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.