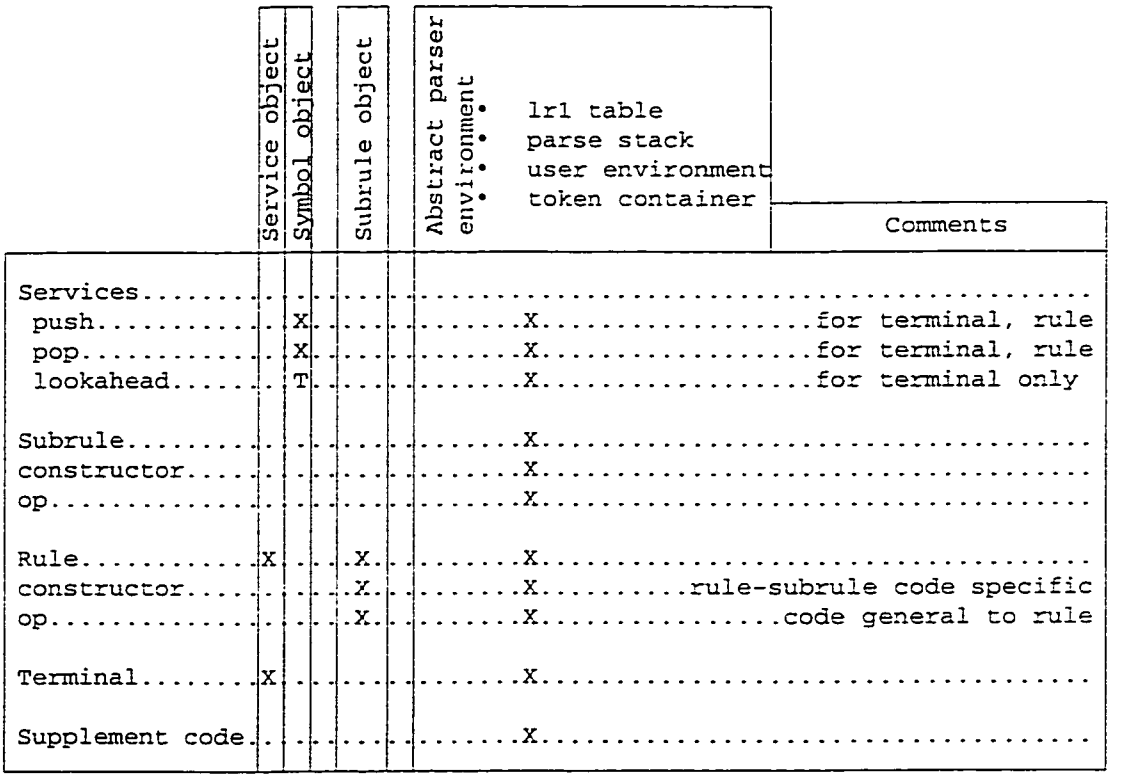

Figure 5.6: Run time information contexts available to parsing objects

Figure 5.6 is a 2-dimensional table whereby its rows describe the various objects in parsing, and the columns describe the contexts available to the row's objects. The last column is comments specific to the row's object. Unless noted, "X" in the row's column indicates availability. The "T" in some columns indicate the contexts available only to the Terminal's symbol object. The comment against the Rule's *constructor* indicates that the injected code is specific to its subrule-rule relationship having the "X" mark the contexts available for use. The comment against the Rule's *op* procedure indicates that the C++ code is general to the Rule.

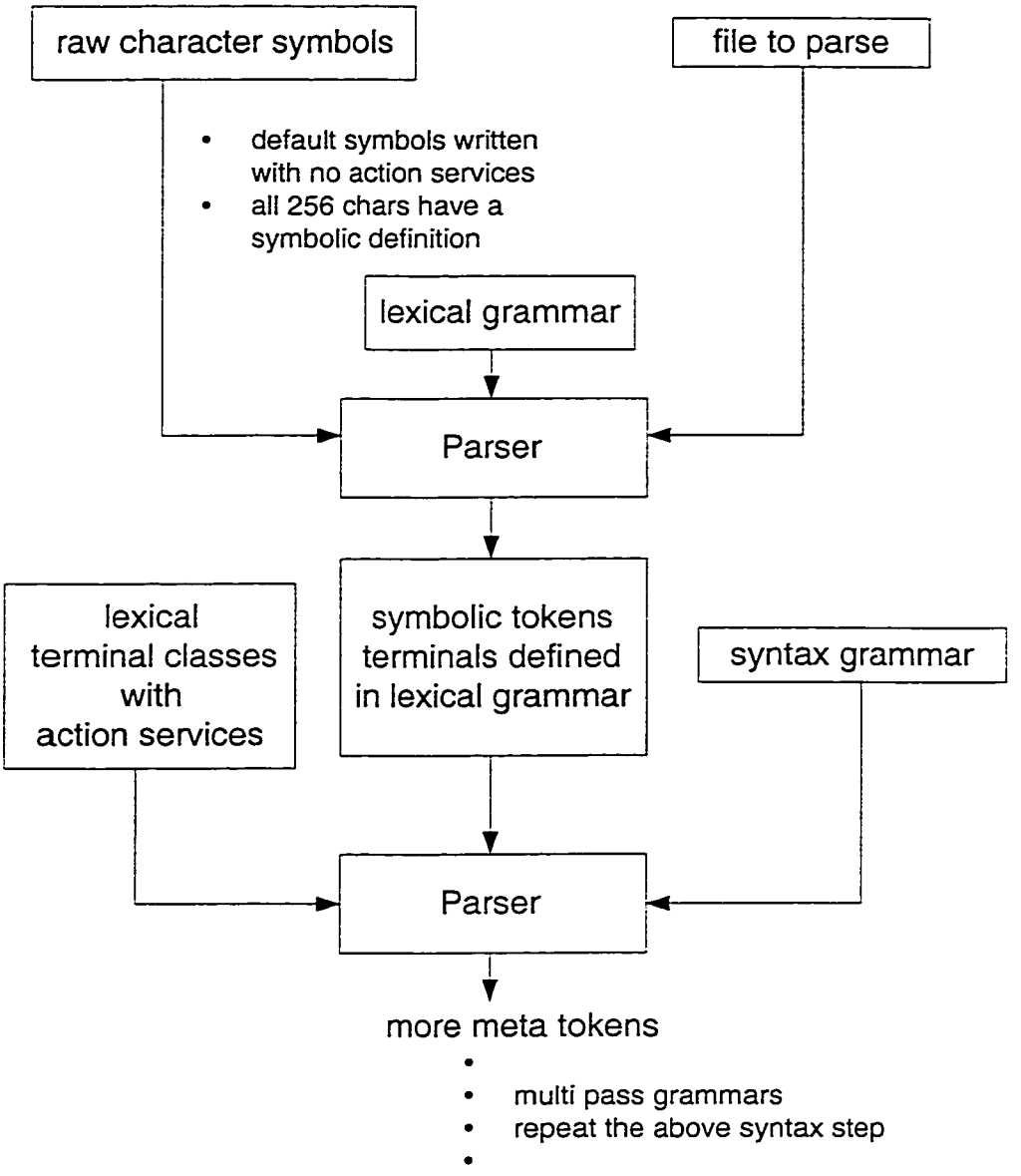

Figure 5.7: Run stages of multi pass grammars

Figure 5.7 gives a sketch of parsing using multiple passes. Shown in the boxes with comments are the common steps to parsing with many grammars. At the bottom of the diagram, references to meta-tokens are given. As explained previously in the "Winding down" section, meta tokens are tokens that are built using other tokens. They are the by-product of a grammar used in one of the passes.

Chapter 6

Future Research

Three things surfaced while writing the compiler/compiler code and using its outputted tables:

1. Grammars with object-oriented constructs provide an excellent framework of comprehension. It neatly ties all the entities together. One can think of it as the outgrowth from assembler to a high-level programming language with all its proclaimed strengths.

2. This sounds trite but don't take it as such: LR(1) grammars have their limitations — ambiguous grammars are hard to resolve. LR(1) grammars, as I found out, are not the be-all end-all to the grammar resolution technique. It's still easy to write ambiguous grammars. LR(1) is powerful but ambiguity resolution needs to be tackled differently. This is seen in the research extending the lookahead resolution to on-demand techniques [16].

3. Monolithic grammars are too hard to maintain. The size of the grammar with lots of rules produces a psychological clutter. A blurring of understanding can take place. They are also not re-usable. Somehow grammars should be broken into little pieces providing re-use and somehow assembled into a cohesive whole participating together.

In tasting my own output, items 2 and 3 forced me to rethink the approach to parsing.

Let us look at item 2 — ambiguity. From a bottom-up parsing perspective, there are

two types of ambiguity: shift / reduce, and reduce / reduce. I will not go into the theory behind it but I will pose the problem in a more general way: ambiguity comes down to a choice between two or more things competing for the same resource. Looked at this way, why not allow ambiguity to exist but have an arbitrator determine who should win. Now, item 3 provides another dilemma. When a population becomes too dense, one loses the autonomy of each resource. So, why not think of grammars as individual entities unto themselves and allow them to run semi-autonomously. From an operating system perspective, this is done with threads and guarded resources: critical regions, semaphores, etc.

Given these two thoughts, autonomy and arbitration, my future development is taking me into multi-threaded parallel parsing with arbitration. To do this, new constructs are presently being added to the grammar's language to support:

- arbitration at the grammar's rule level with C++ code injection support
- a regular expression to calculate lookahead boundaries at the parallel grammar level
- meta terminals used as signals returned back from parallel parsing threads
- a new meta symbol to represent parallelism. This symbol can be placed anywhere inside a subrule's string of symbols supplying the parallel grammar to-be-run and the meta terminal used for signaling a result.
- nested parallel grammar support
- a dispatcher of grammar threads in the parser with extensions to the LR(1) table supporting thread dispatch
- a communication protocol supporting the dispensing of tokens and the reporting of results from the grammar threads
- dispatch thread optimization using the viable prefixes of the parallel threads

This is quite a mouthful but it does resolve items 2 and 3. It handles the subset/super set problem: usually in a language's documentation, a grammar defines the keywords but the grammar employed is the superset defining a variable with post-processing using a keyword table lookup to resolve the variable. Well, parallel parsing handles this. And what about a literal? To properly process the literal, post-processing is used on the escape sequences inside it. Again, multi-parallel grammars solve this at the grammar level using nested grammars. Parallel parsing supports the modular development of grammars and their reuse. Using quasi context parsing, ambiguity resolution from a arbitration perspective lets the compiler writer play with his grammars in defining the language designed. It could become a fun experience in solving structural flow sequences in a non-deterministic way.

Chapter 7

Conclusion

The basic conclusion drawn from this study is classes represent well a grammar's components — rules, terminals, subrules — and various semantic actions associated with them. All the benefits of object-oriented discipline fit well into the grammar's implementation — inheritance, encapsulation, polymorphism, and reader comprehension. Ancillary actions are also well looked after using class definitions. Not addressed in the study is the run efficiency of symbol processing used with this compiler/compiler's implementation: the symbol's identity is used heavily in tracing and LR(1) table lookup. The thesis's emphasis was put on the feasibility of class use in translated grammars within a table-driven environment.

My experience from this project concurs with Dr. Stroustrup: a compiler/compiler is not quite powerful and flexible enough to design recognizers quickly — particularly if the language explored is tricky to define with a deterministic grammar. My problems dealt with resolving ambiguous grammars: this really burns up the patience of a language designer — somehow the grammar(s) must be refined enough to run in local contexts. This frustration is leading to future investigations into why the

LR(1) compiler/compiler is not defeating the language dragon — only keeping it at bay. Run-time ambiguity resolution using arbitration will be explored within a multi-threaded multi-grammared parallel-parsing environment.

In conclusion, applying the object-oriented discipline to a grammar's entities provides a comprehensive, tightly bound framework to table driven processing: this is the structural part to parsing. To paraphrase a rose: a yak is a yacc is a yac^{co_2}? Quite possibly using an object-oriented framework with better ambiguity resolving techniques, just maybe the language dragon might be tamed, respected but not conquered.

Bibliography

[1] *Microsoft Visual C++ Reference Volume 1 Class Library Reference.* Microsoft Corporation, One Microsoft Way, Redmond, WA, U.S.A., 1993.

[2] William A. Barrett, Rodney M. Bates, David A. Gustafson, and John D. Couch. *Compiler Construction Theory and Practice.* Science Research Associates, Inc, Toronto, Ont. Canada, 1986.

[3] Frank DeRemer and Thomas Pennello. Efficient Computation of LALR(1) Look-Ahead Sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, 1982.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++.* Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1990.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1994.

[6] J.E. Hopcroft and J.D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1969.

[7] S. C. Johnson. Yacc — Yet Another Compiler-Compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[8] Donald E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8(6):607–639, 1965.

[9] Bent Bruun Kristensen and Ole Lehrmann Madsen. Correspondence. *ACM Sigplan Notices*, 19(8), August 1984.

[10] M. E. Lesk. Lex — A Lexical Analyzer Generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[11] P. Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications ACM*, 6(1):1–17, 1963.

[12] David Pager. A Practical General Method for Constructing LR(k) Parsers. *Acta Informatica7*, 7:249–268, 1977.

[13] David Spector. Full LR(1) Parser Generator. *ACM Sigplan Notices*, 16(8), August 1981.

[14] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 1994. Section 3.3.2 Parsing C++ – comment on LALR(1) versus recursive descent.

[15] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, U.S.A., 3 edition, 1997.

[16] Russell W. Quong Terence J. Parr. LL and LR Translators Need k > 1 Lookahead. *ACM Sigplan Notices*, 31(2), February 1996.

[17] Peter Trott. Programming Languages Past, Present, and Future. *ACM Sigplan Notices*, 32(1). January 1997. What do you consider the most significant contribution to compiling?

Appendix A

The Grammar Language

A.1 Introduction

The language design borrows heavily from the object-oriented paradigm: class definitions, inheritance, encapsulation, and constructor-destructor execution control flows. These concepts are integrated into a modified Backus-Naur grammar form (BNF) [11]. For the syntax-directed code, the C++ language is used; its class's birth-run-death life cycle is designed into the grammar's structures. The complete C++ language can be used along with compiler directives, preprocessing macros and any other assorted code — remember, any code or class name for the emitted C++ program must observe C++'s naming conventions. The emitted files are pure C++ programs to be compiled by the compiler writer's C++ compiler. To help the compiler writer, other facilities are built into the grammar's language which are characteristic of syntax-directed bottom-up parsing:

- automatic execution of programmer defined objects during bottom-up actions. For example, when an object of rule or terminal is being pushed onto the stack, being popped off the stack, and when a terminal is the look-ahead boundary in recognizing the right-hand subrule. the associated action objects are automatically executed.

- stack flow tracing to aid in the debugging of a grammar. All stacked objects are symbolically traced relative to the stack position and its LR(1) table state.

- compiler writer supplemental operations for the parser to execute. These extra actions enhance the shift, reduce. accept, and error operations.

When using bottom-up parsing, well defined execution boundaries exist. Delineation happens at the parser's shift, reduce, and accept actions. To inject the compiler

writer's code into these execution points, code placement directives are used throughout the grammar. Depending on the context within the grammar, these directives place the code either into the definition header file or into its implementation file.

To introduce you to the language features, an informal top-down view of the language components is presented. To begin with, the language is divided into six major parts:

- The C++ code sections sprinkled throughout the grammar's structures. The code block has one of two purposes: declaratory for class definitions or implementation for class's code.
- The "fsm" section defining the LR(1) table class for the parser.
- The "parallel-parser" section — facilitates parallel parsing. This is for future development.
- The "fsm-operation" section — classes used to enhance the parser's normal operations.
- The "terminals" section defining the terminals in the language.
- The "rules" section defining the grammar of the language. Each rule has the accustomed BNF form extended with the annotated syntax-directed code.

Each language section is described in three stages. Stage one provides an example with running comments. Stage two gives a skeleton view of its components. Stage three describes the individual components.

A.2 @ — File Include Operator

The compiler/compiler has a file include operator denoted by "@". It must be in the first column of a line followed by the local operating system's file name ended by a line delimiter. The file name may be given as a string literal. The compiler/compiler supports 16 nested levels of file includes. The facility is used in the following test suite to include the common terminal definitions used across all the test grammars. Here is an example of a file include statement: @"d:Ctsuite.h".

A.3 C++ Code Sections

Because the compiler/compiler program emits plain C++ code, the compiler writer can include any valid C++ code and preprocessor directives into the code sections.

These code snippets get injected into the emitted files at the spots directed by code placement directives. Regardless where the code section is, the code's intent is always one of definition or implementation. To give a consistency of meaning to the directives, keywords indicate either declaratory or implementation intent. The fixed base of words are the norm to the directives. In special cases, compound directives are built from these words. The base directives are:

- "declaration" — directive putting code into the C++ header file. Depending on its context — usually compiler writer support definitions, the code might be part of a class definition or by itself.
- "implementation" — directive putting code into the C++ implementation file. This is the complement to the "declaration" directive.
- "constructor" — directive putting code into the class's constructor C++ implementation.
- "destructor" — directive putting code into the class's destructor C++ implementation.
- "op" — directive putting code into the class's op method C++ implementation. This method gets called always after the class's constuctor is finished executing. It is used in tracing and as a boundary point for the compiler writer to execute code between the class's constructor and destructor executions.

All code block sections are optional as are their individual directives inside the code block. The syntax is the same for all code sections. The code block is defined as follows:

- { to start the code block
- a list of code directives and their associated C++ code. All directives use this format:
  - the directive's keyword to introduce the code block of the directive
  - the C++ code
  - *** to close the directive's code block
- }; to close the code block

There is no input order to the directives. The compiler writer uses his own style of ordering. As one builds up C++ code, the grammar starts to take on a bloated appearance. At present, only a file include facility is supported. In the future, a hypertext editor will be created.

The following sub-sections catalog each of the directives removed from their contexts. Later within each language structure, their appropriate meanings are described.

A.3.1 user-prefix-declaration Directive

The "user-prefix-declaration" directive is used in the "fsm-class"'s code block to add foreign code to the emitted header file. This code could be a library definition or the compiler writer's own support functions that are used by the grammar but are not manufactured into the emitted grammar's objects. The "declaration" in the directive indicates its intent to be attached to the emitted header file. The "prefix" in the directive indicates its placement at the start of the emitted header file.

A.3.2 user-suffix-declaration Directive

The "user-suffix-declaration" directive is used in the "fsm-class"'s code block to add code to the emitted header file. The "declaration" in the directive indicates its intent to be attached to the emitted header file. The "suffix" in the directive indicates its placement at the end of the emitted header file. The same comments apply to the code character as the "user-prefix-declaration" directive above.

A.3.3 user-declaration Directive

The "user-declaration" directive is used to add code to a class definition. The following grammar contructs are the contexts of use:

- fsm's "fsm-class" context
- a terminal definition context — "sym-class", "sym-look-ahead", "stack-pushed-rtn", "stack-popped-rtn"
- a rule definition context — "sym-class", "lhs", "stack-pushed-rtn", "stack-popped-rtn". The "lhs" directive is for general code insertion used throughout the rule's existence. Generally, the code gets run after the rule's specific contructor.

The subrule is not included in the above list due to the interplay between itself and its rule: its variation is described below with its directives beginning with a "rhs-" prefix.

A.3.4 user-implementation Directive

Implementation code for the "user-declaration" directive's code.

A.3.5 constructor Directive

Code injected into all of the class constructors implementations. It is general to all constructors in the code block context.

A.3.6 destructor Directive

Code injected into the class's destructor implementation.

A.3.7 op Directive

Code injected into the class's op procedure implementation.

A.3.8 rhs-user-declaration Directive

The "rhs" stands for right-hand-side indicating a directive for a subrule. The "rhs-user-declaration" directive declares code for the subrule's class definition. It is the compiler writer's code augmenting the subrule class's functionality.

A.3.9 rhs-user-implementation Directive

Implementation code for the "rhs-user-declaration" directive's code.

A.3.10 rhs-constructor Directive

Code injected into the subrule's class constructor implementation.

A.3.11 rhs-destructor Directive

Code injected into the subrule's class destructor implementation.

A.3.12 rhs-op Directive

Code injected into the subrule's class op procedure implementation.

A.3.13 lhs-constructor Directive

Code injected into the rule's constructor class implementation. There is an individual "rule constructor" for each of its subrules defined so that the code is specific to each subrule-rule pairing reduce operation.

A.4 Fsm Section

The "fsm" plays the role of a wrapper class in the emitted code. It supplies the LR(1) table name that gets passed to the parser as a class object along with the compiler writer's own run environments. All six parameters in its body are required each having one of two purposes:

- C++ class name for the LR(1) table.
- general information about the grammar: this is the where, when, and about data — file name used to compile the table, version number, general comments and the like.

The "fsm" uses a call procedure format with the parameters delimited by commas. Each parameter has two parts:

- a keyword attribute identifying the parameter
- an associated data value. The data value is required; it is a literal string enclosed in quotes for all the parameters except the "fsm-class" parameter which is a C++ class name.

A.4.1 Example

The example shows C++ compiler directives for the emitted code preventing multiple definition errors. Because the emitted files are just C++ programs, the compiler writer can use any valid C++ language construct or directive as seen in this example.

```
fsm              // keyword identifying the grammar's contruct
 (fsm-id         "char_literal.lex"
 ,fsm-class      Clr_char_literal_lexer
    {            // code block section
      user-prefix-declaration
       #ifndef __lr_char_literal_lexer_h__
       #define __lr_char_literal_lexer_h__
       #include "clr_pass1_lexer.h"
      ***
      user-suffix-declaration
       #endif
      ***
    };
 ,fsm-version    "1.0"
 ,fsm-date       "8-oct-97"
 ,fsm-debug      "no"
 ,fsm-comments   "parallel parser of character literal"
);
```

A.4.2 Skeleton View

```
fsm
 (fsm-id          "xxx"        // descriptive string
 ,fsm-class       xxxx         // C++ class name of LR(1) table
     <C++ code block>
     <see definition in individual components>
 ,fsm-version    "1.0"         // descriptive C++ string
 ,fsm-date       "23-jun-97"   // descriptive C++ string
 ,fsm-debug      "no"          // yes , no for tracing
 ,fsm-comments   "xxxxxxxxx"   // descriptive C++ string
 );
```

A.4.3 Individual Components

A.4.3.1 fsm-id Attribute

fsm-id "<value>"

The "fsm-id"'s parameter identifies the grammar file name used to compile the table. The string value is the local operating system's file name. This value is included in the class definition and the main purpose is general information about the table. Built within the emitted LR(1) table class is its access method of the same name without its "fsm-" prefix.

A.4.3.2 fsm-class Attribute

fsm-class <C++ class name for lr(1) table> <Optional C++ Code>

The "fsm-class"'s parameter is the C++ class name for the emitted LR(1) table. Accompanying this is the optional code block for the "fsm" wrapper class.

```
{// optional code block
  user-prefix-declaration // directive - code added at the
                          // beginning of emitted header file
     // C++ code
  ***
 //.....*** ..... means C++ code, and *** ends code block
  user-suffix-declaration // directive - code added at the
   .....***               // end of the emitted header file
  user-declaration        // directive - code added to the
   .....***               //  fsm class in the header file
  user-implementation     // directive - code added to the
   .....***               // emitted implementation file
  constructor             // directive - code injected into
```

```
    .....***                // fsm class's constructor implementation
  destructor                // directive - code injected into
    .....***                // fsm class's destructor implementation
  op                        // directive - code injected into
    .....***                // fsm class's op method implementation
};
```

The running C++ comments above provide the intent and placement in the emitted code.

A.4.3.3 fsm-version Attribute

fsm-version "<value>"

The "fsm-version"'s parameter identifies the grammar's version. The string value is up to the language designer fancy. This value is included in the class definition as general information about the table. Built within the emitted class is its access method of the same name without its "fsm-" prefix.

A.4.3.4 fsm-date Attribute

fsm-date "<value>"

The "fsm-date"'s parameter identifies when the grammar was designed. Same comments apply about it worth described above in the "fsm-version" value.

A.4.3.5 fsm-debug Attribute

fsm-debug "<value>"

The "fsm-debug"'s parameter allows the turning on and off of the dynamic tracing ability built into the parser. Tracing depends on the default parser's action routines used by the compiler writer. The string value is either "yes" or "no". This tracing ability is only in the debug version of the parser's action routines. This is controlled by the symbol "_DEBUG". The production version normally omits this conditional code by not defining this symbol. The emitted "fsm" class has a read method for this variable of name debug. You turn on or off the debug facility by the parse class's debug parameter defined in the constructor. Its default is the value given in the grammar definition — see the constructor in "A Real Lr1 Table Class Definition" page 57.

A.4.3.6 fsm-comments Attribute

fsm-comments "<value>"

The "fsm-comments" 's parameter is a general description about the grammar definition. Same comments apply about it worth as described above in the "fsm-version" value.

A.5 Parallel-parser Section

Omitted — this is for future research.

A.5.1 Example

```
parallel-parser
( parallel-thread-function{
    UINT PF_char_lexer(LPVOID Parm);
  };
  parallel-la-boundry{
    eolr // regular expression of terminals and rule's LA
  };
  code
    $$parallel_parser_code$$(PF_char_lexer,Clr_char_lexer)
  ***
);
```

A.5.2 Skeleton View

Omitted — this is for future research.

A.5.3 Individual Components

Omitted — this is for future research.

A.6 Fsm-Operation Section — Supplemental Operations

The "fsm-operation" supplies class names of objects complementing the parser's behaviors and the operation the generic parser is to perform — for example, shift, reduce, and accept. This facility allows the compiler writer to import his own action routines;

it is very rudimentary. At present, I have not really used this feature — apart from identifying the object in tracing, and I cannot comment as to its usefulness. These action classes get automatically created in the LR(1) table class contructor and are triggered by the generic parser's operations. The "fsm-operation" uses a call procedure format with the parameters delimited by commas. Each parameter is composed of two parts:

- a keyword identifying the parser's action
- the compiler writer's C++ class name for that operation

The default operations have tracing facilities built into them. If the generic parser is not to the taste of the compiler writer, one can always write his own abstract parser with its own operations.

A.6.1 Example

This example is usually the defaults used by the compiler writer.

```
fsm-operation
(fsm-operation-shift-class  fsm_Shift
,fsm-operation-reduce-class fsm_Reduce
,fsm-operation-accept-class fsm_Accept
,fsm-operation-error-class  fsm_Error
);
```

A.6.2 Skeleton View

```
fsm-operation    // identifies the section
 (fsm-operation-shift-class    aaa // C++ class name
 ,fsm-operation-reduce-class   bbb // C++ class name
 ,fsm-operation-accept-class   ccc // C++ class name
 ,fsm-operation-error-class    ddd // C++ class name
 );
```

A.6.3 Individual Components

A.6.3.1 fsm-operation-shift-class Attribute

fsm-operation-shift-class <C++ class name>

The "fsm-operation-shift-class" provides the C++ class name to enrich the parser's

shift operation. A default "fsm_shift" class with its source comes with the LR(1) system. The compiler writer can use the source code as a model in writing his own enriching operation. The model is very rudimentary.

A.6.3.2 fsm-operation-reduce-class Attribute

fsm-operation-reduce-class <C++ class name>

The "fsm-operation-reduce-class" provides the C++ class name to enrich the parser's reduce operation. A default "fsm_reduce" class with its source comes with the LR(1) system. The compiler writer can use the source code as a model in writing his own operations. The model is very rudimentary.

A.6.3.3 fsm-operation-accept-class Attribute

fsm-operation-accept-class <C++ class name>

The "fsm-operation-accept-class" provides the C++ class name to enrich the parser's accept operation. A default "fsm_accept" class with its source comes with the LR(1) system. The compiler writer can use the source code as a model in writing his own operations. The model is very crude.

A.6.3.4 fsm-operation-error-class Attribute

fsm-operation-error-class <C++ class name>

The "fsm-operation-error-class" provides the C++ class name to enrich the parser's error operation. A default "fsm_error" class with its source comes with the LR(1) system. It has a very limited error handling capability. It just traces out the stack and offending look ahead token and lets the generic parser do want it wants which is stop processing. The compiler writer can use the source code as a model in writing his own operations. The model is very rudimentary.

A.7 Terminals Section

This section defines the grammar's terminal vocabulary. Each terminal definition gets emitted as a C++ class definition. The compiler writer can add anything to its definition and implementation. Built into its definition are keyword directives that can be used for syntax-directed purposes.

To provide an integration between the lexical recognizer and the syntax parser, the compiler/compiler emits the lexical table as a push down automata. The generic parser is used in both phases of compiling. This is one way to chain together parsing stages where one phase's output tokens becomes the next phase's input tokens. Multi-staging terminals can be built with each pass provided by multiple grammars. Each grammar shares the terminal vocabulary by using the file include operator; their emitted code would also use the common emitted C++ terminal header file produced usually by the lexical grammar.

With its class definitions and the sharing of the terminal header file amongst the different grammars, the compiler writer has a better grasp of the grammar's run phases. Because the terminals are classes, the compiler writer's code references the token as a class object. The terminals section is defined as:

- terminals — keyword introducing the terminal vocabulary section
- { to start the terminals to be defined.
- a list of terminal definitions. The terminal definition template has the form:
  - the terminal's symbolic name
  - AD optional automatic delete attribute
  - (to start the terminal's attributes section
    - a list of attributes where each attribute has the form:
      - keyword identifying the attribute
      - C++ class name
      - an optional C++ code section. The code block has the same format as previously described.
  -); to end the terminal's definition
- }; to end the terminal vocabulary section

A.7.1 Example

The following example shows the flexibility that the compiler writer has at his disposal in defining terminal objects. Default definitions, stack tracing, code directives, and C++ code are shown.

```
terminals    // identifies the section
{
  eolr   (sym-class          Sym_eolr);
  "|||"  (sym-class          Sym_parallel_operator);
```

```
  "}"    (sym-class          Sym_close_brace);
  "\x7f" (sym-class          Sym_del);
  a      (sym-class          Csymbol_a
         ,sym-look-ahead-rtn Csymbol_a_look_ahead_rtn
         ,stack-pushed-rtn   Csymbol_a_pushed_rtn
         ,stack-popped-rtn   Csymbol_a_popped_rtn);
"string-literal" AD
  (sym-class          Sym_string_literal
    {
     user-declaration{
       public:
         // additional constructor needed by compiler writer
         Sym_string_literal
           (CObList* List_conduits
           ,char     Char_len
           ,const char* Id="string-literal");
         Sym_string_literal(const char* Id="string-literal");
         CObList* string_conduits();
         char char_len();
       private:
         CObList* string_conduits_;
         char     char_len_;
     };
     destructor{
       if (string_conduits_ != NULL) delete string_conduits_;
     };
     user-implementation{
       Sym_string_literal::Sym_string_literal
         (CObList* List_conduits
         ,char Char_len
         ,const char* Id)
         :CAbs_Clr_pass1_lexer_terminal_sym(NULL,NULL,NULL,Id)
         ,string_conduits_(List_conduits)
         ,char_len_(Char_len){};
       Sym_string_literal::Sym_string_literal
         (const char* Id)
         :CAbs_Clr_pass1_lexer_terminal_sym(NULL,NULL,NULL,Id)
       ,string_conduits_(NULL)
       ,char_len_('0'){};
       CObList* Sym_string_literal::
       string_conduits(){return string_conduits_;};
       char Sym_string_literal::char_len(){return char_len_;};
     };
    }; // end of "string-literal" code block
  );   //end of "string-literal" definition
};     // close terminals
```

A.7.2 Skeleton View

```
terminals    // identifies the terminal vocabulary
{// each terminal uses this definition template
  <symbolic terminal name>  is one of xxx or literal \Cppb string
  AD optional automatic delete indicator: if turned on parser deletes terminal
    (        // start of terminal's attributes section
       sym-class             aaa    // C++ class name of the terminal
         {   // optional sym-class syntax directed code section
             // syntax directed code directives
         };
       // Following are optional parameters
       ,sym-look-ahead       bbb    // C++ class name
             // optional sym-look-ahead code section
       ,stack-pushed-rtn     ccc    // C++ class name
             // optional stack-pushed-rtn code section
       ,stack-popped-rtn     ddd    // C++ class name
             // optional stack-popped-rtn code section
    );       // end of terminal definition
             ....
      Repeat of above template for each additional
      terminal definition
             ....
};
```

A.7.3 Terminal Definition of Individual Components

A.7.3.1 Symbolic Name of Terminal

<Symbolic Name of Terminal>

The symbolic name is the reference to the terminal within the defining grammar. The compiler writer uses this name throughout his grammar rules. It can be a literal string with all C++'s possible variations or a single word. Some of these variations are shown in the above example — "\x7f", a, "string-literal", "|||". Using the terminal in a rule's definition must be consistent with its symbolic name: if it is in quotes then reference it as such. The symbolic name is also used in the tracing of the grammar, and as the key for the parser's LR(1) table lookup routine. In defining the symbolic string, the compiler writer can use tricks to lookup keywords that might conflict with the keywords of this grammar definition. For example, the symbolic key for the "fsm" keyword could be defined as a quoted string composed of the fsm word suffixed with a space. This technique is employed by the author in writing the self defining grammar of this language. Remember, it is a symbolic representation; its physical value is recognized elsewhere — during the lexical phase — and figuratively

referenced thereafter.

A.7.3.2 Automatic Delete Attribute

AD

This parameter is optional and has only one value — AD. It indicates whether the parser should delete the terminal when it is popped from the parse stack. It aids the compiler writer in the managing of memory leaks of his grammar's symbols — terminals and rules. This is because each grammar's symbol comes into existence as a pointer of a class object.

A.7.3.3 sym-class Attribute

sym-class <C++ class name> <Optional C++ Code>

The "sym-class" provides the emitted C++ class name for the terminal. It is required and it is the reference name used in the C++ code — compiler writer's code blocks. Additional constructors can be defined using the "user-declaration" directive defined earlier. The emitted class inherits an abstract base class for all terminals. The compiler writer is free to add any functionality to the terminal.

```
{// optional sym-class C++ code block
  user-declaration        // directive -- code added to the terminal
                          // class definition in the header file
     // C++ code
  ***
 //.....*** ..... means C++ code, and *** ends code block
  user-implementation     // directive -- code added to the emitted
   .....***               // implementation file
  constructor             // directive -- code injected into terminal
   .....***               //  class's constructor implementation
  destructor              // directive -- code injected into terminal
   .....***               // class's destructor implementation
  op                      // directive -- code injected into terminal
   .....***               // class's op method implementation
};
```

The running C++ comments above provide their intent and placement in the emitted code.

A.7.3.4 sym-look-ahead-rtn Attribute

sym-look-ahead-rtn <Optional C++ class name> <Optional C++ Code>

The "sym-look-ahead-rtn" is an action class that automatically is executed by the parser when this terminal is used as the look ahead symbol in the reduce operation of the parser. It gets wedged into the terminal's defining class as a referenced variable with an access method of the same name. The "sym-class" provides the emitted C++ class name for the action service used when the terminal is the boundary symbol used to collapse the subrule into its rule. It is optional along with the code block. The same code block format defined in the terminal's "sym-class" applies.

A.7.3.5 sym-pushed-rtn Attribute

sym-pushed-rtn <Optional C++ class name> <Optional C++ Code>

The "sym-pushed-rtn" is an action class that automatically is executed by the parser when this terminal is pushed onto the parse stack. It gets wedged into the terminal's defining class as a referenced variable with an access method of the same name. The "sym-class" provides the emitted C++ class name for the action service used when the terminal is pushed onto the parse stack. It is optional along with the code block. The same code block format defined in the terminal's "sym-class" applies.

A.7.3.6 sym-popped-rtn Attribute

sym-popped-rtn <Optional C++ class name> <Optional C++ Code>

The "sym-popped-rtn" is an action class that automatically is executed by the parser when this terminal is popped from the parse stack. It gets wedged into the terminal's defining class as a referenced variable with an access method of the same name. The "sym-class" provides the emitted C++ class name for the action service used when the terminal is popped off the parse stack. It is optional along with the code block. The same code block format defined in the terminal's "sym-class" applies.

A.8 Rules Section

This section defines the grammar's rule vocabulary. The "start rule" of the grammar is the first rule defined in the section. Each rule definition gets emitted as a C++ class along with each of its right-side subrules. The compiler/compiler program emits the class definitions for each rule and their subrules, and ties together their relationships

inside the "fsm" class definition using enumerated types. In the normal flow of syntax recognition, the rule's right-hand-side gets recognized before its defining rule is reduced. The same order holds for execution of the syntax-directed code. As in the definition of a terminal, the compiler writer can add anything to the rule definition, its subrule definitions, and all their implementations.

The rules section is defined as:

- rules — keyword introducing the start of the rules vocabulary
- { to start the rules to be defined.
- a list of rule definitions. The rule definition template has the form:
  - the rule's symbolic name. This name is used to reference it throughout the defining grammar.
  - AD optional automatic delete attribute
  - (to start the rule's attributes section
    - a list of attributes where each attribute has the form:
      - keyword identifying the attribute
      - C++ class name except the "lhs" attribute which uses the rule's symbolic name
      - an optional C++ code section. The code block has the same format as previously described.
  -) to end the attributes section
  - { to start the subrule definitions
    - a list of subrules where each subrule has the form:
      - -> to start a subrule definition
      - the subrule's mixed sequence of terminals and rules symbolic names. The sequence can be empty.
      - an optional C++ code block. Due to the relationship between the the subrule and its defining rule, the code block introduces some new directives.
  - } to end the attributes section
- }; to end the rules section

A.8.1 Example

The following example shows the flexibility the compiler writer has in defining rule objects. It shows just one start-rule and its subrules accepting keyword definitions and emitting their results as new terminal objects defined by the "sym-class" in their terminal definition.

```
rules{
$p2_keywords
 (sym-class Sym_$p2_keywords
 ,lhs
  // C++ code block
   user-declaration
    public:  CAbs_lr1_sym* key_word();
    private: CAbs_lr1_sym* key_word_;
   ***
   user-implementation
     CAbs_lr1_sym* $p2_keywords::key_word(){return key_word_;};
   ***
   op
     Parser_env->parallel_accept_conduit(key_word());
   ***
  };
 ,stack-pushed-rtn
 ,stack-popped-rtn )
 {
   -> $ c + + $
      {lhs-constructor key_word_ = new Sym_c2_start_blk;***};
   -> $ c "-" "-" $
      {lhs-constructor key_word_ = new Sym_c2_stop_blk;***};
   -> "/" "/"
      {lhs-constructor key_word_ = new Sym_slash2_comment_start;***};
 };// end of $p2_keywords
};// end of rules vocabulary
```

A.8.2 Skeleton View

```
rules     // identifies the rules vocabulary section
{  // list of rule templates with a format of:
  <symbolic rule name>  a valid C++ name started with a $
  AD an optional automatic delete attribute
    (
       sym-class              aaa    // C++ class name
         {// optional sym-class syntax directed code section
           // syntax directed code directives
         };
```

```
        // Following are optional parameters
        ,lhs // NO C++ class name: the <symbolic rule name> defines it!
          // optional lhs code section
        ,sym-look-ahead-rtn   bbb    // C++ class name
          // optional sym-look-ahead code section
        ,stack-pushed-rtn     ccc    // C++ class name
          // optional stack-pushed-rtn code section
        ,stack-popped-rtn     ddd    // C++ class name
          // optional stack-popped-rtn code section
    )
    {// list of subrule templates with a format of:
      -> possible empty sequence of
         symbolic names of rules and terminals
         <optional C++ syntax directed code block>
               ....
      repeat subrule template to define additional subrules
               ....
    };
  .... repeat rule template to define additional rules ....
};
```

A.8.3 Rule Definition of Individual Components

A.8.3.1 Symbolic Name of Rule

<Symbolic Name of Rule>

The symbolic name is the reference to the rule within the defining grammar. The compiler writer uses this name throughout his grammar rule's right-side phrases. It must be a valid C++ class name. This is important because it is the emitted rule's class name and it is part of each subrule's class name. To make life easy for the compiler/compiler writer and for the reader of a grammar, rules must start with the "$" sign. The compiler/compiler emits the subrule's class name with a "rhs#_" prefix and the rule's symbolic name. The "#" sign in the name is the subrule's relative position inside the rule. If there are three subrules being defined, the "#" sign in each subrule name would be replaced with the number one, or two, or three noting relative position. The symbolic name is also used in the tracing of the grammar, and as the key for the parser's LR(1) table lookup routine.

A.8.3.2 Automatic Delete Attribute

AD

This parameter is optional and has only one value — AD. It indicates whether the parser should delete the rule when it is popped from the parse stack. It aids the compiler writer in the managing of memory leaks of his grammar's symbols — terminals and rules. This is because each grammar's symbol comes into existence as a pointer of a class object.

A.8.3.3 sym-class Attribute

sym-class < C++ class name > < Optional C++ Code Block >

The "sym-class" provides a wrapper class for the stack action routines of the rule. It must be defined even though a wrapper class might be empty of its stack routines. Accompanying this is the optional C++ code block for the defining rule class. The compiler writer is free to add any other functionality to the rule.

```
{ // optional sym-class C++ code block
  user-declaration        // directive -- code added to the rule
                          // class definition in the header file
     // C++ code
  ***
 //.....*** ..... means C++ code, and *** ends code block
 user-implementation     // directive -- code added to the emitted
   .....***               // implementation file
  constructor             // directive -- code injected into rule
   .....***               //  class's constructor implementation
  destructor              // directive -- code injected into rule
   .....***               // class's destructor implementation
  op                      // directive -- code injected into rule
   .....***               // class's op method implementation
};
```

The running comments above provide their intent and placement in the emitted code.

A.8.3.4 lhs Attribute

lhs < Optional C++ syntax directed code >

"lhs" is the rule's definition body and stands for left-hand-side. It has no C++ name parameter because this is supplied by the symbolic name of the rule. The "lhs" directive does one thing:

- it directs the code for the rule's class being defined. The code is general to the rule and not specific to its subrules. The specific code is defined in each subrule code block using its "rhs" and "lhs" prefixed directives.

The same comments apply to the code block format as described before.

```
{// optional lhs C++ code block
  user-declaration        // directive -- code added to the rule
                          // class definition in the header file
     // C++ code
  ***
  //.....*** ..... means C++ code, and *** ends code block
 user-implementation      // directive -- code added to the emitted
  .....***                // implementation file
  constructor             // directive -- code injected into rule
  .....***                //  class's constructors implementation
                          // this is across all its constructors
  destructor              // directive -- code injected into rule
  .....***                // class's destructor implementation
  op                      // directive -- code injected into rule
  .....***                // class's op method implementation
};
```

The running comments above provide their intent and placement in the emitted code.

A.8.3.5 sym-pushed-rtn Attribute

sym-pushed-rtn < C++ class name > < Optional C++ Code Block >

The "sym-pushed-rtn" is an action class that automatically is executed by the parser when this rule is pushed onto the parse stack. It gets wedged into the rule's "sym-class" wrapper class as a reference variable with an access method of the same name. The rule's "sym-class" comments on the class name and optional code block apply to "sym-pushed-rtn"'s code block.

A.8.3.6 sym-popped-rtn Attribute

sym-popped-rtn < C++ class name > < Optional C++ Code Block >

The "sym-popped-rtn" is an action class that automatically is executed by the parser when this rule is popped from the parse stack. It gets wedged into the rule's "sym-class" wrapper class as a reference variable with an access method of the same name. The rule's "sym-class" comments on the class name and optional code block apply to "sym-popped-rtn"'s code blocks.

A.8.3.7 Subrule Definition

-> Subrule Definition < Optional C++ syntax directed code >

The subrule's definition is introduced by a "->" — consider it a subrule definition operator. The subrule is composed of a mixed sequence of terminals and rules symbolic names. This sequence can be empty — normal usuage defines the empty subrule first.

The < Optional C++ syntax directed code > is optional. When used, it has some additional code directives relating the subrule to its defining rule. The subrule's code block takes the same form as defined above with refinements to the directives being:

- "rhs-" prefixes the directives indicating that the code is for the subrule's class. The mnemonic stands for right-hand-side.
- "lhs-" prefixes an extra constructor directive. This directive is for the defining rule's specific constructor that gets called when this subrule is being reduced. The mnemonic stands for left-hand-side.

Because the rule and its subrules are emitted as individual classes, these directives indicate which class to inject the code into. The format is:

```
{ // optional subrule's syntax directed code block
  rhs-user-declaration    // directive -- code added to the subrule
                          // class definition in the header file
     // C++ code
  ***
 //.....*** ..... means C++ code, and *** ends code block
  rhs-user-implementation // directive -- code added to the emitted
   .....***               // subrule's implementation file
  rhs-constructor         // directive -- code injected into subrule
   .....***               //  class's constructor implementation
  rhs-destructor          // directive -- code injected into subrule
   .....***               // class's destructor implementation
  rhs-op                  // directive -- code injected into subrule
   .....***               // class's op method implementation
  lhs-constructor         // directive -- code injected into rule
   .....***               //  class's specific constructor
                          // implementation. Note: rule has a
                          //  constructor per defined subrule
};
```

The running C++ comments above provide their intent and placement in the emitted code.

Appendix B

IDOW Pattern — Iterator Dispatching Objects for Workers

From [5] the following style is used to describe patterns. In general, this idiom is used in published documents; it is not a hard-and-fast rule but it is a reasonable starting point of expectations towards documented patterns.

B.1 Intent

To provide a variable framework structuring code-specific procedures to be executed in some distributed fashion against related data objects. The pattern enforces modular programming.

B.2 Reader's Pre-requisites

A general knowledge of Object-oriented paradigm, familiarity with [5] , and the C++ programming language.

B.3 Motivation

From my experience with some individual patterns Visitor, Iterator, and Strategy described in [5] , the following programming idiom kept occurring:

- with any bag of data objects varying in composition, a traversal method is required.
- the traversal is done within a specific context.

- upon access, the object's associated code-procedure is executed.

- keep the local code-to-be-executed stand alone for maintenance ease and reader comprehension.

- support general work to be done against all objects

- allow the possibility for co-operative signaling between the dispatch and the iterator.

This pattern achieves the above points in a coherent way. In executing the pattern, it is the iterator that co-ordinates this: all the other activities are hidden from the programmer. The principle idea behind the pattern is iteration drives the visiting of each object that needs work done. Iterator knows the route to take through the objects; it is the navigator following its pre-planned route. Upon accessing the object, the iterator calls dispatch. The dispatcher receives the object needing work; it now co-ordinates, with its pool of workers, who is assigned to work on the accessed object.

As a memory aid, the pattern's name is an acronym of its composing objects:

- an iterator, a dispatch-of-workers, objects of varying types, and workers.

B.4 Comments

The concrete iterator travels over the objects in a constant way — for example, a postfix traversal of a tree. To traverse the tree in prefix order, a prefix iterator or a parameterized iterator would be written. Once written, the iterator is re-useable for any local processing to take place. The local processing is provided by the dispatch-of-workers which is imported into the iterator along with the structure to traverse. The iterator executes and calls the dispatcher giving it the object to be worked on.

The dispatcher then gives his pool-of-workers to the object; the object selects and engages the artisan to do the commission. The control flow is the same as the letters in the pattern name: Iterator calls Dispatch, Dispatch calls Object, Object calls Worker. Double dispatch is between Dispatch to Object, and Object to Worker.

Each object's worker provides the code isolation from the other data object types. It is this division of labour that makes this pattern simple in coding out the problem. This code-localization protects the programmer from conceptual overload. It also forces the programmer to think about the execution flow determining the code-procedures logic.

B.5 Uses

Here are some highlights describing IDOW variations:

1. normal use:

    - traverse a complete structure executing the local-code per object type. The execution of the pattern finishes when the iterator reads all the data objects.

2. general work:

    - traverse a complete structure using one general contractor to do the work. The default "pool of workers" is a general contractor; the same work is applied to each object — for example, just delete the object. The execution of the pattern finishes when the iterator reads all the data objects.

3. co-operative signaling between the dispatch-of-workers and the iterator:

    - the dispatch has public methods for signaling and the specific iterator tests the signal after each dispatch call. It usually is the worker-specific procedure that sets the signal by calling the dispatch's public method. The signal gets raised somewhere within the data objects being traversed. The signal can be of any object type: for example, a new object structure returned for the iterator to traverse, or a simple signal as stop-your-iterating. The iterator tests the signal and reacts accordingly. Usually when used this way, the iterator and the worker are of one-time-only use. The iterator re-use is locked to the specific dispatch-of-workers. This is due to the iterator's explicit knowledge of the dispatch-workers's signaling procedures which are not abstractly defined.

4. no iteration required:

    - The iterator is eliminated and only one object is dispatched for code-execution. The dispatch's call_dispatcher method is called directly with the one object at hand.

B.6 Applicability

Use the IDOW pattern when many code visits are needed against the to-be-iterated data structure: i.e. polymorphic code-workers. For example compilers have multiple passes on the abstract tree; each pass comprises a code context for execution.

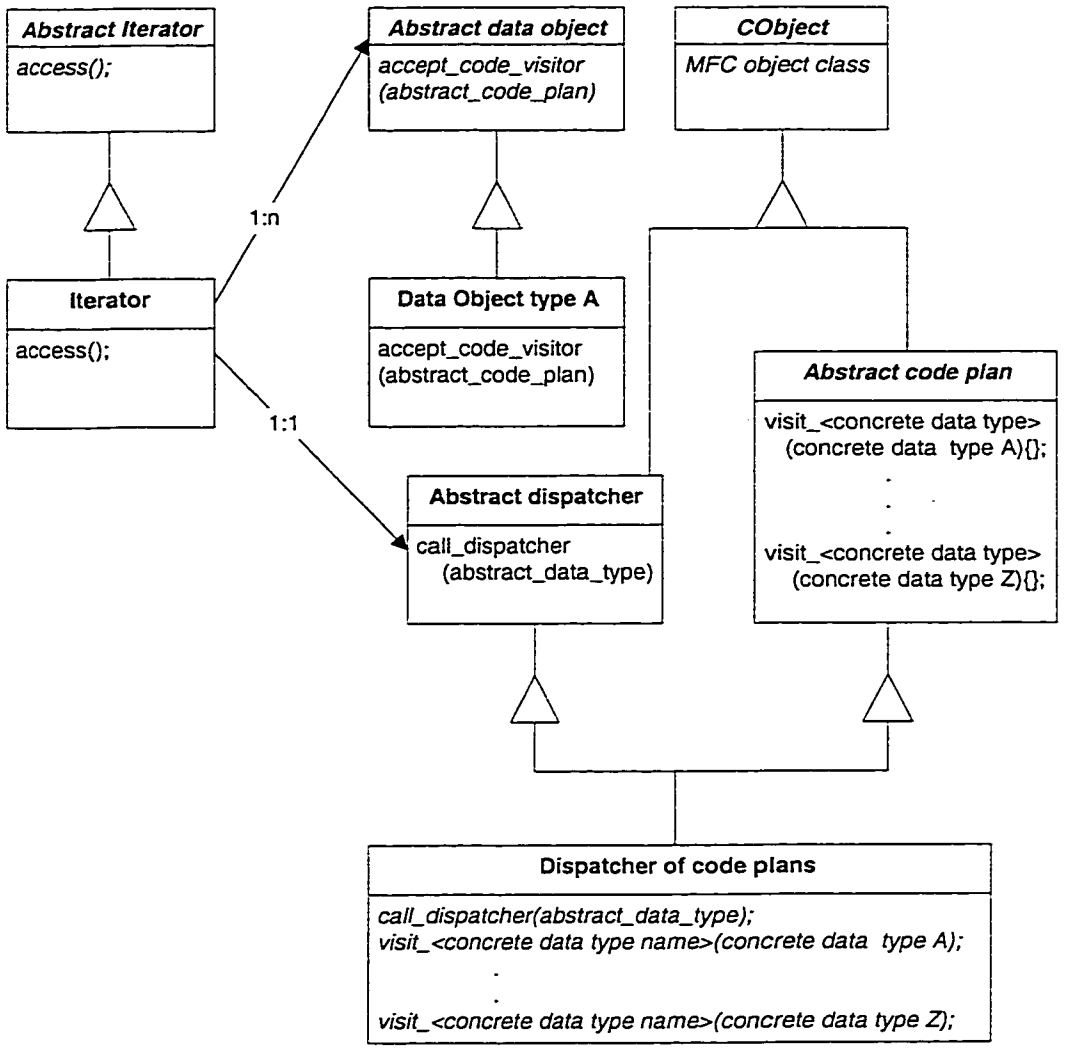

Figure B.1: IDOW class hierarchy

Figure B.1 shows the class hierarchy with multiple inheritance being used to construct the IDOW pattern. From the diagram, **iterator** imports the data structure to-be-traversed and the abstract dispatcher which uses polymorphism allowing the iterator to receive an open-ended number of code plans. IDOW's multiple inheritance bridges the dispatcher with the abstract code plan. Variations on this pattern occur when the **iterator** needs specific knowledge within the code plan. Due to the specifics, the **iterator** and the dispatcher lose their abstractness to become close-ended.

B.7 Participants

- Abstract Data Type
  - declares an accept_workers method taking an abstract_workers as an argu-

ment.

- Concrete Data Type
  - implements an accept_workers method of its base class Abstract Data type. The passed parameter is the concrete pool of workers to select from.
- Abstract iterator
  - declares an access method.
- Concrete Iterator
  - implements the access method that traverses the concrete data objects. The "starting point within the data objects to-be-traversed" and the "dispatch of workers" are the arguments to the iterator's constructor. The access method initiates the IDOW pattern execution. Inside the access method is the call to the dispatch-of-workers's call_dispatcher method.
- Abstract Dispatch
  - defines the call_dispatcher method taking the abstract accessed element of the iterator as an argument.
- Abstract pool-of-Workers
  - defines and implements the specific default "worker code" procedures for each concrete data object. Normally the default action is do-nothing.
- Concrete Dispatch of Workers — Combined Dispatch and Pool of Workers
  - implements, using multiple inheritance, the call_dispatcher method and the individual worker method for each concrete data type. If a specific data element does nothing, that appropriate worker method is omitted defaulting to the base class implementation.

B.8 C++ Implementation

This pattern is made from four abstract ingredients. The real classes are derived from these base classes. The base ingredients are as follows:

- the abstract data-object class: Define the abstract data object class having an abstract accept_code_worker method. This method receives, via a parameter, the abstract dispatch-of-workers class defined below. Real data classes are then derived from this class.

```
// 1) An example of an Abstract Data-object Class
class Cabstract_state: public CObject{
  public:
    virtual ~CAbstract_state(){};
    virtual void accept_workers
       (CAbstract_state_dispatcher_workers* Workers)=0;
  protected:
    CAbstract_state(){};
};
```

- the abstract iterator: Define an abstract iterator class with its abstract access method; this access method provides a constant name to all the iterators initiating the IDOW pattern — it is not mandatory but forces a consistency in use. Real iterators are derived from this base class and implement the access method that traverses the data objects.

```
//  2) An example of an Abstract Iterator
class Cabstract_iterator: public CObject{
  public:
    virtual ~CAbstract_iterator(){};
    virtual void access()=0;
  protected:
    CAbstract_iterator(){};
};
```

- the abstract pool-of-workers class: This base class contains the code-procedures for each concrete data object defined with its parameter being that of the real data object. The local procedure name is composed of a prefix "worker_" and the real data type name; suppose the concrete data type is client record, the local procedure name would be named "worker_client_record". As a default action, these worker-code procedures are implemented with no action; this frees up the programmer to code only the subset of workers needed.

  For objects that are general in activity and do not have a specific worker, a general contracting routine is available in the pool-of-workers.

```
//  3) An example of an Abstract pool of workers
//     shows 3 real data objects in use
class Cabstract_state_workers:
  virtual public CObject{
                // virtual due to multiple inheritance
                //            on CObject
```

```
                // when concretely combined with the
                // abstract dispatcher class
  public:
    Cabstract_state_workers(){};
    virtual ~Cabstract_state_workers(){};
    virtual void worker_state(Cstate* Object_state){};
    virtual void worker_state_sub_rule_vector_list
            (Cstate_sub_rule_vector_list* Object_state){};
    virtual void worker_state_sub_rule_vector
            (Cstate_sub_rule_vector* Object_state){};
};
```

- the abstract dispatch-of-workers class: For each abstract data type, an abstract dispatch class is defined.

```
//  4) An example of an Abstract Dispatch
class CAbs_dispatch_of_state:
  virtual public CObject{
                // virtual due to multiple inheritance
                //            on CObject
                // when concretely combined with the
                // abstract code-workers class
  public:
    virtual ~CAbs_dispatch_of_state(){};
    virtual
    void call_dispatcher(CAbstract_state* Object_state)=0;
  protected:
    CAbs_dispatch_of_state(){};
};
```

Now the concrete classes are derived and assembled from these base classes creating the IDOW pattern. The only tricky part is using multiple inheritance on the "abstract pool of workers" class and the "abstract dispatch" class producing a combined Dispatch-Workers real class. This is the DW part of the pattern name. The call_dispatcher method and all the local worker code methods are implemented. Normally, one iterator (I) is defined driving many DW classes. The DW part is the local code context that needs executing. From my experience in writing the lrl compiler/compiler program, nearly one hundred DW definitions were created.

```
//  An example of Concrete Class Definitions
//  which builds on the abstract definitions defined above
//
//  1) Real data objects derived from Cabstract_state
```

```
//
class Cstate_sub_rule_vector:public CAbstract_state{public:
  Cstate_sub_rule_vector();
  Cstate_sub_rule_vector
    (Celement*                          sbr_elem
    ,Cstate*                            closure_state
    ,CMapPtrToPtr*                      la_list = NULL);
  ~Cstate_sub_rule_vector();
            ....
  void accept_workers(CAbstract_state_workers* Workers);
  private:
  Celement*                       sbr_elem_;
            ....
  CMapPtrToPtr*                   birthing_list_;
};

class Cstate_sub_rule_vector_list: public CAbstract_state{
  public:
    Cstate_sub_rule_vector_list(Crule* name);
    ~Cstate_sub_rule_vector_list();
    Cstate_sub_rule_vector& operator []
                               (Cstate_sub_rule_vector* a);
    int                       no_of_vectors();
    Cstate_sub_rule_vector* get_spec_vector(int id);
    void                      closure_rule(Crule* a);
    Crule*                    closure_rule();
    void accept_workers(CAbstract_state_workers* Workers);
    friend ostream& operator <<
          (ostream& os,Cstate_sub_rule_vector_list& e);
  private:
    Crule*              closure_rule_;
    CObArray            state_sub_rule_vector_list_;
};

class Cstate:public CAbstract_state{public:
    Cstate(Cgrammar_vocabulary* vector,Cgrammar* Grammar);
    ~Cstate();
            ....
    void accept_workers(CAbstract_state_workers* Workers);
    friend ostream& operator <<(ostream& os,Cstate& e);
  private:
    int             state_no_;
    unsigned long   state_signature_;
            ....
    CMapPtrToPtr*   shift_la_set_;
    CObList*        eosr_list_;
};
```

```
//
//  The real object implementations are omitted except
//  their accept_workers routine. They do not add any value
//  in demonstrating the IDOW pattern. The accept_workers
//  shows which specific worker engaged by the object.
//
void Cstate_sub_rule_vector::
accept_workers(CAbstract_state_workers* Worker){
  Worker->worker_state_sub_rule_vector(this);
}

void Cstate_sub_rule_vector_list::
accept_workers(CAbstract_state_workers* Worker){
  Worker->worker_state_sub_rule_vector_list(this);
}

void Cstate::
accept_workers(CAbstract_state_workers* Worker){
  Worker->worker_state(this);
}

// 2) Concrete Iterator
class Cstate_iterator:public CAbstract_iterator{public:
    enum state_part {closure_part,transitive_part};
    Cstate_iterator
       (state_part                State_part
       ,Cstate*                   Root
       ,CAbs_dispatcher_of_state* Plan);
    virtual ~Cstate_iterator();
    void access();
  private:
  state_part access_what_;Cstate* root_;CAbs_dispatcher_of_state* plan_;
};

//
// state Iterator Implementation
//
Cstate_iterator::Cstate_iterator
  (Cstate_iterator::state_part       State_part
  ,Cstate*                           Root
  ,CAbs_dispatcher_of_state_workers* Plan)
  :access_what_(State_part),root_(Root),plan_(Plan){};

Cstate_iterator::~Cstate_iterator(){};

void Cstate_iterator::access(){
  plan_->call_dispatcher(root_); // state data object
  int i,y;
```

```
  Cstate_sub_rule_vector_list* ssrvl;
  Cstate_sub_rule_vector* ssrv;
  // always check dynamically against no of rules
  // because they can be iterated while being built
  // hence the result per test is not CONSTANT!!
  for (i=1;;i++){
    if (access_what_ == closure_part){
      if ( i > root_->no_of_closure_rules()) break;
    }else{ if ( i > root_->no_of_transitive_rules()) break;}
    if (access_what_ == closure_part)
      ssrvl = root_->get_spec_closure_rule_list(i);
    else ssrvl = root_->get_spec_transitive_rule_list(i);
    plan_->call_dispatcher(ssrvl);// sub rule vector list
    for (y=1,ssrv = ssrvl->get_spec_vector(y);
      ssrv!=NULL;y++,ssrv = ssrvl->get_spec_vector(y)){
      plan_->call_dispatcher(ssrv);// sub rule vector
    }
  }
  return;
}

//  3) Dispatch-Workers (DW) Definition
//  This example only defines 2 of the three worker-code
//  procedures. It shows the flexibility of the IDOW pattern
//  where not all the code-workers have to be defined.
//  The default code for the worker_state_sub_rule_vector_list
//  is the base class implementation which does nothing.
class Cstate_dump_dw:
      public Cabs_dispatch_of_state,Cabstract_state_workers{public:
    Cstate_dump_dv(ostream&  DC);
    ~Cstate_dump_dv();
    void call_dispatcher(CAbstract_state* Object_state);
    void worker_state(Cstate* Object_state);
    void worker_state_sub_rule_vector
            (Cstate_sub_rule_vector* Object_sub_rule_vector);
  private: ostream& dc_;
}

//
// implementation
//
Cstate_dump_dw::Cstate_dump_dw(ostream& DC):dc_(DC){};

void Cstate_dump_dw::call_dispatcher(CAbstract_state* Object_state){
  Object_state->accept_code_worker(this);
}

void Cstate_dump_dw::worker_state(Cstate* Object_state){
```

```
  dc_ << "--->state no: "
      << Object_state->state_no() << " state label:: " << endl;
     .
     . // balance of details irrelevant
     .
}

void Cstate_dump_dw::worker_state_sub_rule_vector
    (Cstate_sub_rule_vector* Object_sub_rule_vector){
  Celement* e = Object_sub_rule_vector->sbr_elem();
     . // dump this record's data
     . // balance of details irrelevant
     .
}

//
// 4) Example of IDOW pattern being used to dump a state
//
Cstate_dump_dw state_dump(clog);  // DW part
Cstate_iterator state_iterator
                  (Cstate_iterator::closure_part
                  ,A_state_ptr    // structure to iterate
                  ,&state_dump);
state_iterator.access();          //  execution of the IDOW Pattern
```

B.9 Consequences

The major benefit is code-isolation within the pool-of-workers's class. It is the co-operation between the Iterator, Dispatcher, and Workers that makes the pattern easy to use.

The downside to a normal IDOW pattern is all elements are visited by the iterator and their worker-code procedures are executed; no discrimination is done on the data element types being traversed. It is this abstractness on the elements traversed that make this pattern so attractive in use — you just go ahead and execute the iterator: it handles all the other details. Unless the code is time critical, the minor overhead in executing default worker-procedures — usually inlined — is more than off-set by the code simplicity. By tailoring the iterator and DW parts, variations three and four described above, allow the programmer to over come these overheads.

Appendix C

Lr1 Grammar Test Suite

In this section, I give some grammars used to test the concepts developed in the thesis.

C.1 Common Terminals — lr1_test_terminals.h

This is the grammar's terminals header file used by the test suite grammars. It defines each terminal with some terminals having stack action routines. All the test suite grammars defined later include this file into their definition header. Defined later, the grammar demonstrating the parse tracing capability shows its inclusion by "@lr1_test_terminals.h" (without the quotes).

```
//
//  file:    lr1_test_terminals.h
// purpose: common terminals for test suite grammars:
//
fsm-operation
(fsm-operation-shift-class  fsm_Shift
,fsm-operation-reduce-class fsm_Reduce
,fsm-operation-accept-class fsm_Accept
,fsm-operation-error-class  fsm_Error);
terminals
{
  a (sym-class Csymbol_a); b (sym-class Csymbol_b);
  c (sym-class          Csymbol_c
    ,sym-look-ahead-rtn Csymbol_c_look_ahead_rtn
    ,stack-pushed-rtn   Csymbol_c_pushed_rtn
    ,stack-popped-rtn   Csymbol_c_popped_rtn);
  d (sym-class Csymbol_d); e (sym-class Csymbol_e);
  f (sym-class Csymbol_f); g (sym-class Csymbol_g);
  q (sym-class Csymbol_q); t (sym-class Csymbol_t);
```

```
  u (sym-class Csymbol_u); v (sym-class Csymbol_v);
  w (sym-class Csymbol_w); x (sym-class Csymbol_x);
  y (sym-class Csymbol_y); z (sym-class Csymbol_z);
  eog(sym-class         Csymbol_eog
    ,sym-look-ahead-rtn Csymbol_eog_look_ahead_rtn
    ,stack-pushed-rtn   Csymbol_eog_pushed_rtn
    ,stack-popped-rtn   Csymbol_eog_popped_rtn);
  eolr(sym-class        Sym_eolr);
  "|||" (sym-class      Sym_parallel_operator);
  "|||r"(sym-class      Sym_parallel_reduce_operator);
  "|||b"(sym-class      Sym_parallel_bndry_operator);
  "|||0"(sym-class bs$$_0); "|||1"(sym-class bs$$_1);
  "|||2"(sym-class bs$$_2); "|||3"(sym-class bs$$_3);
  "|||4"(sym-class bs$$_4); "|||5"(sym-class bs$$_5);
};
```

The "eog" terminal is a special terminal for "end of grammar reached" condition. It is automatically generated by the "end of file" condition when the raw characters are being read. For continuity or the compiler writer's propensity, multi pass grammars can append this special terminal to their containers — two "eog"es are appended if the start rule's right-hand-side contains the "eog" terminal; this allows the start rule to be reduced along with the execution of its action services. The terminals following "eog" are used internally by the compiler/compiler.

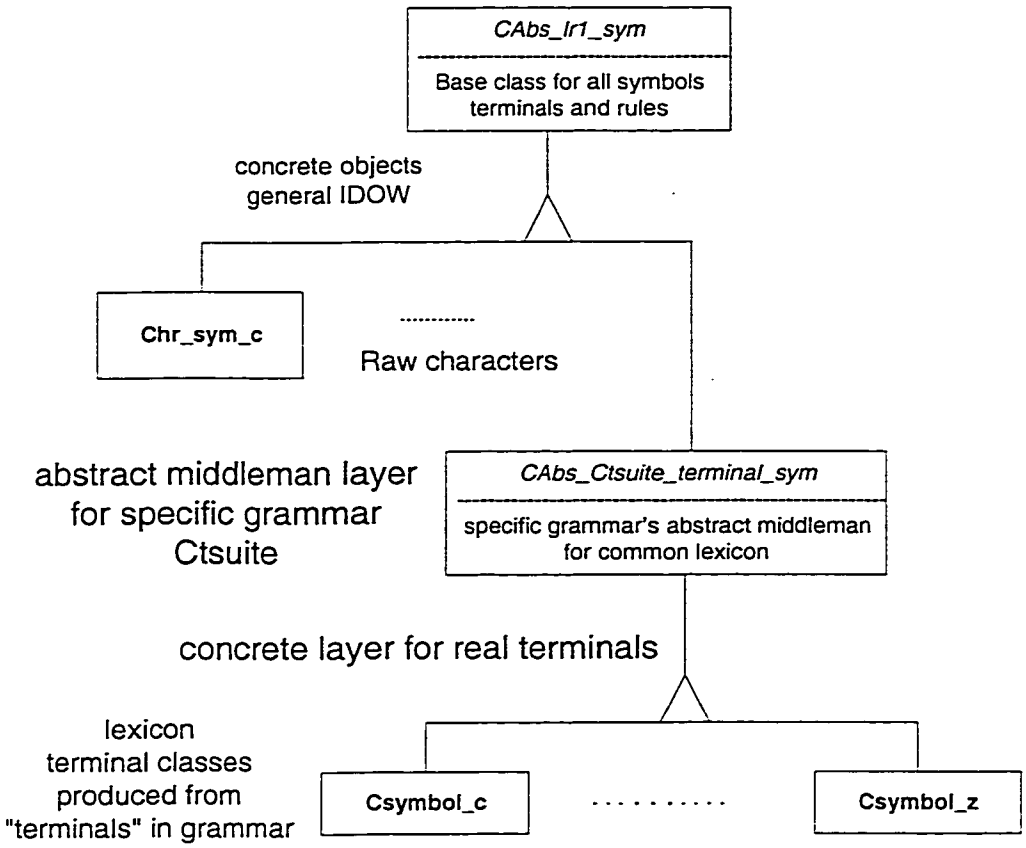

Figure C.1: Terminal class hierarchy for Ctsuite

Figure C.1 graphically describes the class hierarchies emitted for the terminals. From the diagram, the general base class *CAbs_lr1_sym* derives the specific middleman terminal class *CAbs_Ctsuite_terminal_sym*, a manufactured name from the compiler/compiler which derives the real terminal symbols of **Ctsuite** grammar. In a multi-pass environment, these class definitions are shared.

C.2 Test Suite's Lexical Grammar

The following grammar produces tokens for the test suite. One of its emitted files — Ctsuite.h — is the C++ terminal classes' header file which is included in every test suite grammar file using the "@" file include operator; this included file gets passed through to the C++ emitted files of each grammar.

```
/*  FILE:  lr1_test_terminals.lex
    Common lexer for suite of lr1 test grammars */
fsm (fsm-id       "lr1_test_terminals.lex"
,fsm-class    Ctsuite
,fsm-version  "1.0",fsm-date "8-nov-97",fsm-debug "yes"
,fsm-comments "lexer for lr1 test suite");
@lr1_test_terminals.h
rules{
  $ts_lexer
    (sym-class Sym_$ts_lexer
    ,lhs{
        op
          Parser_env->token_producer()->add_token(new Csymbol_eog);
          Parser_env->token_producer()->add_token(new Csymbol_eog);
        ***
     };
    ){-> $ts_toks eog };
  $ts_toks(sym-class Sym_$ts_toks){
      -> $ts_tok
      -> $ts_toks $ts_tok
    };
  $ts_tok
    (sym-class    Sym_$ts_tok
    ,lhs
      { user-declaration
          public: CAbs_lr1_sym* tok();private: CAbs_lr1_sym* tok_;***
        user-implementation
          CAbs_lr1_sym* $ts_tok::tok(){return tok_;};***
        op Parser_env->token_producer()->add_token(tok());***
      };
    ){-> a{lhs-constructor tok_ = new Csymbol_a;***};
      -> b{lhs-constructor tok_ = new Csymbol_b;***};
      -> c{lhs-constructor tok_ = new Csymbol_c;***};
      -> d{lhs-constructor tok_ = new Csymbol_d;***};
      -> e{lhs-constructor tok_ = new Csymbol_e;***};
      -> f{lhs-constructor tok_ = new Csymbol_f;***};
      -> g{lhs-constructor tok_ = new Csymbol_g;***};
      -> q{lhs-constructor tok_ = new Csymbol_q;***};
      -> t{lhs-constructor tok_ = new Csymbol_t;***};
      -> u{lhs-constructor tok_ = new Csymbol_u;***};
      -> v{lhs-constructor tok_ = new Csymbol_v;***};
      -> w{lhs-constructor tok_ = new Csymbol_w;***};
      -> x{lhs-constructor tok_ = new Csymbol_x;***};
      -> y{lhs-constructor tok_ = new Csymbol_y;***};
      -> z{lhs-constructor tok_ = new Csymbol_z;***};
    };
}; // end of grammar
```

C.3 Common C++ Terminals — Ctsuite.h

This is the C++ header file used by the grammars of the test suite. It is generated from the lexical grammar and used by all the other grammars: there is an option in the compiler/compiler whether to generate the terminals C++ code. The three include files are automatically added to the emitted files by the compiler/compiler to use: Microsoft's "MFC" class environment, the "IDOW" pattern, raw character processing for the lexical grammar, and parallel grammar processing which is being experimented with. To conserve space, the emitted file has been edited.

```
#include "stdafx.h"
#include "emit_idow.h"
#include "parallel_idow.h"
#define new DEBUG_NEW
class CAbs_Ctsuite_terminal_objects_workers;
// Definition: terminals middleman
class CAbs_Ctsuite_terminal_sym:public CAbs_lr1_sym{public:
    virtual               ~CAbs_Ctsuite_terminal_sym();
    virtual void
    accept_workers(CAbs_Ctsuite_terminal_objects_workers* Workers)=0;
  protected:
    CAbs_Ctsuite_terminal_sym
        (CAbs_s_r_rtn*       Look_ahead_rtn
        ,CAbs_s_r_rtn*       Shift_rtn
        ,CAbs_s_r_rtn*       Reduce_rtn
        ,const char*         Id);
};
// Definition: terminals idow middleman
class CAbs_Ctsuite_terminal_dw:virtual public CObject{public:
    virtual             ~CAbs_Ctsuite_terminal_dw(){};
    virtual void
    call_dispatcher(CAbs_Ctsuite_terminal_sym* Object_terminal)=0;
  protected:
    CAbs_Ctsuite_terminal_dw(){};
};
class CAbs_Ctsuite_rule_objects_workers;
// Definition: rules middleman
class CAbs_Ctsuite_rule_sym:public CAbs_rule_sym{public:
    virtual               ~CAbs_Ctsuite_rule_sym();
    virtual void
    accept_workers(CAbs_Ctsuite_rule_objects_workers* Workers)=0;
  protected:
    CAbs_Ctsuite_rule_sym       // rule
        (CAbs_lr1_sym*       Rule_sym_class
        ,CAbs_rule_sym*      Sub_rule
        ,const char*         Id);
    CAbs_Ctsuite_rule_sym       // subrule
```

```
        (int                Rhs_no_of_parms
        ,int                Sub_rule_id
        ,const char*        Id);
};
// Definition: idow middleman for rules
class CAbs_Ctsuite_rule_dw:virtual public CObject{public:
    virtual             ~CAbs_Ctsuite_rule_dw(){};
    virtual void call_dispatcher(CAbs_Ctsuite_rule_sym* Object_rule)=0;
  protected:CAbs_Ctsuite_rule_dw(){};
};
// Terminals classes with their action services classes
class Csymbol_w:public CAbs_Ctsuite_terminal_sym{public:
    ~Csymbol_w();
    void accept_workers(CAbs_Ctsuite_terminal_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);Csymbol_w(const char*   Id = "w");
};
class Csymbol_eog_look_ahead_rtn:public CAbs_s_r_rtn{public:
    Csymbol_eog_look_ahead_rtn(const char* Id = "Csymbol_eog_look_ahead_rtn");
    ~Csymbol_eog_look_ahead_rtn();
    void op(CAbs_lr1_sym* Terminal,CAbs_parser* Parser_env);
};
class Csymbol_eog_pushed_rtn:public CAbs_s_r_rtn{public:
    Csymbol_eog_pushed_rtn(const char* Id = "Csymbol_eog_pushed_rtn");
    ~Csymbol_eog_pushed_rtn();
    void op(CAbs_lr1_sym* Terminal,CAbs_parser* Parser_env);
};
class Csymbol_eog_popped_rtn:public CAbs_s_r_rtn{public:
    Csymbol_eog_popped_rtn(const char* Id = "Csymbol_eog_popped_rtn");
    ~Csymbol_eog_popped_rtn();
    void op(CAbs_lr1_sym* Terminal,CAbs_parser* Parser_env);
};
class Csymbol_eog:public CAbs_Ctsuite_terminal_sym{public:
    ~Csymbol_eog();
    void accept_workers(CAbs_Ctsuite_terminal_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);Csymbol_eog(const char* Id = "eog");
    static class Csymbol_eog_look_ahead_rtn look_ahead_rtn;
    static class Csymbol_eog_pushed_rtn shift_rtn;
    static class Csymbol_eog_popped_rtn reduce_rtn;
};
// The following terminals have been edited out for space reasons
// They follow the same emit pattern as exampled terminals above
// f x g y z |||  |||r |||b |||0 |||1 |||2 |||3 |||4 |||5 t c u d eolr v e
//

// Ctsuite FSM class definition
class Ctsuite: public CAbs_fsm{public:
enum sub_rules{start_of_sub_rule_list = 0
        ,rhs1_$ts_lexer
```

```
          ,rhs1_$ts_toks,rhs2_$ts_toks
          ,rhs1_$ts_tok,rhs2_$ts_tok,rhs3_$ts_tok,rhs4_$ts_tok
          ,rhs5_$ts_tok,rhs6_$ts_tok,rhs7_$ts_tok,rhs8_$ts_tok
          ,rhs9_$ts_tok,rhs10_$ts_tok,rhs11_$ts_tok,rhs12_$ts_tok
          ,rhs13_$ts_tok,rhs14_$ts_tok,rhs15_$ts_tok};
enum rules{start_of_rule_list = 0,rule_$ts_lexer
               ,rule_$ts_toks,rule_$ts_tok};
    enum states{reduce = 0,R = reduce,no_of_states = 20};
    Ctsuite
      (CMapStringToOb** Gbl_sym_tbl=NULL
      ,const char*      Debug="yes"
      ,const char*      Comments="common lexer for lr1 test suite grammars"
      ,const char*      Id="lr1_test_terminals.lex"
      ,const char*      Version="1.0"
      ,const char*      Date="8-nov-97");
    ~Ctsuite();
    void              op(CAbs_parser* Parser_env);
    CAbs_fsm*         entry();
    CAbs_fsm*         exit();
    CAbs_rule_sym*    reduce_rhs_of_rule
                          (int          T  // top of stack
                          ,CAbs_parser* Parser_env
                          ,int          Sub_rule_no);
    CAbs_fsm_oper*    s_op();//shift operation
    CAbs_fsm_oper*    r_op();//reduce operation
    CAbs_fsm_oper*    a_op();//accept operation
    CAbs_fsm_oper*    e_op();//error operation
  private:
    void load_symbols_into_fsm_symbol_table();
    void load_symbols_into_parallel_bndry_fsm_symbol_table();
    void load_states_into_fsm_state_table();
  private:
  CAbs_fsm_oper* s_;// shift
  CAbs_fsm_oper* r_;// reduce
  CAbs_fsm_oper* a_;// accept
  CAbs_fsm_oper* e_;// error
};

class $ts_lexer;
class $ts_toks;
class $ts_tok;
// Rules Classes with action services classes
class Sym_$ts_lexer:public CAbs_lr1_sym{public:
    Sym_$ts_lexer(const char* Id = "$ts_lexer");
    ~Sym_$ts_lexer();
    void accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);
};
```

```
class Rhs1_$ts_lexer:public CAbs_Ctsuite_rule_sym{public:
    Rhs1_$ts_lexer(int P1,int P2,CAbs_parser* Parser_env
                  ,int Sub_rule_id = Ctsuite::rhs1_$ts_lexer
                  ,const char* Id = "Rhs1_$ts_lexer");
    ~Rhs1_$ts_lexer();
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);
    $ts_toks* get_p1();Csymbol_eog* get_p2();
  private:friend $ts_lexer;
  $ts_toks* p1_;Csymbol_eog* p2_;
  CAbs_parser* parser_env_;
};
class $ts_lexer:public CAbs_Ctsuite_rule_sym{public:
    ~$ts_lexer();
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);
    $ts_lexer(Sym_$ts_lexer* Rule_sym_class,Rhs1_$ts_lexer* Sub_rule_1
             ,CAbs_parser* Parser_env,const char* Id = "$ts_lexer");
  private:CAbs_parser* parser_env_;
};
class Sym_$ts_toks:public CAbs_lr1_sym{public:
    Sym_$ts_toks(const char*   Id = "$ts_toks");
    ~Sym_$ts_toks();
    void accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);
};
class Rhs1_$ts_toks:public CAbs_Ctsuite_rule_sym{public:
    Rhs1_$ts_toks(int P1,CAbs_parser* Parser_env
                  ,int Sub_rule_id = Ctsuite::rhs1_$ts_toks
                  ,const char* Id = "Rhs1_$ts_toks");
    ~Rhs1_$ts_toks();
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);$ts_tok* get_p1();
  private:friend $ts_toks;
  $ts_tok* p1_;CAbs_parser* parser_env_;
};
class Rhs2_$ts_toks:public CAbs_Ctsuite_rule_sym{public:
    Rhs2_$ts_toks(int P1,int P2,CAbs_parser* Parser_env
                  ,int Sub_rule_id = Ctsuite::rhs2_$ts_toks
                  ,const char* Id = "Rhs2_$ts_toks");
    ~Rhs2_$ts_toks();
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);$ts_toks* get_p1();$ts_tok* get_p2();
  private:friend $ts_toks;
  $ts_toks* p1_;$ts_tok* p2_;CAbs_parser* parser_env_;
};
class $ts_toks:public CAbs_Ctsuite_rule_sym{public:
    ~$ts_toks();
```

```
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);
    $ts_toks(Sym_$ts_toks* Rule_sym_class,Rhs1_$ts_toks* Sub_rule_1
            ,CAbs_parser* Parser_env,const char* Id = "$ts_toks");
    $ts_toks(Sym_$ts_toks* Rule_sym_class,Rhs2_$ts_toks* Sub_rule_2
            ,CAbs_parser* Parser_env,const char* Id = "$ts_toks");
  private:CAbs_parser* parser_env_;
};

class Sym_$ts_tok:public CAbs_lr1_sym{public:
    Sym_$ts_tok(const char*   Id = "$ts_tok");
    ~Sym_$ts_tok();
    void accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void op(CAbs_parser* Parser_env);
};
class Rhs1_$ts_tok:public CAbs_Ctsuite_rule_sym{public:
    Rhs1_$ts_tok(int P1,CAbs_parser* Parser_env
                ,int Sub_rule_id = Ctsuite::rhs1_$ts_tok
                ,const char* Id = "Rhs1_$ts_tok");
    ~Rhs1_$ts_tok();
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);Csymbol_a* get_p1();
  private:friend $ts_tok;Csymbol_a* p1_;CAbs_parser* parser_env_;
};
class Rhs2_$ts_tok:public CAbs_Ctsuite_rule_sym{public:
    Rhs2_$ts_tok(int P1,CAbs_parser* Parser_env
                ,int Sub_rule_id = Ctsuite::rhs2_$ts_tok
                ,const char* Id = "Rhs2_$ts_tok");
    ~Rhs2_$ts_tok();
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);Csymbol_b* get_p1();
  private:friend $ts_tok;Csymbol_b* p1_;CAbs_parser* parser_env_;
};
//
// The balance of subrule classes edited out for space reasons
// along with the corresponding rule's constructors.
// They follow the same emit pattern.
// Rhs3_$ts_tok to Rhs15_$ts_tok
//
class $ts_tok:public CAbs_Ctsuite_rule_sym{public:
    ~$ts_tok();
    void   accept_workers(CAbs_Ctsuite_rule_objects_workers* Worker);
    void   op(CAbs_parser* Parser_env);
    $ts_tok(Sym_$ts_tok* Rule_sym_class,Rhs1_$ts_tok* Sub_rule_1
           ,CAbs_parser* Parser_env,const char* Id = "$ts_tok");
    $ts_tok(Sym_$ts_tok* Rule_sym_class,Rhs2_$ts_tok* Sub_rule_2
           ,CAbs_parser* Parser_env,const char* Id = "$ts_tok");
    ..... constructors for Rhs3_$ts_tok - Rhs15_$ts_tok omitted
```

```
  private:
  CAbs_parser* parser_env_;
          public:  CAbs_lr1_sym* tok();
          private: CAbs_lr1_sym* tok_;
};

// rules idow worker middleman class
class CAbs_Ctsuite_rule_objects_workers:public CAbs_lr1_sym_workers{public:
    CAbs_Ctsuite_rule_objects_workers();
    ~CAbs_Ctsuite_rule_objects_workers();
 // visitor functions
 virtual void worker_$ts_lexer($ts_lexer* Object);
 virtual void worker_Sym_$ts_lexer(Sym_$ts_lexer* Object);
 virtual void worker_Rhs1_$ts_lexer(Rhs1_$ts_lexer* Object);
 virtual void worker_$ts_toks($ts_toks* Object);
 virtual void worker_Sym_$ts_toks(Sym_$ts_toks* Object);
 virtual void worker_Rhs1_$ts_toks(Rhs1_$ts_toks* Object);
 virtual void worker_Rhs2_$ts_toks(Rhs2_$ts_toks* Object);
 virtual void worker_$ts_tok($ts_tok* Object);
 virtual void worker_Sym_$ts_tok(Sym_$ts_tok* Object);
 virtual void worker_Rhs1_$ts_tok(Rhs1_$ts_tok* Object);
 virtual void worker_Rhs2_$ts_tok(Rhs2_$ts_tok* Object);
  ..... worker_Rhs3_$ts_tok to worker_Rhs15_$ts_tok omitted
};

// terminals idow worker middleman class
class CAbs_Ctsuite_terminal_objects_workers:public CAbs_lr1_sym_workers{
 public:
    CAbs_Ctsuite_terminal_objects_workers();
    ~CAbs_Ctsuite_terminal_objects_workers();
 // visitor functions
 virtual void worker_Csymbol_w(Csymbol_w* Obj);
 virtual void worker_Csymbol_f(Csymbol_f* Obj);
 virtual void worker_Csymbol_x(Csymbol_x* Obj);
 virtual void worker_Csymbol_g(Csymbol_g* Obj);
 virtual void worker_Csymbol_y(Csymbol_y* Obj);
 virtual void worker_Csymbol_z(Csymbol_z* Obj);
 virtual void worker_Sym_parallel_operator(Sym_parallel_operator* Obj);
 virtual void worker_Sym_parallel_reduce_operator
                 (Sym_parallel_reduce_operator* Obj);
 virtual void worker_Sym_parallel_bndry_operator
                 (Sym_parallel_bndry_operator* Obj);
  ..... terminals a,b,c,d,e,q,t,u,v,eog,eolr,|||0 - |||5 omitted
 };
```

C.4 Test Suite of Grammars

To test the LR1 capability, a collection of grammars supplied by various authors between the years of 1977 to 1988 is used. Each grammar is presented in a stylized way with the class names omitted to conserve page space. The compiler/compiler's output is also compressed — it graphically shows the state, its rules and the parse position per subrule, the goto vector, and the lookahead set if the subrule is consumed. An asterisk beside the state denotes the state was not merged due to "lrness" conflict. For their "lrness" discussions, the reader should read the original papers referenced. Also shown for one of the grammars is its tracings by the parser.

C.4.1 Deremer and Pennello Grammars

C.4.1.1 LALR(1) Grammar: Page 632 of [3]

The grammar tests out LALR resolution on states 14 and 15.

Rules			
$S	*$S1*	*$A*	*$B*
→ $S1 eog	→ a $A c	→ $B	→ g
	→ a g d		
	→ b $A d		
	→ b g c		

1	$S→.$S1 eog	12	
	$S1→.a $A c	2	
	$S1→.a g d	2	
	$S1→.b $A d	7	
	$S1→.b g c	7	
2	$S1→ a.$A c	3	
	$S1→ a.g d	5	
	$A→.$B	14	
	$B→.g	5	
3	$S1→ a $A.c	4	
4	$S1→ a $A c.		eog
5	$S1→ a g.d	6	
	$B→ g.		c
6	$S1→ a g d.		eog

7	$S1→ b.$A d	8	
	$S1→ b.g c	10	
	$A→.$B	15	
	$B→.g	10	
8	$S1→ b $A.d	9	
9	$S1→ b $A d.		eog
10	$S1→ b g.c	11	
	$B→ g.		d
11	$S1→ b g c.		eog
12	$S→ $S1.eog	13	
13	$S→ $S1 eog.		accept
14*	$A→ $B.		c
15*	$A→ $B.		d

Graphic output of lr parse table.

C.4.1.2 LR(0) Grammar: Page 633 of [3]

The grammar has no state conflicts.

Rules					
$S	*$S1*	*$B*	*$C*	*$D*	*$E*
→ $S1 eog	→ c d → a $B d → $C $B g	→ $D $E	→ c	→	→

1	$S→.$S1 eog	10	
	$S1→.c d	2	
	$S1→.a $B d	4	
	$S1→.$C $B g	7	
	$C→.c	2	
2	$C→ c.		g
	$S1→ c.d	3	
3	$S1→ c d.		eog
4	$S1→ a.$B d	5	
	$B→.$D $E	12	
	$D→.		d g
5	$S1→ a $B.d	6	

6	$S1→ a $B d.		eog
7	$S1→ $C.$B g	8	
	$B→.$D $E	12	
	$D→.		g
8	$S1→ $C $B.g	9	
9	$S1→ $C $B g.		eog
10	$S→ $S1.eog	11	
11	$S→ $S1 eog.		accept
12	$B→ $D.$E	13	
	$E→.		d g
13	$B→ $D $E.		d g

Graphic output of lr parse table.

C.4.2 Kristensen and Madsen Grammar

C.4.2.1 LALR(1) Grammar from [9]

The grammar tests LALR resolution on states 7 and 24.

Rules			
$S	*$E*	*$A*	*$B*
→ $E eog	→ $A a	→ x y z $A	→ x y z $B
	→ $B b	→ x q	→ x q
	→ u $A b		
	→ u $B a		

1	$S→.$E eog	17		12	$E→ u.$A b	13	
	$E→.$A a	8			$E→ u.$B a	15	
	$E→.$B b	10			$A→.x y z $A	19	
	$E→.u $A b	12			$A→.x q	19	
	$E→.u $B a	12			$B→.x y z $B	19	
	$A→.x y z $A	2			$B→.x q	19	
	$A→.x q	2					
	$B→.x y z $B	2		13	$E→ u $A.b	14	
	$B→.x q	2		14	$E→ u $A b.		eog
2	$A→ x.y z $A	3					
	$A→ x.q	7		15	$E→ u $B.a	16	
	$B→ x.y z $B	3					
	$B→ x.q	7		16	$E→ u $B a.		eog
3	$A→ x y.z $A	4		17	$S→ $E.eog	18	
	$B→ x y.z $B	4		18	$S→ $E eog.		accept
4	$A→ x y z.$A	5		19	$A→ x.y z $A	20	
	$B→ x y z.$B	6			$A→ x.q	24	
	$A→.x y z $A	2			$B→ x.y z $B	20	
	$A→.x q	2			$B→ x.q	24	
	$B→.x y z $B	2					
	$B→.x q	2		20	$A→ x y.z $A	21	
					$B→ x y.z $B	21	
5	$A→ x y z $A.		a	21	$A→ x y z.$A	22	
6	$B→ x y z $B.		b		$B→ x y z.$B	23	
					$A→.x y z $A	19	
7*	$A→ x q.		a		$A→.x q	19	
	$B→ x q.		b		$B→.x y z $B	19	
					$B→.x q	19	
8	$E→ $A.a	9		22	$A→ x y z $A.		b
9	$E→ $A a.		eog	23	$B→ x y z $B.		a
10	$E→ $B.b	11		24*	$A→ x q.		b
11	$E→ $B b.		eog		$B→ x q.		a

Graphic output of lr parse table.

C.4.3 Spector Grammars

C.4.3.1 LR(1) G3 Grammar with Example of a Parsing Trace: Page 64 of [13]

The grammar tests LR(1) resolution between states 4 and 15, and between states 6 and 17. The automatic tracing capabilities of the compiler/compiler is also illustrated.

Rules			
$MM	*$S*	*$A*	*$B*
→ $S eog	→ $A b	→ a $A c	→ a $B b
	→ $B c	→ a	→ a

1	$MM→.$S eog	11		7	$S→ $A.b	8	
	$S→.$A b	7					
	$S→.$B c	9		8	$S→ $A b.		eog
	$A→.a $A c	2					
	$A→.a	2		9	$S→ $B.c	10	
	$B→.a $B b	2					
	$B→.a	2		10	$S→ $B c.		eog
2	$A→ a.$A c	3		11	$MM→ $S.eog	12	
	$A→ a.		b				
	$B→ a.$B b	5		12	$MM→ $S eog.		accept
	$B→ a.		c				
	$A→.a $A c	13		13	$A→ a.$A c	14	
	$A→.a	13			$A→ a.		c
	$B→.a $B b	13			$B→ a.$B b	16	
	$B→.a	13			$B→ a.		b
					$A→.a $A c	13	
3	$A→ a $A.c	4			$A→.a	13	
					$B→.a $B b	13	
4*	$A→ a $A c.		b		$B→.a	13	
5	$B→ a $B.b	6		14	$A→ a $A.c	15	
6*	$B→ a $B b.		c	15*	$A→ a $A c.		c
				16	$B→ a $B.b	17	
				17*	$B→ a $B b.		b

Graphic output of lr parse table.

The grammar is presented with comments referenced in the annotated trace. The trace shows LR(1) resolution between states 4 and state 15. It also shows action service routines (syntax-directed) being exercised for some terminals and the start rule's syntax-directed code being executed. These services highlight the extra execution points that the compiler writer can inject code into the emitted program.

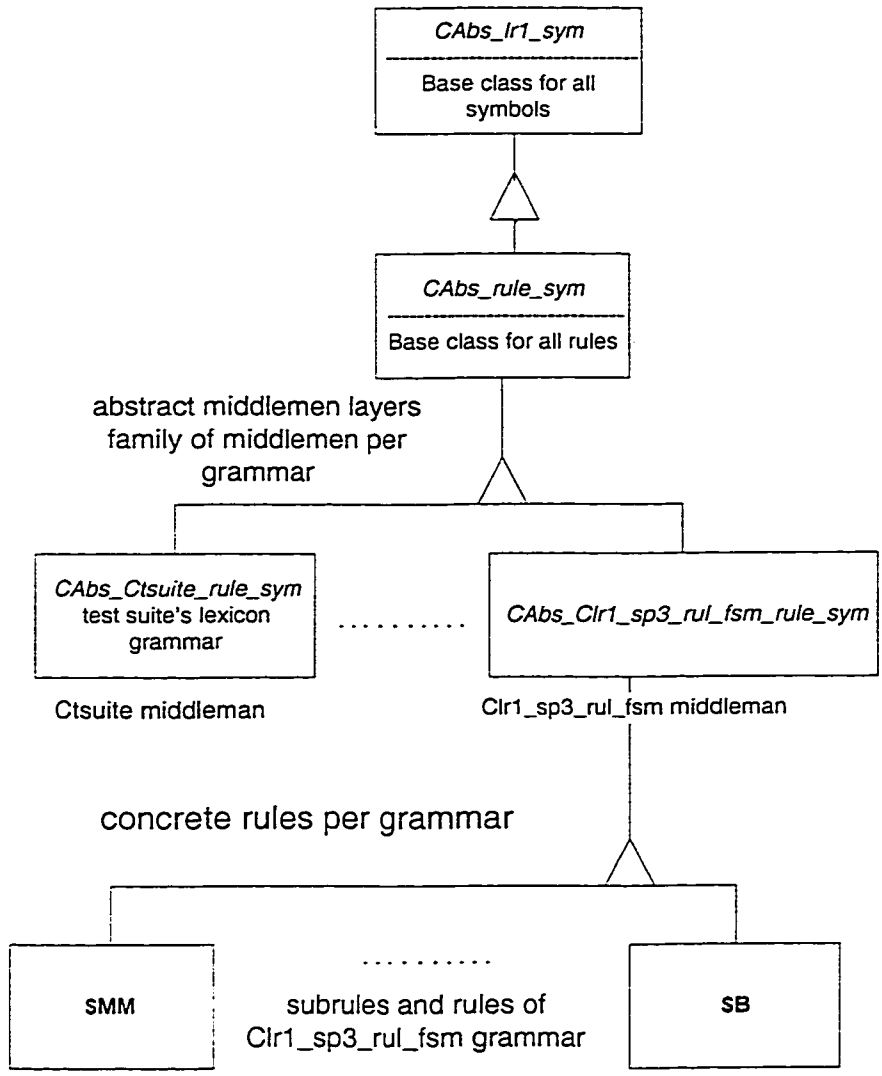

Figure C.2: Rule and Subrule class hierarchy for Clr1_sp3_rule_fsm using Ctsuite

Figure C.2 outlines the class hierarchies emitted for the rules and their subrules. It identifies the base classes, middlemen classes deriving from the base classes, and finally the symbol classes of **Clr1_sp3_fsm** grammar. The middlemen class names are manufactured as described in the thesis.

On the next page, the source grammar **Clr1_sp3_fsm** is given which this subsubsection's lead-in graphic depicts.

```
/*  FILE: LR1_SP3.RUL
    A LR1 GRAMMAR FROM DAVID SPECTOR
    SIGPLAN VOL 23 NO 12 DEC/88 */
fsm (fsm-id    "lr1_sp3.rul"
    ,fsm-class Clr1_sp3_fsm
    {
        user-prefix-declaration
#include "Ctsuite.h"
#define new DEBUG_NEW
        ***
      constructor clog << "Start of lr1_sp3\n";*** //ref::1
      destructor clog << "Stop of lr1_sp3\n";***   //ref::2
    };
   ,fsm-version "1.0",fsm-date     "8-oct-96"
   ,fsm-debug   "yes",fsm-comments "test out lr1");
@lr1_test_terminals.h
rules
{
  $MM(sym-class R$MM
     ,lhs
     ,stack-pushed-rtn C$MM_pushed_rtn            //ref::6
     ,stack-popped-rtn C$MM_popped_rtn){          //ref::7
      -> $S eog
    };
  $S(sym-class R$S){
      -> $A b
      -> $B c };
  $A(sym-class R$A
     ,lhs{destructor
           clog << "destructor -- $A" << endl;  //ref::3
          ***
         };
    ){
      -> a $A c
         {lhs-constructor                         //ref::4
            clog << "constructor $A -- subrule 1" << endl;
          ***
         };
      -> a
         {lhs-constructor                         //ref::5
             clog << "constructor $A -- subrule 2" << endl;
          ***
         };
    };
  $B(sym-class R$B){
      -> a $B b
      -> a };
}; // end of grammar
```

Shown below is the traced output run on Microsoft's NT Operating System. The "63" prefixing each output line is the thread's id being executed at the time of the trace. Stacked operations are prefaced by the number of "." indicating its position on the stack. The current parse stack configuration is symbolically displayed as linked arrows composed of the current state number, shifted symbol, and the goto state number. This is repeated for the number of items on the stack in a "bottom to top" stack order. Sprinkled within the trace are the parse objects — terminals, rules, subrules, stack action routines, and some simple C++ syntax-directed code — that express their tracing facilities. Optionally to the right of the trace lines are running comments. Some comments reference the labels within the grammar above; these reference tags are identified by the label "ref::#" with the "#" being a unique number.

"aaaccb" is the data to parse: "lr1ness" is tested between states 4 and 15.

```
Start of lr1_sp3                    // C++ syntax code ref::1
.63::shift_op token: 0x005248A0 a pos: 1 // parse operation
63::1--a-> 2                        // symbolic stack trace
                                    // state 1 with 'a' goes to state 2
..63::shift_op token: 0x00524E20 a pos: 2
63::1--a-> 2--a-> 13
...63::shift_op token: 0x00524B60 a pos: 3
63::1--a-> 2--a-> 13--a-> 13
....63::reduce_op
....63::Csymbol_c_look_ahead_rtn::op()// action service
                                    // defined in terminal
....63::Rhs2_$A::op()               // subrule op always executed
constructor $A -- subrule 2         // c++ injected code ref::5
....63::$A::op()                    // rule's op always executed
....63::popped state:: 13
...63::exposed sym:: a
63::1--a-> 2--a-> 13--$A-> 14       // result of reduce
....63::shift_op token: 0x00524720 c pos: 4
.....63::Csymbol_c_pushed_rtn::op()
63::1--a-> 2--a-> 13--$A-> 14--c-> 15
.....63::reduce_op                  // state 15 lr1ness resolution
.....63::Csymbol_c_look_ahead_rtn::op()
.....63::Rhs1_$A::op()
constructor $A -- subrule 1         // c++ injected code ref::4
.....63::$A::op()
.....63::popped state:: 15
....63::exposed sym:: c
....63::Csymbol_c_popped_rtn::op()// action service
                                    // defined in terminal
....63::popped state:: 14
```

```
...63::exposed sym:: $A
...63::popped state:: 13
destructor -- $A                    // c++ injected code ref::3
..63::exposed sym:: a
63::1--a-> 2--$A-> 3
...63::shift_op token: 0x00524ED0 c pos: 5
....63::Csymbol_c_pushed_rtn::op()// action service
63::1--a-> 2--$A-> 3--c-> 4
....63::reduce_op                   // state 4 lr1ness resolution
....63::Rhs1_$A::op()
constructor $A -- subrule 1
....63::$A::op()
....63::popped state:: 4
...63::exposed sym:: c
...63::Csymbol_c_popped_rtn::op() // action service
                                    // defined in terminal
...63::popped state:: 3
..63::exposed sym:: $A
..63::popped state:: 2
destructor -- $A
.63::exposed sym:: a
63::1--$A-> 7
..63::shift_op token: 0x005253E0 b pos: 6
63::1--$A-> 7--b-> 8
...63::reduce_op
...63::Csymbol_eog_look_ahead_rtn::op()// action service
...63::Rhs1_$S::op()
...63::$S::op()
...63::popped state:: 8
..63::exposed sym:: b
..63::popped state:: 7
.63::exposed sym:: $A
destructor -- $A
63::1--$S-> 11
..63::shift_op token: 0x00525870 eog pos: 7
...63::Csymbol_eog_pushed_rtn::op()
63::1--$S-> 11--eog-> 12            // accepted with action services
...63::accept_op
...63::reduce_op
...63::Csymbol_eog_look_ahead_rtn::op()
...63::Rhs1_$MM::op()               // start rule's rhs
...63::$MM::op()                    // start rule
...63::popped state:: 12
..63::exposed sym:: eog
..63::Csymbol_eog_popped_rtn::op()// even eog action service
..63::popped state:: 11
.63::exposed sym:: $S
..63::C$MM_pushed_rtn::op()         // even the start rule
```

```
                                        // can have action services
                                        // ref::6
63::1--$MM-> 1                          // finally finished
..63::popped state:: 1
.63::exposed sym:: $MM
.63::C$MM_popped_rtn::op()              // action service ref::7
.63::popped state:: 1
Stop of lr1_sp3                         // C++ syntax code ref::2
```

Well this is what its all about: flexibility to execute code in a bottom-up fashion with lots of do-dads for the compiler writer to hang code onto.

C.4.3.2 LR(1) G1 Grammar: Page 61 of [13]

The grammar tests LR1 resolution on states 14 and 15.

Rules			
$G	*$S*	*$A*	*$B*
→ $S eog	→ a $A c	→ f	→ f
	→ a $B d		
	→ b $A d		
	→ b $B c		

1	$G→.$S eog	12	
	$S→.a $A c	2	
	$S→.a $B d	2	
	$S→.b $A d	7	
	$S→.b $B c	7	
2	$S→ a.$A c	3	
	$S→ a.$B d	5	
	$A→.f	14	
	$B→.f	14	
3	$S→ a $A.c	4	
4	$S→ a $A c.		eog
5	$S→ a $B.d	6	
6	$S→ a $B d.		eog

7	$S→ b.$A d	8	
	$S→ b.$B c	10	
	$A→.f	15	
	$B→.f	15	
8	$S→ b $A.d	9	
9	$S→ b $A d.		eog
10	$S→ b $B.c	11	
11	$S→ b $B c.		eog
12	$G→ $S.eog	13	
13	$G→ $S eog.		accept
14*	$A→ f.		c
	$B→ f.		d
15*	$A→ f.		d
	$B→ f.		c

Graphic output of lr parse table.

Index