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.