Implementation of A Compiler-Compiler

by

Neil Joseph Savasta, II

**Submitted in partial fulfillment
of the requirements for the
degree of**

Bachelor of Science

at the

Massachusetts Institute of Technology

June 1984

Signature of Author ..... Neil J. Savasta .....
Department of Electrical Engineering and Computer Science
17 May 1984

Certified by ..... Nancy A. Lynch .....
Thesis Supervisor

Accepted by ..... David Allen .....
Chairman, Departmental Committee

Implementation of A Compiler-Compiler

by

Neil Joseph Savasta, II

Submitted to the
Department of Electrical Engineering and Computer Science
on 17 May 1984 in partial fulfillment of the requirements
for the degree of Bachelor of Science

Abstract

This paper describes the design and implementation of a LALR(1) parser generator CLUCC (CLU Compiler-Compiler). CLUCC accepts as input a BNF-like grammar and produces as output a set of tables for a table-driven shift-reduce parsing routine. The parsing routine and the tables together form a parser which recognizes the language defined by the grammar. The parsing tables generated by CLUCC can be used as the core of a compiler by including calls to user-provided action routines in the grammar. These calls are made by the parser at specified points in the analysis of the input string.

Acknowledgments

The list of people I would like to thank is as long as my arm, I simply cannot name them all, however there are some that cannot go without mention. Many, many thanks go to Professors Nancy Lynch and John Guttag who provided me with this thesis topic and let me take the bull by the horns. I owe Howard Reubenstein a special thanks for his constructive criticisms and being as much a friend as I could ever expect. I have deep gratitude for Paul Bradford for driving me to Nancy's house to get her last minute drafts, and suggesting beer as a solution to some of the problems I encountered. I would also like to thank Kathy Yelick, who put up with my formatting problems, Ron Kownacki, Randy Forgaard, and Gary Leavens for giving me some of their time to handle many of my unanswered questions, and my various office mates who put up with me. A special thanks goes to John Goree who's talents as both a computer scientist and a teacher are an inspiration to me. There are no words that can adequately show the thanks and appreciation I have for Lora Silverman. If there is one person that has helped me through this thesis and MIT, it is her; no one has ever provided me with so much support and understanding in so many ways.

Table of Contents

Chapter One: Introduction 5

1.1 CLU and the Division of the Project 7

Chapter Two: Overview 9

2.1 Notational Conventions 9
2.2 Syntax 9

Chapter Three: Input Subsystem 13

3.1 Scanner 13
3.1.1 Lexical Analysis 13
3.1.2 Interfacing to the Parser 17
3.2 Parsing 17
3.3 Error Recovery 19
3.4 Grammar Building 21

Chapter Four: Processing Subsystem 22

4.1 Grammar Normalization 22
4.2 Function Descriptions 23
4.3 Constructing LALR(1) Sets-of-Items 23
4.3.1 Items, Lookaheads, and Sets-of-Items 24
4.3.2 LR(0) Sets-of-Items Construction 24
4.3.3 ε-Kernels 25
4.4 Lookahead Generation 26
4.5 Action Table 26

Chapter Five: Output Subsystem 28

5.1 The Documentation 28
5.2 The Parsing Tables 29

Chapter Six: Experiences with the Development of CLUCC 33

Chapter One

Introduction

Many tools have been developed specifically to help construct compilers. One of these tools is a compiler-compiler. A compiler-compiler produces the core of a compiler from an input specification. The core is produced in the form of parsing tables, and the input specification is a grammar that describes a the syntax of a source language. A compiler-compiler is useful as a program development tool because it helps produce a compiler quickly and reliably. In addition, compiler-compilers are flexible and adaptable to changes, and encourage an inherent modularity. Many compiler-compilers already exist. A very common and widely available compiler-compiler, available with the UNIX operating system, is YACC (Yet Another Compiler-Compiler). YACC is written in C and was developed by Steve Johnson at Bell Laboratories. YACC takes, as input, an attributed grammar and produces a description of the LALR(1) parsing tables and the code to be used by a shift-reduce parser. A grammar is attributed in that the user may associate a parsing action with each production rule. Parsing actions are performed when a production rule is reduced. Before the symbols matching the right side of a rule are popped off the parse stack, the parsing action associated with the rule is executed. Parsing actions can return values. YACC performs very well on 16- and 32- bit machines, but has some performance difficulties on 36-bit machines. This is a result of C not working well on 36-bit machines because a 36-bit word does not divide evenly into 8-bit bytes. In 1977 Alan Snyder, then a graduate student at MIT, decided to rewrite YACC in C for TOPS-20 (a 36-bit machine). This compiler-compiler was also called YACC but to eliminate any confusion it will hereafter be referred to as YACC-20.

YACC-20 seemed to serve its usefulness, but as time went on the Laboratory for Computer Science at MIT became less involved with C and grew more dependent on its own language, CLU [CLU 79]. In late 1982, Gary Leavens, another MIT graduate student, implemented a compiler-compiler along with a parser in CLU. This new compiler-compiler was not quite a new one, but rather a modification of YACC-20. This modification was a quick, *ad hoc* way of implementing a compiler-compiler in CLU. His solution ran YACC-20 and then converted YACC-20's output, parsing tables in C format, into CLU tables. (The C tables were arrays of integers and were changed into sequences of integers in CLU.) The resulting program became known as CLU-YACC. CLU-YACC, however, has one major drawback: it is entirely TOPS-20 dependent. CLU-YACC was used in an introductory undergraduate course in language engineering during the spring of 1983, and during this course many bugs were found in both CLU-YACC and YACC-20.

At the conclusion of the course, the lecturers decided that a compiler-compiler written from the ground up and solely in CLU was desperately needed. A compiler-compiler written in CLU could have no other name than CLUCC. The main motivation to develop CLUCC came from the need to have a working compiler-compiler in CLU. This motivation was a result of the poor performance of CLU-YACC. Aside from the bugs and the system dependencies that CLU-YACC contains, it is slow because it must do a lot of I/O and string manipulation to convert the C tables to CLU tables. In addition to this, CLU-YACC was written in two different source languages, making it very difficult to maintain. One of the purposes of CLUCC was to provide a clearly documented, efficient compiler-compiler written in CLU. The aim of this thesis is to provide a thorough and comprehensive design and implementation document for CLUCC. This, along with the code, will provide a maintainable software system.

CLUCC is largely based on the algorithms presented by Alfred Aho and Jeffery Ullman in *Principles of Compiler Design.* CLUCC is based on the theory of LR Parsing. It is not within the scope of this thesis to explain this theory, and a reader unfamiliar with the theory is referred to the text. The advantages of LR parsing should, however, be cited. LR Parsers scan the input from left-to-right and construct a rightmost derivation in reverse. LR Parsers can be constructed to recognize virtually all language constructs for which context-free grammars can be written. LR Parsers are more general and dominate the common forms of top-down parsing. Another advantage of LR Parsing is that they can detect syntax errors as soon as possible in a left-to-right scan. [CLU 79] The parsing tables that CLUCC produces are LALR(1) parsing tables. CLUCC produces these tables because they are considerably smaller than the canonical LR tables, and yet most common syntactic constructs can be expressed by an LALR grammar.

1.1 CLU and the Division of the Project

CLU is designed to support modular programming, and thus, a great deal of insight about how the algorithms work can be gained from reading the code. This is unlike other languages, where the code is often not self-documenting. That is not to say that there is no need to document any of the CLU programs. Rather, undocumented CLU programs are, in general, easier to read than undocumented programs of another language, particularly C. Both YACC and YACC-20 contain almost no documentation.

The task of writing a compiler-compiler is a large one, and thus should be broken down into parts that are easily defined and can be implemented and tested incrementally. CLU supports this sort of modular programming very well and allowed for simple clear operations to be designed. A great deal of effort was invested in the overall design and implementation strategy to provide a clear and

well-written product that could be used by students learning about parsing and compiler design theory.

Writing a compiler-compiler is very much like writing any other large programming systems, particularly compilers. Most large programming systems can be broken into three major subsystems: an input subsystem, a processing subsystem, and an output subsystem. Each subsystem is further broken down into phases. CLUCC and compilers are similar in that their input subsystems are essentially the same. Both consist of four phases: scanning, parsing, error recovery, and semantics. The first three phases are self-explanatory. The semantics phase of CLUCC builds an internal form of the grammar in much the same way the semantics phase of a compiler builds an internal form of a program, usually in the form of a symbol table. The processing subsystem consists of five phases: grammar normalization, LR(0) automaton construction, ε-kernel generation, lookahead generation, and action table compilation. The output subsystem consists of two phases: CLU-code output, and documented tables.

The rest of this thesis is organized as follows. Chapter Two is an overview describing the notational conventions used in this thesis and the development of CLUCC's syntax. The next three Chapters discuss the implementation of CLUCC. Chapters Three, Four and Five discuss the Input, Processing and Output subsystems respectively. Chapter Six discusses some of the my experiences in developing CLUCC.

Chapter Two

Overview

2.1 Notational Conventions

The notational convention used in this thesis are explained in [AU 77] *pg. 128.* They are summarized below:

nonterminal symbols *A, B, C,...* , lower case names and the letter *S,*
terminal symbols *a, b, c, ...* and upper case names,
nonterminals or terminals ..., *X, Y, Z,*
strings of terminals ..., *x, y, z,*
strings of grammar symbols $\alpha, \beta, \gamma, \ldots$.

Unless otherwise stated, the left side of the first production is the start symbol.

2.2 Syntax

CLUCC input is described by a grammar and CLUCC, itself, represents another grammar. To eliminate any confusion, the grammar specification of CLUCC shall be referred to as its syntax, and the grammars that CLUCC takes as its input shall be referred to as CLUCC input.

A great amount of care was taken in specifying the syntax for CLUCC in order to maintain a flexible environment and keep the parser for CLUCC down to a reasonable size. CLUCC was designed to interface with CLU and thus should maintain the same conventions as CLU. The syntax for identifiers, literals and numbers is the same in CLUCC as it is in CLU. A nonterminal can only be an identifier, and a terminal can be an identifier or a literal. A nonterminal is somewhat analogous to a variable, likewise a terminal is analogous to a constant. The BNF for a production rule in BNF is:

```
rule ::= LHS "::=" rhs_list

rhs_list ::= empty
         |   rhs_list "|" rhs

rhs ::= symbol_list action

symbol_list ::= empty
            |   symbol_list symbol
```

symbol is either a nonterminal or a terminal, and its BNF is therefore:

```
symbol ::= IDENTIFIER
       |   LITERAL
```

since the left hand side of a production can only be a nonterminal, a **LHS** is an identifier. Note, if the rule for a **rule** were:

```
rule ::= IDENTIFIER "::=" rhs
```

this would result in a shift-reduce conflict. Thus, the syntax for CLUCC would be LR(2). By introducing a new terminal **LHS** into the language this shift-reduce conflict is resolved. Identification of an **LHS** token is done by the scanner. Once the scanner has successfully recognized an identifier, it must lookahead to the next token and check if it is a **"::="** symbol. If it is, then this identifier is a **LHS** token, otherwise it is just an **IDENTIFIER** token. Having the scanner determine whether an **IDENTIFIER** is an **LHS** or just a symbol, weakens the strength of the parser, by making error recovery mechanisms more difficult.

The syntax for an **action** is:

```
action ::= empty
       |   "{" invocation "}"
```

where **invocation** is a subset of the syntax of an **invocation** in CLU. The CLU syntax for a procedure invocation is quite large, almost four times the size of the CLUCC syntax thus specified. CLUCC itself does not contain any reserved words or operators, just separators. CLU has many reserved and words operators that can be used in the syntax of an invocation. In the interest of keeping the parser and scanner small and quick, all reserved words and operators have been eliminated.

This greatly reduces the size of the grammar for an invocation, although it is still quite large. Next, array and record constructors were eliminated. The inclusion of these constructors would increase the size of the parsing tables by thirty percent, and add tokens to the scanner. In addition to the space savings, these constructors can be constructed by calling external procedures. If these constructors are left in they may also create some semantic problems that would go unnoticed until the parsing tables are compiled. This is a result of CLUCC not being aware of the compilation environment that the tables are going to be used in. If the complete CLU syntax of an invocation or the constructors were used, it would require a good deal of semantic checking. This seems to be rather inefficient since CLU will do this when the parsing tables are compiled. Rather than add a great amount of complexity to CLUCC, it seems a reasonable trade off to limit the syntax in this way.

The syntax for an **action** in CLUCC can now be fully described:

```
action ::= "{" invocation "}"

invocation ::= primary "(" opt_expression_list ")"

primary ::= primary "." element
          |   primary "[" expression "]"
          |   invocation
          |   type_spec
          |   type_spec "$" type_spec

type_spec ::= IDENTIFIER
            |   IDENTIFIER "[" expression_list "]"

opt_expression_list ::= empty
                      |   expression_list

expression_list ::= expression
                  |   expression_list "," expression

expression ::= primary
             |   type_spec
             |   IDENTIFIER
             |   LITERAL
             |   NUMBER
             |   "#" NUMBER
```

The addition of the production:

```
expression ::= "#" NUMBER
```

refers to the attribute of the **NUMBER**th symbol on the right hand side of the current production. This is useful for passing semantic information along with the parse. Each action is required to return an attribute which is assigned to the left hand side nonterminal of the production the parser is reducing. If the action does not return an attribute, then an error will result when the parsing tables are compiled. The attributes are user defined, but must be called **attribs** and support the following operations:

```
where attribs has
      make_error: proctype( null ) returns( attribs )
      make_undef: proctype( null ) returns( attribs ),
      make_empty: proctype( null ) returns( attribs ),
```

The first operation is used by the parser when an error arises. The second operation is used when the parser is not executing the user supplied actions (for example, the parser does not execute any user-actions during error recovery, but an attribute must be created). The last operation is used for production rules that do not have actions associated with them.

In addition to its syntax, CLUCC, like CLU, does not distinguish between upper- and lower case letters. In an effort to force CLUCC input to look more preventable, an **LHS** token must also be the first token on a line. Literals that are used as grammar symbols, have their blanks and leading and trailing quotes stripped, and all upper case letters changed to their lower case letters. The resulting literal may not be empty.

Chapter Three

Input Subsystem

3.1 Scanner

The purpose of the scanner is to provide a stream of tokens to the parser and to report lexical error messages. To provide a stream of tokens to the parser, the scanner performs the lexical analysis of the source stream and interfaces to the parser. These two functions are performed by two main clusters in the scanner. The lexical analysis cluster, `lex`, determines what the tokens are and the token cluster, `token`, has many operations which create tokens suitable for the parser. The lexical analysis and token clusters are separated for modularity, the lexical analyzer recognizes tokens independently of the parser, and the token cluster serves as the interface between the lexical analyzer and a particular parser.

The scanner that was designed and implemented for CLUCC was a very straightforward scanner that can easily be adapted to handle tokens for another language. While the scanner is general, a great amount of effort went into making it very efficient.

3.1.1 Lexical Analysis

A `lex` object is internally represented as a record which contains the following elements:

- a source stream from which `lex` gets its tokens,
- a lookahead buffer that is used during error recovery and for determining `LHS` nonterminals.
- a compiler state for logging scanner errors.

- the current line in the source stream being scanned, and
- a new line flag which is true if and only if no tokens have been read from the current line.

The `lex` cluster performs four operations:

`create`	create a new `lex` given the name of the source file, and a compiler state
`close`	close the source stream.
`insert`	insert a token in the front of the lookahead buffer, and
`get_next_token`	return the next available token.

The first three operations are implemented just as they are described. `get_next_token` is not quite as straightforward. `get_next_token` performs the lexical analysis. It is a modified implementation of an FSA and is dependent on a character class table. This table contains 128 elements and is based on the 7-bit ASCII representation of each character. The *index* of an element in the table corresponds to the character with an ASCII value of *index - 1*. Each element contains one of nine values which define its class. There are nine classes:

letter	`{ "A" - "Z", "", "a" - "z" }`
digit	`{ "0" - "9" }`
colon	`{ ":" }`
slash	`{ "\" }`
blank	`{ <SP>, <BS>, <VT>, <FF>, <HT>, <CR> }`
end-of-line	`{ <LF> }`
separator	`{ "#", "$", "(", ")", ",", ".", "[", "]", "{", "|" ,"}" }`
quote	`{ "'", """ }`
error	anything else not defined above

From this table an efficient lexical analyzer can be written. The operation `get_next_token` first checks if the buffer is empty. If the buffer is not empty, then the token on top of the lookahead buffer is popped off and returned. Otherwise, a token must be read from the source file. All comments, blanks, and end-of-lines are eliminated. Note, a comment in CLUCC is the same as a comment in CLU.

The following definitions have been establish for the tokens:

`LITERAL`	any group of characters surrounded by matching quotes.
`IDENTIFIER`	*letter* (*letter* | *digit*)*
`":::="`	`":::="` *(referred to as the derivation symbol)*
`"\\"`	`"\\"`
`LHS`	`IDENTIFIER` *(context sensitive)* A LHS must be the first token on a line and be immediately followed by a derivation symbol.
separators	as defined in the character class table.

Once all comments, blanks, and line terminators have been eliminated, the class is obtained for the next available character in the source stream. A lexical error may occur in some of the following modules. When an error occurs, no token can be returned. Therefore, an internal error routine is called which logs the error with the `compiler_state` and calls `get_next_token` to get the next valid token to return. A different routine is called for each class. Since all blanks, comments, and line terminators have been eliminated, the only classes which are possible are *error*, *separator*, *slash*, *colon*, *quote*, *digit*, and *letter*.

error. If the character is an error class character, then the internal error procedure previously described is invoked.

separator. A separator class character calls `token$make_separator` with the character and the current line number.

slash. This calls the `get_slash_slash` routine, and its purpose is to recognize a `"\\"` delimiter. It checks if the next character is a `"\"`, if it is, it returns `slashslash` token, otherwise the internal error handling procedure is called for with the error being the first back-slash character.

colon. This calls the `get_derivation_symbol` routine, and its purpose is to recognize

a derivation symbol. This procedure works analogously to **get_slash_slash**, except that it must go a little further. First, it checks if the next character is a ":", if it is, then this colon is read from the source stream, otherwise the internal error handling procedure is called for with the error being the first colon character. At this point "::" has been recognized. Next, it checks if the next character is an "=", if it is, it returns **derivation_symbol** token, otherwise **get_derivation_symbol** is called. It is called again because the scanner has detected a colon.

quote. This calls **get_literal** routine, and its purpose is to recognize literals. This procedure simply reads the source stream until it reads a matching quote or an end-of-line. If an *end-of-line* character is read, then an error message is logged stating that the literal was improperly terminated. In either case, a **literal** token is returned, and the literal will have a trailing quote that matches its leading quote.

digit. This calls **get_number** routine, and its purpose is to recognize numbers. This procedure simply reads digits from the source stream and appends them to a string until it reads a non-digit character. Once it reads a non-digit character, a number is recognized and **token$make_number** is called with the number string and the line number.

letter. This calls the **get_either_symbol** routine, and its purpose is to recognize symbols. This procedure simply reads characters from the source stream and appends them to a string while it reads a letter or digit class character. Once it has read a symbol, it must be determined whether this symbol is an **LHS** or just a name symbol. In order to determine this, it is necessary to establish the context of this symbol. Remember a symbol is an **LHS** if it is followed by a derivation symbol. This is accomplished by getting the next token, inserting it into the lookahead buffer, and checking if if it is a derivation symbol. If it is a derivation symbol then this symbol is a nonterminal symbol, otherwise this symbol is a name symbol. This new token is

placed in the front of the lookahead buffer because of the recursive nature of this operation.

If the *end-of-file* is reached, then an end-of-file token is returned.

3.1.2 Interfacing to the Parser

The basic purpose of the token cluster is to assign numbers to each of the tokens so that the parser and error recovery phases can recognize them. Token numbers must conform to the numbers associated with each terminal by CLUCC. The number of a token is in the range [1, ... *n*], where *n* is the total number of tokens. A token is a structure consisting of three elements, a token number that is used by the parser, a line number for this token, and an attribute, which contains the semantic information about a token. The attribute for a token is simply the string that the scanner has read in. There are several `make_` operations which construct particular tokens, and three `get_` operations each of which return one of the elements in the structure. Along with these operations is an operation that will return true if a token is a derivation symbol, and false otherwise; an operation that will make dummy tokens given the token number and the line number; this is used in the error recovery section; and an operation, converts a token value to its string value. This last operation actually returns the string value of the token which is contained in the attribute.

3.2 Parsing

CLUCC's parsing phase was actually implemented twice. The first time the parser was written using CLU-YACC, and when CLUCC was finished it was used to help bootstrap CLUCC. Even though CLU-YACC has some bugs it did generate correct parsing tables for CLUCC. These parsing tables successfully parsed CLUCC's

syntax. This parser only needed to correctly parse CLUCC's syntax because the parsing tables that CLUCC would produce for itself would be substituted for the parsing tables that CLU-YACC produced. This was not, however, the only test case that this parser was given, it was also successfully tested with a grammar for a subset of Modula2. This extra test was not necessary at this point, but it proved to be useful input when testing later phases.

CLU-YACC and CLUCC use essentially the same parser. When CLU-YACC outputs the tables it outputs them lexically contained within the parser. The CLU-YACC parser references the tables through two internal functions `lraction` and `lrgoto`. The CLUCC parser is effectively the same parser except that the tables, and the action and goto functions exist in the `lrtables` cluster, and the parser is a stand alone cluster that is independent of any particular grammar. The parser is independent of the parsing tables for modularity. The implementation also adds flexibility and time savings to the compiler being built. The user is not forced to use a specific parser with his tables, and is free to use another LR Parser. Since the parser is independent of the tables, there is no need to keep recompiling the tables or the parser while developing the other.

The parser, in both cases, is implemented as a deterministic push-down automaton (DPDA). Each frame on the stack contains three values: a grammar symbol, a state, and the attribute associated with the state. The state on top of the parse stack is related to preceding states by the action tables and goto tables constructed for the grammar. Several of the parsing operations support syntax error recovery.

3.3 Error Recovery

A very crude error recovery could have been accomplished by reading tokens until an LHS or end-of-file token is read. If an end-of-file token is read, then the error cannot be recovered from, and parsing halts. Otherwise, this token is an LHS and states are popped from the stack until an LHS token can be legally read. This recovery scheme, while crude, would work very well with the syntax for BNF. However, the syntax of CLUCC is much more complicated than a BNF syntax because of the syntax of an action. Because of this added complexity, an error recovery scheme like this would prove to be inadequate, and a more general error recovery scheme is necessary. Even though the syntax of an action makes the CLUCC syntax more complicated, the syntax is still rather simple and the error recovery schemes necessary for each possible error are similar.

The error recovery scheme that CLUCC uses is a simple bounded range error recovery scheme. The basic strategy is to make a patch, and then try to parse ahead a fixed number of tokens, *n*, called the *bound*, to see if the patch is correct.

When CLUCC first encounters a syntax error, the error recovery scheme is invoked, and an error message is written stating which line number the error occurred on, what the contents of the token were, and what types of tokens it was expecting. CLUCC then tries to patch the error. Patches fall into four different categories: insert a token, replace the current token with another token, push a nonterminal onto the parse stack, and pop the top state off the stack. Deleting a token is considered a last ditch effort, and is only used when the other four types of patches fail. It is considered a last ditch effort because discarding input should be avoided unless absolutely necessary.

A patch is considered to be successful if the parser can parse ahead *n* tokens without a syntax error, otherwise a patch is not successful and another one must be tried. If

all the patches are tried and none are successful, then the current token is deleted from the token stream. This deletion, however, cannot occur if the current token is an end-of-file token, if this occurs then, the error recovery has failed and a fatal error is generated.

The minimum distance the error recovery scheme must parse ahead is seven tokens. A smaller *bound* would not adequately test the patch to see if it were good and a larger *bound*, along with being more expensive, may encounter another error and therefore not allow recovery from the current error. If another error is encountered within *n* tokens then the original token will automatically be deleted because no other patch will work. This has dramatic effects because the recovery scheme may delete all the tokens between the two tokens.

The order in which patches are tried is as follows: insert a token into the token stream before the current token, replace the current token in the token stream, push a nonterminal onto the parse stack, and pop the top state from the parse stack. Once again, deleting the current token is the last operation performed. This is the order because it the order of these patches minimizes the loss of information. Replacing a token does discard some input, but there is no net loss of input since the token is replaced with another token. Any state can be popped off the stack, because the parser must parse ahead *n* tokens and this should prevent trouble from occurring. If this patch does not work, the only alternative is to delete the current input token.

There is a list of terminals which is used to sort and eliminate tokens from the parsers' expected token list. This terminal list determines which terminals will be inserted or replaced. The sort is performed by intersecting this terminal list with the expected token list. The order of the resulting list is consistent with the order in the terminal list. There is a corresponding nonterminal list that determines which nonterminals will be pushed onto the parse stack. These lists are ordered in

preference of a patch, in other words, for example, there may exists two or more terminals in the terminal list which will successfully parse ahead *n* tokens. The error recovery scheme will take its first successful patch; therefore, the terminal list and the nonterminal list should be ordered according to their preference for a successful patch. Terminals and nonterminals that are undesirable at any cost must be left out of their respective lists.

3.4 Grammar Building

The next phase of the project builds a representation of the input grammar and associates an action with each production rule.

The grammar and actions are built by executing the translation rules that are associated with the syntax reduction being performed. The grammar is initialized with the list of tokens that precedes the rules. The semantic rules build the production rules one at a time. Internally a rule is a record that consists of two parts, a left hand side and a right hand side list. The left hand side is a string, and the right hand side list is an array of right hand sides. A right hand side is a record that consists of a symbol list and an action. The symbol list, is just a list of strings, and the action is also just a string that represents a procedure call or is empty in the case where no action is given. As each rule is synthesized it is inserted into the grammar. If the left hand side has already been inserted into the grammar then the right hand side list of this new rule is appended to the right hand side list of the existing rule. The tokens which constitute an `invocation` are simply concatenated together to generate a procedure call. The only exception is: `"#" NUMBER`, which is replaced by `pv[ NUMBER ]`, where `pv` is the name of the array containing the attributes. `pv` stands for production variables. The grammar rules are represented by an AVL tree. An additional list is also maintained to hold the order in which the nonterminals and appear on the left hand side of the CLUCC input. The left hand side of the first production symbol entered into the grammar is the start symbol.

Chapter Four

Processing Subsystem

4.1 Grammar Normalization

Once the grammar specification has been parsed, another pass is made over the grammar. This pass assigns a distinct integer to each grammar symbol. The set of integers associated with the nonterminals ranges from one to the total number of nonterminals. Each nonterminal is assigned a number based on its first occurrence on the left hand side of a production rule. The terminal numbers range from the number of nonterminals plus one to the number of nonterminals plus the number of terminals. The first terminal is assigned the value of the number of nonterminals plus one. Each terminal is assigned a number based on its first appearance in the grammar. Usually the order of the terminals is defined in the prefix section, but if a symbol appears in the right hand side of some production and does not appear on the left hand side of any production then it is assumed to be a terminal. Mapping all the grammar symbols into a continuous set of integers allows for compact and efficient representation of the structures that depend on the grammar. Since terminals and nonterminals have a numeric representation, information about sets of them is stored in bit vectors. Bit vectors provide an efficient way to store sets of terminals and nonterminals. By using the bit vectors as sets, elements can easily be added, and deleted. Bit vectors also merge two sets together very efficiently. Merging quickly will be very useful in later parts of the project.

4.2 Function Descriptions

After the grammar has been normalized, there are a five sets of computations that are useful to perform before computing the tables. They are:

EMPTY	The set of all nonterminals A such that $A \rightarrow^* \varepsilon$.
FIRST	For each nonterminal A in the grammar, the set of all terminals a such that and $A \Rightarrow^* a\gamma$.
LEFTNONTERM	For each nonterminal A in the grammar, the set of all nonterminals B such that $A \Rightarrow^*_{rm} Bz$ and the last step does not use an ε-production.
LEFTTERM	For each nonterminal A in the grammar, the set of all terminals a such that $A \Rightarrow^*_{rm} az$ and the last step does not use an ε-production.
LEFTEMPTY	For each nonterminal A in the grammar, the set of all nonterminals B such that B is a member of LEFTNONTERM(A) and and $B \rightarrow \varepsilon$

These functions help generate the parsing tables for a grammar. The *rm* of $\Rightarrow^*_{rm}$ means that the derivation is a *rightmost* derivation. These functions can be calculated quite easily and quickly using bit vectors as previously discussed. The computation of the LEFTTERM, LEFTNONTERM and LEFTEMPTY functions can be carried out simultaneously, because of the recursive depth-first nature of the production rules.

4.3 Constructing LALR(1) Sets-of-Items

The parser is a one operation cluster rather than a procedure. This was done to isolate the internal operations that are used only to generate the parsing tables.

4.3.1 Items, Lookaheads, and Sets-of-Items

An *item* is a dotted production rule. The dot indicates how much of the right-hand-side has been seen by the parser when the parser is in that state. An item also contains a lookahead set, to assist the parser in making action decisions. A lookahead set is a bit vector. This greatly helps the calculating speed of the lookaheads.

A *set-of-items* is an ordered set of items. The symbol to the immediate right of the dot is a transition symbol for the state identified by this set-of-items. The items are sorted based on the production number and the position of the dot. Each state is identified by its set-of-items and completely specified when all the GOTO transition information has been generated.

The act of *closing* a set-of-items consists of adding certain new dotted productions to the existing set-of-items. For each symbol immediately to the right of a dot, the set of productions with this non-terminal symbol as a left hand side is added, with the dot appearing in the leftmost position of the right hand side. In the case of a canonical LR(1) closure, lookaheads are propagated from one item the the next.

A *kernel* is a set of items. It is the collection of those items not added in the closure.

4.3.2 LR(0) Sets-of-Items Construction

The GOTO automaton consists of parsing states and the transitions between them. In order to keep track of the construction of the GOTO automaton, an "exploration list" will be kept. The exploration list contains those states that have not had their GOTO states generated. Each element of this list is a state from which exploration is still needed, together with those grammar symbols for which shifts are defined from that state, and for which exploration from that state still needs to be carried out. This exploration list is a queue and the automaton is generated in breadth-first order.

Initially, this list consists of only a standard initial state, and that state is the only state in the initial automaton. Now, repeatedly states are removed from the exploration list until it is empty. When state s is removed from the exploration list, the goto states are generated for state s. All the goto states for a state are generated at the same time. The goto states are computed by going through each item in the kernel and generating all its goto items. For each item, $[A \rightarrow \alpha.X\beta]$, the goto items are all the items such that $[B \rightarrow Y.\gamma]$, where B is in LEFTNONTERM(X). $[A \rightarrow \alpha X.\beta]$ is also a goto item of this item. The goto items are grouped into states according to the grammar symbol to the left of the dot. This symbol is the transition symbol. Next, the automaton is checked to see if any of the goto states already exist. If a goto state already exists, then a new edge from state s to this goto state is added the automaton. If a goto state is new, then the automaton is augmented with a new state and an edge, and the new state is placed in the exploration list. The goto states are entered into the exploration list so that their goto states can be found.

4.3.3 ε-Kernels

An *ε-kernel* of a set-of-items is defined to be the kernel and those items $[C \rightarrow .]$ such that there exists an item $[A \rightarrow \beta.C\delta]$ in the kernel and $B \Rightarrow^*_{rm} Cx$ and $C \rightarrow \varepsilon$. Empty productions are added to the kernels because computing reductions becomes quite complicated without them. A reduction of the form $C \rightarrow \varepsilon$ is called for on input a if and only if there is a kernel item $[A \rightarrow \beta.B\gamma, b]$ such that $B \Rightarrow^*_{rm} C\delta$ for some δ, and a is in FIRST($\delta\gamma b$). This definition is quite complicated, but by adding all empty productions to the kernel a reduction of the form $C \rightarrow \varepsilon$ is called for on input a if and only if there is a kernel item $[C \rightarrow ., a]$. A The kernels already exist from the LR(0) construction, and all that needs to be added are the rules that derive an empty string. This is done by going through each state and examining each item in the kernel. If $[A \rightarrow \beta.C\delta]$ is in the kernel and $B \Rightarrow^*_{rm} Cx$ and $C \rightarrow \varepsilon$, then item $[C \rightarrow .]$ is added to the kernel. LEFTEMPTY contains the set of nonterminals C such that $B \Rightarrow^*_{rm} Cx$ and $C \rightarrow .$.

4.4 Lookahead Generation

Lookaheads are generated for each set of items much in the same way to goto states for the LR(0) automaton are generated. The initial item has the *end-of-file* terminal for a lookahead, and is placed in a new exploration list. This exploration list contains items that have to propagate lookaheads to their goto states. Again, states are removed from the list and processed until the list is empty. The Canonical LR(1) Closure [AU 77] is computed for each set of items, *s*, as it is removed from the closure. For each item, $[A \rightarrow \alpha.X\beta]$, in this closure, the lookaheads are propagated to $[A \rightarrow \alpha X.\beta]$ in the goto state of under the transition symbol X. If any of the lookaheads were not in $[A \rightarrow \alpha X.\beta]$ then this goto state is placed in the exploration list.

4.5 Action Table

The actions for each state are computed as follows: an item that calls for a reduction will be in the kernel and a shift will occur for all terminals a such that $X \Rightarrow^{*}_{rm} ax$ such that $[A \rightarrow \alpha.X\beta, b]$ is in the kernel. First all the shift actions are computed, and then the reduce actions are computed. The shift actions are simply entered into the table. The reduce actions must be check for shift-reduce and reduce-reduce conflicts, and if a conflict arises, it must be resolved. In addition to this, each reduce action must be counted, so that the most frequent reduce action will be the default action for that state. An accept occurs when the first production is reduced, this is checked before the reduction is entered into the table.

A conflict indicates that the grammar is not LALR(1). These conflicts, however, may be intentional, and so a crude mechanism for handling conflicts exists for resolving them. Shift/reduce conflicts are resolved in favor of shifting. Reduce/reduce conflicts are resolved in favor of the production appearing earlier in the input file.

In order to represent the parsing tables compactly, there are several space optimizations that can be performed. Many states have the same actions, and a great amount of space can be saved if a pointer is created for each state to a list of actions for that state. Pointers for states with the same actions point to the same list of actions. Further space efficiency can be achieved by creating a default action. The default action would be the most frequent reduce action in the state. States with a reduce action can be considerably compacted by the addition of a "default" condition for the most popular reduction. The only apparent difficulty with the above optimization is that delayed error detection might allow certain very obscure errors to pass undetected, but this in fact is not true. If the lookahead symbol were not in the original complete lookahead set, then the "default" action would be taken. However, a subsequent state would eventually be forced to shift the next token. This token in fact would not be legal in any subsequent state since it was not included in the lookahead state (the state is created by looking at the surrounding states). Therefore, the error will still have been detected. Since the ACTION function determines its results by searching through linear lists, then any reduction in the size of these lists will obviously increase the parsing speed. Therefore, any compaction of this sort is of great value.

Using the compaction techniques described above, it is very easy to generate the action table. The process simply examines each state and makes the actions for that state. Once the actions for that state have been determined, the list of actions is entered into the action table and an offset is returned. The table will only contain distinct action lists, and therefore states with identical actions have the same offset value.

Chapter Five

Output Subsystem

Once CLUCC has finished computing all the actions, it will create two output files. The first file is a documentation file that describes the LALR(1) parsing tables, and the second file contains the CLU code for the LALR(1) parsing tables. The documentation file can be used to understand the parsing states. The CLU code contains the tables needed by the parser.

5.1 The Documentation

This file contains description of the grammar, the goto and action tables. In addition to these descriptions, this file will contain information about conflicts that may have arisen in the process of constructing the tables for the grammar. The modules that write out these descriptions are quite straightforward.

The description of the grammar consists of three parts: terminals, nonterminals, and production rules. Each of these three grammar parts is numbered from one to the number of elements in each part. The terminal and nonterminal numbers correspond to the same numbers that are used by scanning, parsing, and error recovery phases of a compiler. The production rules are numbered so that it is easier to describe reductions in the action table section.

The goto table description is organized by nonterminals. This is also how it is organized in the CLU tables file. The description is a list of all the nonterminals with their lists of corresponding state transitions pairs, (*current*, *next*).

The action table description is a description of each state in the LR(0) automaton. The description of a state consists of two parts, the ε-kernel and the actions. The ε-kernel is a set of items, and each item is printed out as a production rule with its dot (`"."`) in the correct position. The actions are printed out according to the terminal symbol. The shift and reduce actions have the form:

shift *s*	where *s* is the next state.
reduce *p*	where *p* is the production rule number.

Since an accept action signals that parsing has been completed successfully, and an error action only occurs as a default action, no arguments accompany these actions. The default action is the last action printed for a state and is preceded by a dot `"."`.

If conflicts exist in the grammar, they are written out first. A conflict is described by its state, the two actions which are in conflict, and the terminal that caused it. The description file may be used to determine the cause of these conflicts by examining the description of the particular states where the conflicts occurred.

5.2 The Parsing Tables

The parsing tables produced by CLUCC are in the form of a CLU cluster called **lrtables**. The parsing tables are embedded lexically in the the cluster. The cluster consists of three parts, a head, the parsing tables, and a tail. The head of the cluster has the names of the operations that may be performed on the tables, and the tail of the cluster contains the code for the operations. The head and tail of the cluster are always the same, independent of the CLUCC input. **lrtables** provides six external operations to use with a shift-reduce parser:

action	lookup the action given a state and and a terminal.
goto	lookup the next state given a state and a nonterminal.
terminal	fetch the string associated with a given terminal number.

`nonterminal`	fetch string associated with a given nonterminal number.
`termcount`	return the number of terminals in the input grammar.
`nontermcount`	return the number of nonterminals in the input grammar.

The tables are output in the middle of the cluster, and turned into sequences of integers, and strings. Integers are unparsed into strings, and strings have double quotes appended to the beginning and end. There is one other type of sequence in the CLU file, the sequence of procedure calls to be associated with each reduction. These are referenced by name only. The procedures themselves are also output into the middle of the cluster. The procedures have the form:

```
ruleN =
proc( cstate: compiler_state, pv: attributes )
  returns( attribs )
    return( PROC_CALL )
    end ruleN
```

where `N` is the production rule number, and `PROC_CALL` is the procedure call associated with this production rule in the clucc input. If a procedure call was not associated with a production rule, then the default procedure rule is `empty_rule`. `empty_rule` will return the first attribute on the right side of the production; if the right side is empty, an empty attribute is returned.

The contents of the parsing tables fall into three categories: the grammar sequences, the goto sequences, and the action sequences.

The grammar sequences consist of five sequences. The first two sequences are sequences of strings. One contains the string names of each of the terminals, and the other one contains the string names of each of the non-terminals. The next three sequences contain information about the grammar rules. The index of the sequences corresponds to the production rule number. One contains the sizes of the right hand side of each production. One contains the number of the nonterminal for that production. The last sequence contains the procedure names that will be called upon the reduction of a production rule.

The primary function of the goto table is to choose the next state after a reduction. Thus there is no need to keep information about terminals and their transitions in the goto table. The goto table is a list of nonterminals followed by a list of pairs of states (*current*, *next*). The goto table is organized by nonterminals for space efficiency. All the transitions in this list are valid under that nonterminal. There are three integer sequences which comprise the goto table. The first two sequences contain all the valid (*current*, *next*) transitions, one sequence contains the *current* states, and the other contains the *next* states. The last sequence contains the offsets into these sequences for each nonterminal. The index of the offset sequence is the nonterminal number, and its content is the offset into the transition sequences. The offset sequence has one more element than the total number of nonterminals. The offset sequence is an ordered sequence, *element* $i \leq$ *element* $i+1$. This means that transitions for nonterminal i are located in positions *element* i through *(element* $i+1$*)-1* of the transition sequences.

The action tables is made up of four integer sequences. There are two offset sequences. These two offset sequences are used because of the table compaction algorithm for states with identical actions presented earlier. The first offset sequence contains the offsets for each state into the second offset sequence. The second offset sequence contains offsets into the action sequence. The action sequence is organized in groups of three elements starting from the first position. The first element in the group is the terminal number for the lookahead, the second number is that action number, and the third number is the argument for the action. Since error and accept actions have no arguments, the element is not consulted for this argument. For a shift action this argument is the next state number, and for a reduce action this argument is the production number. The list of actions valid for a particular state is terminated by a -1 in the first position. The last sequence contains the defaults actions. The default action for a particular state is located at *element s*

where *s* is the state number. The default action will be the production number in the case of a reduction, or zero in the case of an error.

Chapter Six

Experiences with the Development of CLUCC

CLUCC has been designed and implemented to efficiently produce a parser in time and space. The one shortcoming CLUCC has with respect to YACC is that there is no mechanism for controlling the use of ambiguous grammars. YACC controls the use of ambiguous grammars, by specifying precedence and associativity.

CLUCC took about 25 40-hour weeks to write and debug. In that time many different versions evolved in an attempt to gain time and space efficiency wherever possible. For example, the normalization of the grammar, and the subsequent use of bit vectors lead to an improvement of almost an order of magnitude for the time necessary to generate the lookaheads.

In order to gather timing statistics CLUCC was made to display the cpu time at similar points to YACC-20 during the parsing table generation. The total cpu time used by CLUCC to produce parsing tables in CLU is about five times greater than the cpu time used by YACC-20 to produce parsing tables in C for the same input. This time factor is relatively the same for different sized grammars. This factor drops a little with very large grammars. This time factor can largely be attributed to the use of CLU instead of C. YACC-20 displayed the cpu time as it generated to parsing tables. The timing statistics are not exactly comparable because the CLU cpu time includes the time used for paging and C does not include this time.

When CLUCC is compared with CLU-YACC, it is found that CLUCC is about four times faster than CLU-YACC. This statistic is purely speculative because the cpu time for CLU-YACC to process the YACC-20 parsing tables is not displayed.

The time it takes to convert the tables from C to CLU can only be estimated. However rough an estimate, this is the important time statistic, since we are concerned with the performance of compiler-compilers which generate CLU code, not C code. CLUCC is also more practical than CLU-YACC because the parsing tables are independent of the parser. This is advantageous because there is no need to recompile the parser and the tables when changes are made to only one of them.

Displaying the cpu time at intervals in the generation process, proved to be very useful. The breakdown of the total cpu time showed where the bottlenecks were in CLUCC. The time used by YACC-20 and CLUCC for I/O is essentially the same. The time to generate the tables using CLUCC is much greater than the time it takes in C. The time used to compute the lookaheads using CLUCC is about ten times slower than using YACC-20. Fortunately, the lookahead computation occupies the least amount of time relative to the other sections.

Virtually no bugs have been discovered in CLUCC. This is a result of the way the project was divided so that it could be incrementally tested. Testing was performed bottom up; as something was added, it was tested. The testing was greatly simplified because CLU codes algorithms so well, and it is easy to see which test cases are necessary. This eliminates a great deal of testing redundancy, and results in a great time savings. Much of the testing was also simplified because CLU supports modular programming so well.

CLUCC was used in an introductory course in compiler design at MIT this past spring. This proved to be the best testing ground of all. Students are notoriously good at finding bugs that exist in a program and MIT students are among the best. There were one or two bugs discovered in the parser provided to work with the tables that CLUCC generated, but aside from these bugs CLUCC has worked marvelously.

References

[AU 77]
Aho, A. V. and Ullman, J. D.
Principles of Compiler Design.
Addison-Wesley, 1977.

[CLU 79]
B. Liskov et al.
CLU Reference Manual.
Technical Report TR-225, Laboratory for Computer Science, Massachusetts Institute of Technology, 1979.