Project 1: Syntax Analyzer (Parser)

Stephen (Blake) Sanders, Jared Wermager, Austin Johns

Professor Nasser Tadayon

CST-405

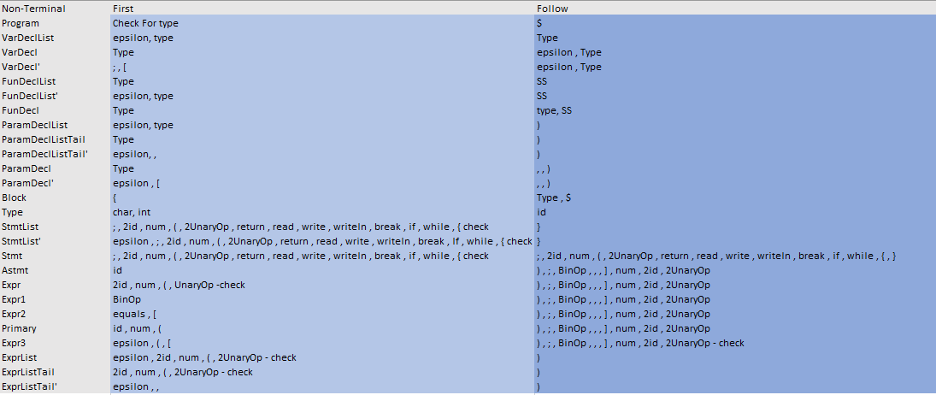
*Planning*

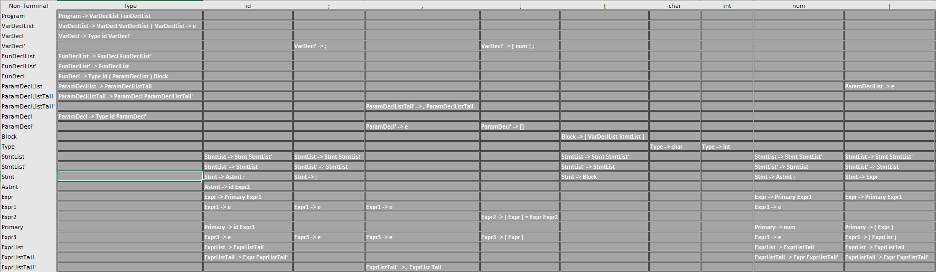
The planning stage represented the general concepts that the syntax analyzer project represented. We did a lot of this process in class by going over the steps that would be necessary in order to complete a working syntax analyzer. The first step was getting the lexer we built running properly and without any issues. Once we had this working, the next step was to take the grammar and implement the rules into a syntax analyzer that recognized the stream of tokens from the lexical analyzer. The syntax analyzer then determines if the tokens were in an appropriate order and if it matched the rules of the grammar. In cases that the stream of tokens fell outside of every rule, the syntax analyzer would return the user with a syntax error.

*Implementation*

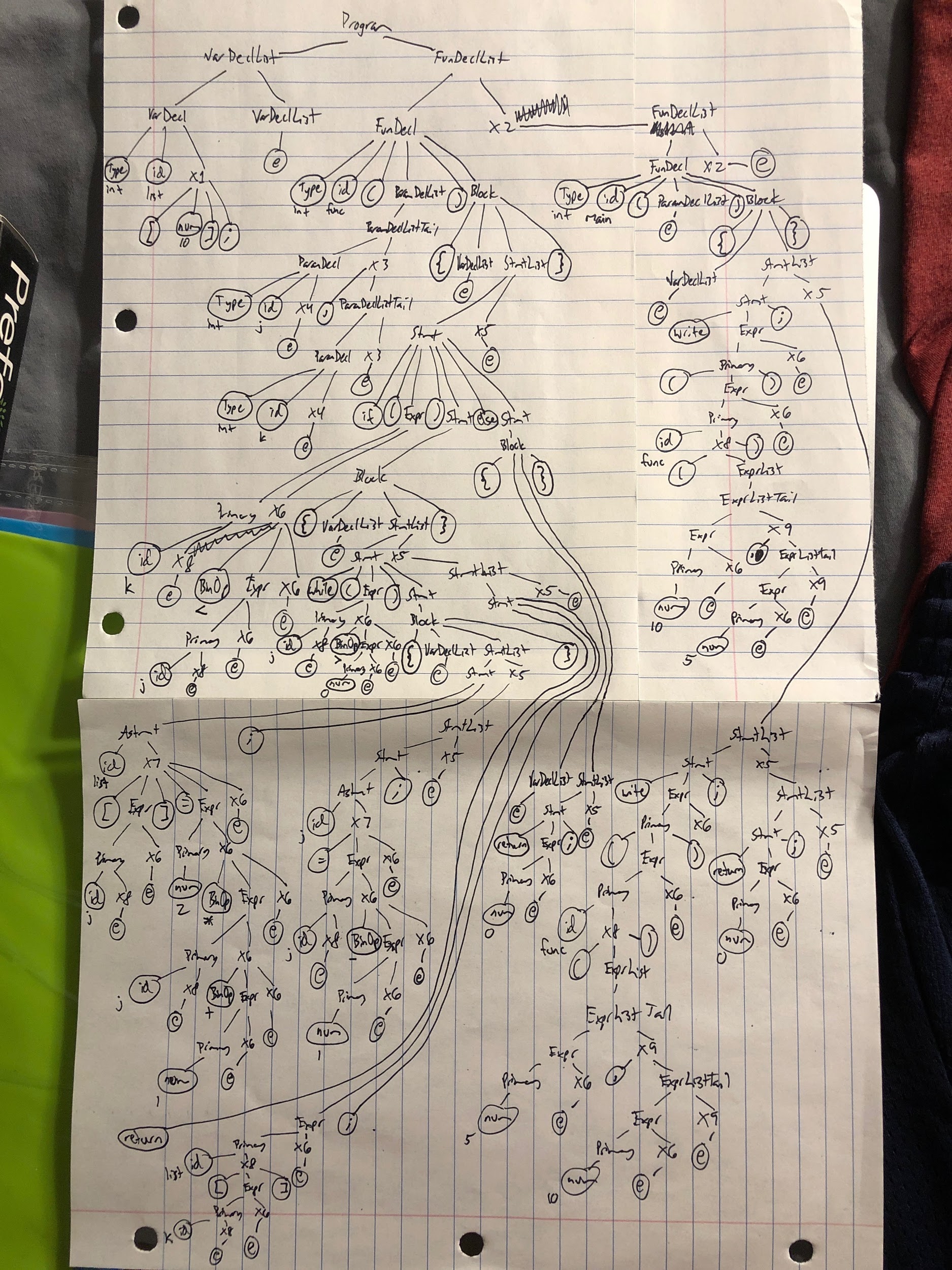
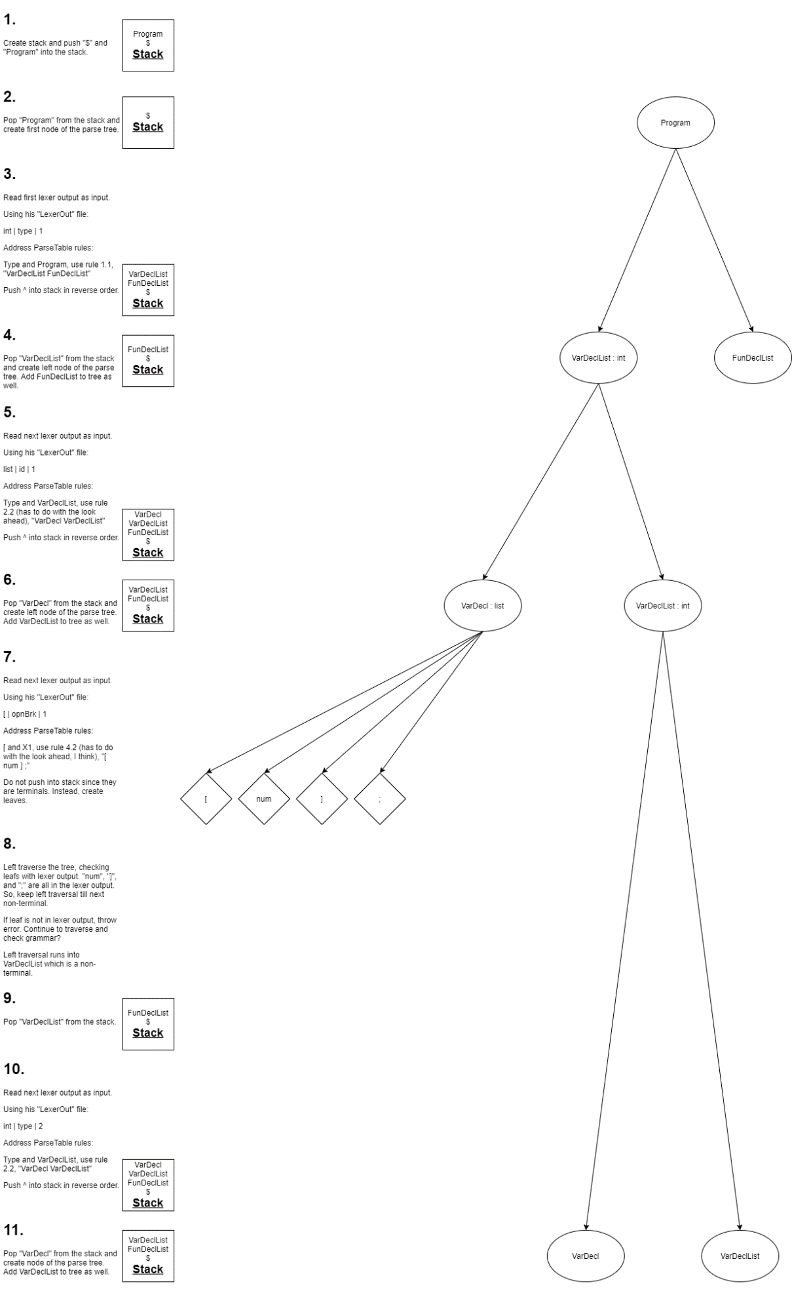
The implementation stage of the syntax parser started with the basic C— grammar. The grammar we were originally provided contained cases of left recursion and left factoring. The first step in the process was to remove left recursion and left factoring from the grammar. Once that was done, a new non-terminal “Astmt” was added in order to provide a discernment between an expression and an assignment statement. Next came the first and follow sets. Our team used the grammar in order to construct a table of the first and follow sets of the grammar and stored the values of the table in excel. Alongside the first and follow sets a parsing table was necessary before a parsing tree could be developed. Using the first and follow set table our team created, the parsing table was constructed and filled out. Then, we created the parsing tree to be implemented later in a GUI flowchart program. This allowed visualization of the tree before implementation. The parsing tree is quite large, so it is attached as a PNG file containing the tree within the zip file. The screenshots below show the first and follow sets table as well as the parsing table, with both excel files in the zip file.

*First and Follow Tables*

**



*Parse Chart*

**

The use of the syntax analyzer is simple. The analyzer takes the tokens from the lexical analyzer and runs them against the rules defined from the grammar. All the user has to do is run the program in an interpreter and the analyzer will construct the parsing tree and output it to the user.

*Purpose*

The syntax analyzer is a key component necessary in order for a compiler to work. The same way the lexical analyzer is necessary for a compiler to work, a syntax analyzer is too. The syntax analyzer’s job is to define the rules of production within the grammar and to check the syntax of the tokens from the lexer. From there the syntax analyzer can determine which rules to use for each group of tokens which will eventually be used as the input to the semantic analyzer, which is the next step in the compilation process. The syntax analyzer will also be able to determine when an incorrect string of tokens is read in, giving the user who wrote the text file a syntax error, allowing them to correct the mistakes they may have made within their code. Syntax analyzers often make use of a parsing tree in order to determine the rule acceptable based on their inputs. The syntax analyzer has a few drawbacks as well, for example, it cannot determine if a token is valid, it cannot determine if a token is declared or initialized before use and it also cannot determine if the operation is appropriate for the given token types.

*Grammar*

PROGRAM ::= VARDECLLIST FUNDECLLIST

VARDECLLIST ::= VARDECL VARDECLLIST |

VARDECL ::= TYPE ID VARDECL'

VARDECL' ::= ; | [ NUM ] ;

FUNDECLLIST ::= FUNDECL FUNDECLLIST'

FUNDECLLIST' ::= FUNDECLLIST |

FUNDECL ::= TYPE ID ( PARAMDECLIST ) BLOCK

PARAMDECLLIST ::= PARAMDECLLISTTAIL |

PARAMDECLLISTTAIL ::= PARAMDECL PARAMDECLLISTTAIL'

PARAMDECLLISTTAIL' ::= , PARAMDECLLISTAIL |

PARAMDECL ::= TYPE ID PARAMDECL'

PARAMDECL' ::= [] |

BLOCK ::= { VARDECLLIST STMTLIST }

TYPE ::= INT | CHAR

STMTLIST ::= STMT STMTLIST'

STMTLIST' ::= STMTLIST |

STMT ::= ; | ASTMT ; | RETURN EXPR ; | READ ID ; | WRITE EXPR ; | WRITELN ; | BREAK ; | IF ( EXPR ) STMT ELSE STMT | BLOCK

ASTMT ::= ID EXPR2

EXPR ::= PRIMARY EXPR1 | UNARYOP EXPR EXPR1

EXPR1 ::= BINOP EXPR EXPR1 |

EXPR2 ::= = EXPR1 | [ EXPR ] = EXPR EXPR1

PRIMARY ::= ID EXPR3 | NUM | ( EXPR )

EXPR3 ::= ( EXPRLIST ) | [EXPR ] |

PRIMARY ::= ID PRIMARY' | NUM | ( EXPR )

PRIMARY' ::= ( EXPRLIST ) | [ EXPR ]

EXPRLIST ::= EXPRLISTTAIL |

EXPRLISTTAIL ::= EXPR EXPRLISTTAIL'

EXPRLISTTAIL' ::= , EXPRLISTTAIL |

UNARYOP ::= - | !

BINOP ::= + | - | \* | / | == | != | < | <= | > | >= | && | ||

*Processing a Stream of Tokens from Lexical Analyzer*

As seen in the loom video provided below, the lexical analyzer passes tokens on to the parser.

*Produced Parse Tree*

As seen in the loom video below, the tokens passed in from the lexical analyzer and processed through the grammar produce a parse tree.

*Identification of Syntax Errors*

As seen in the loom video below, any syntax errors that are produced by incorrect input or computation are reported to the user.

*Loom Video (Replacement for Screenshots)*

<https://www.loom.com/share/97eb5cb0c44645e19fc283e03d504d39>