In this paper we will assume that the reader has a knowledge of LL(1) tables and the C++ language, as well as a working knowledge of context-free grammar. We will further discuss context-free grammar and how it works together with an LL(1) table and C++ in the construction of our parser. Context-free grammar played a crucial role in the design and construction of our parser. The method of parsing that we used, which was based off of an LL(1) table, was entirely built on context-free grammar. All the rules are built in the context-free grammar style. That is, each non-terminal can go to one or more terminals or non-terminals. In our Grammar.cpp the function for parsing <addingOperator> is a good example of this. *AddingOperator* is a non-terminal, with regards to our particular grammar, this means that <addingOperator> is not a token. Any terminal is a token, whereas any non-terminal is not a token but a rule. So, the non-terminal <addingOperator> leads to a variety of terminals (whereas some rules also lead to other non-terminals and/or terminals) through certain rules which are traced in the LL(1) table. A more traditional way of writing a rule in a context-free grammar might be as follows: <addingOperator> -> Plus; <addingOperator> -> Minus; <addingOperator> -> Or; or <addingOperator> -> Plus | Minus | Or. In our project, we simply translated this to C++ code using switch statements:

```
bool Grammar:: addingOperator(SemanticRecord& record){
switch (nextTokenType()) {
                case MP_PLUS:
                        …
```

```
                return true;
        case MP_MINUS:
                …
                return true;
        case MP_OR:
                …
                return true;
default:
                error(TypeList() << MP_PLUS << MP_MINUS << MP_OR );
    }
```

What is actually happening here is that the current function we are looking at is the function for the non-terminal <addingOperator>. This being an implementation of LL(1), we look ahead one spot and hope to see one of the three terminals, or tokens, this rule can accept. Based on the grammar, <addingOperator> is allowed to go to any one of these three tokens and nothing else. Therefor, if what is read next is not one of these three tokens, an error is thrown telling the user what was expected. If however, what is read next is one of the three acceptable tokens a variety of things happens. The token is accepted by the switch statement and then all that needs to happen for that accept state (such as logging the rule, matching the token, writing the machine code, etc.) is told to happen and the parser moves on

to the next spot to repeat the process all over again. This process ends when the eof token is reached.

There are occasionally conflicts in the grammar, meaning that a non-terminal can go to a single terminal by more than one rule (eg. <factor> -> <identifier> by rule 106 or 116). This suggests that the grammar was not designed as an LL(1) grammar, meaning that the intention was, perhaps, to look ahead more than one spot. As we were using LL(1) (looking ahead only one spot) and that was not to change, we had to choose how to deal with these conflicts. What we ended up doing was limiting the grammar by simply choosing one of the rules and only letting the grammar take that path; in the case of <factor>, we modified it so that it was only allowed to get to <identifier> via rule 106 only and was not allowed to go to <identifier> via rule 116.

Context-free grammar, therefor, was crucial in building our compiler and was the rock upon which our parser was built. Using the LL(1) method had a few tradeoffs, but overall worked very well for parsing this context-free grammar.