**A.2.1 Designing the Scanner**

Each token on our compiler represents a terminal symbol of the LL(1) EBNF grammar. The tokens are as follows (revision 23):

DIRECTIVE -> @"<.+?>";

DATATYPE -> @"float|double|u?int|char|u?longlong|u?long|u?short";

INCLUDE -> @"include";

VOID -> @"void";

SWITCH -> @"switch";

CASE -> @"case";

IF -> @"if";

ELSE -> @"else";

FOR -> @"for";

WHILE -> @"while";

DEFAULT -> @"default";

RETURN -> @"return";

SCANF -> @"scanf";

PRINTF -> @"printf";

DO -> @"do";

BREAK -> @"break";

TYPESPEC -> @"@?\""%[dfsc]\""";

REFOPER -> @"&";

IDENTIFIER -> @"[a-zA-Z\_][a-zA-Z0-9\_]\*";

CHARVALUE -> @"([a-zA-Z\_]|[0-9\_])";

ARTOPERATOR -> @"\+|-|%|/|\\*";

CASECHAR -> @"[a-zA-Z\_]\*";

NUMBER -> @"[0-9]+";

QUOT -> @"\'";

DQUOT -> @"\""";

POINTER -> @"\\*";

COMMA -> @",";

EQUALS -> @"=";

SEMICOL -> @";";

COLON -> @":";

SHARP -> @"#";

EOF -> @"^$";

LPAREN -> @"\(";

RPAREN -> @"\)";

LBRACE -> @"\{";

RBRACE -> @"\}";

LARRAY -> @"\[";

RARRAY -> @"\]";

RELOP -> @">=|<=|==|!=|>|<";

LOGOP -> @"&&|\|\|";

INCRE -> @"\+\+|--";

BOOL -> @"true|false";

NOT -> @"!";

NULL -> @"";

STRING -> @"@?\""(\""\""|[^\""])\*\""";

VARIABLEPARAMS -> @"\.\.\.";

[Skip]

WHITESPACE -> @"\s+";

[Skip]

EOL -> @"[\n\r]";

[Skip]

COMMENTLINE -> @"//[^\n]\*\n?";

[Skip]

COMMENTBLOCK -> @"/\\*[^\*]\*\\*+(?:[^/\*][^\*]\*\\*+)\*/";

The terminal symbols of the grammar are defined using terminal production rules and regular expressions. Terminals are defined using .NET’s Regex expression syntax. These terminal definitions will then be used within the Regex by the parser.

**A.2.2 Implementing the Scanner**

The scanner is directly linked to the parser. Thus, a scanner object is needed during the parsing process. The function of the scanner is to get the token type and perform look ahead. Token types are added using regular expressions and are defined by an enum to determine its type. The role of the scanner will be explained further on the parser implementation.

**A.2.3 Designing the Parser**

Each non-terminal symbol on our compiler is represented using production rules of the LL(1) EBNF grammar. The non-terminal symbols are defined on the scanner class (revision 23):

Start -> Header\* GlobalDecl\* EOF;

/\* Pre-processor directives \*/

Header -> SHARP INCLUDE DIRECTIVE;

/\*

\* Global declaration includes all declarations made outside functions.

\* -- Types: Variable, function, and function prototype declaration

\*/

GlobalDecl -> (DATATYPE|VOID) POINTER? IDENTIFIER (Function?

| (Array|DecAssignment)? (COMMA POINTER? IDENTIFIER Array? DecAssignment?)\* SEMICOL);

//Array Declaration

Array -> (LARRAY NUMBER RARRAY)+ ArAssignment?;

ArAssignment -> EQUALS ArContent;

ArContent -> LBRACE STRING (COMMA STRING)\* RBRACE;

//Calling Variable Array

VarArray ->(LARRAY (NUMBER|IDENTIFIER) RARRAY)\*;

//Function? is exluded from LocalDecl

LocalDecl -> DATATYPE POINTER? IDENTIFIER Array? DecAssignment?(COMMA POINTER? IDENTIFIER Array? DecAssignment?)\* SEMICOL;

Assignment -> IDENTIFIER EQUALS Expr SEMICOL;

DecAssignment -> EQUALS (Expr|LBRACE Char RBRACE);

Expr -> Atom (ARTOPERATOR Atom)\*;

Char -> QUOT CHARVALUE QUOT;

Atom -> NUMBER | ((POINTER|REFOPER)? IDENTIFIER VarArray?) | LPAREN Expr RPAREN;

Function -> LPAREN (Parameters|VOID) RPAREN (SEMICOL|CodeBlock);

Parameters -> ((DATATYPE POINTER? IDENTIFIER ParArray?)(COMMA ((DATATYPE POINTER? IDENTIFIER ParArray?)|VARIABLEPARAMS))\*)?;

ParArray -> (LARRAY NUMBER RARRAY)\*;

CodeBlock -> BOOL|(LBRACE Statement\* RBRACE);

Break -> BREAK SEMICOL;

Switch -> SWITCH LPAREN IDENTIFIER VarArray? RPAREN LBRACE SwitchCase RBRACE;

SwitchCase -> ((CASE CaseComp COLON)\* (DEFAULT COLON)? Statement?)\* Statement\*;

CaseComp -> NUMBER|Char;

Statement -> Return|Scanf|Printf|Switch|If|For|While|DoWhile|LocalDecl|Assignment|Break;

If -> IF LPAREN (Condition) RPAREN IfForLoopBlock Else?;

//Condition -> BOOL|(IDENTIFIER RELOP (IDENTIFIER|NUMBER)) (LOGOP Condition)?;

Condition -> BOOL | (CondLogExpr (LOGOP CondLogExpr)\*);

CondLogExpr -> CondExpr (RELOP CondExpr)\*;

CondExpr -> (IDENTIFIER VarArray?|NUMBER|STRING) | NOT? LPAREN Condition RPAREN;

//Condition -> BOOL|(IDENTIFIER RELOP (IDENTIFIER|NUMBER)) (LOGOP Condition)?;

Else -> ELSE IfForLoopBlock;

IfForLoopBlock -> (LBRACE Statement\* RBRACE)|Statement;

For -> FOR LPAREN (ForDeclaration|ForAssignment)? SEMICOL Condition? SEMICOL Increment? RPAREN IfForLoopBlock;

//ForDeclaration and ForAssignment has no SEMICOL at the end

ForDeclaration -> DATATYPE IDENTIFIER DecAssignment?(COMMA IDENTIFIER DecAssignment?)\*;

ForAssignment -> IDENTIFIER EQUALS Expr;

Increment -> IDENTIFIER INCRE;

While -> WHILE LPAREN (Condition) RPAREN WhileLoopBlock;

DoWhile -> DO WhileLoopBlock WHILE LPAREN Condition RPAREN SEMICOL;

//WhileLoopBlock enforces the use of { and } as compared to IfForLoopBlock

WhileLoopBlock -> LBRACE Statement\* RBRACE;

Printf -> PRINTF LPAREN STRING (COMMA (IDENTIFIER VarArray?|NUMBER))\* RPAREN SEMICOL;

Scanf -> SCANF LPAREN TYPESPEC COMMA REFOPER IDENTIFIER RPAREN SEMICOL;

Return -> RETURN Expr SEMICOL;

**A.2.4 Implementing the Parser**

The parser creates a parent parse node while scanning through each token type. The first token type expected would be the start token followed by a look ahead operation. The parent node start then adds a node to the parse node following each production rules as defined earlier. The parse node continues to build up producing a parse tree expecting an EOF token at the end of the tree. If the token is not an EOF or is undetermined then an error will be produced showing the exact location of the error.