parseCompoundStmt

The grammar for compoundStmt is

CompoundStmt → {Declarations Statements}

There can be 0 or more declarations and statements as mentioned by the document. So the logic for the function was to first check for a left brace then loop while there are declarations and statements using the provided parseDecl and parseStmt. The created ASTCompoundStmt node would take those declarations and statements then once there are no more and a right brace is matched, we return.

parseReturnStmt

The idea for the return statement was to check for the return keyword first, then if the next token isn't a semicolon, then there must be an expression and after that must have a semicolon. This order of checking also implicitly covers for the void return case, where the expression will just have the nullptr value when creating the AST node.

parseConstantFactor and parseStringFactor

The logic for these two are pretty much identical because the respective ASTNode constructors both take string values, so I just checked if the current token was a constant or string then used getTokenTxt() to instantiate. The only difference is that the ASTStringExpr constructor also takes the string table mStrings.

parseParenFactor

For ParenFactor or (expr), it was simply just checking for the first left parenthesis then using the provided parseExpr() and then making sure it is enclosed by a right parenthesis with matchToken(), when creating the ASTNode.

parseIncFactor and parseDecFactor

Both are again very similar, first I check for their respective tokens 'Dec' and 'Inc'. Then I check for the identifier and grab it from the symbols table, while consuming the token then making the ASTNodes with the identifier.

parseTerm

The left-recursive grammar of term is

Term → Term * Value

| Term / Value

| Term % Value

| Value

This should be made right recursive to avoid infinite loops with recursive descent parsing. Term generates all strings starting with the non-terminal Value and followed by any number combinations of *, /, % operators and Value(s).

Therefore, the right-recursive grammar of term is

Term → Value Term'

Term' → * Value Term' | / Value Term' | % Value Term' | ε

So, the function just needs to call parseValue then have parseTermPrime handle the rest.

parseTermPrime

Notice how the operators *, /, % all have the same precedence, we should then go by left association. To do this we need the rightmost derivation so that they are evaluated left to right in a post-order traversal of the AST. This is done by placing a recursive call that takes the current operation (LHS and RHS) as the LHS parameter for later possible operations/termPrimes. Here I also had to throw the error in case operand was missing from the RHS.

parseNumExpr/Prime

Uses the same logic as parseTerm/Prime but the right-recursive grammar is

NumExpr → Term NumExpr'

NumExpr' → + Term NumExpr' | - Term NumExpr' | ε

parseRelExpr/Prime

Same as above but right-recursive grammar is

RelExpr → NumExpr RelExpr'

RelExpr' → == NumExpr RelExpr' | != NumExpr RelExpr' | < NumExpr RelExpr'

| > NumExpr RelExpr' | ε

parseAndTerm/Prime

Same as above but right-recursive grammar is

AndTerm ⟶ RelExpr AndTerm'

AndTerm' ⟶ && RelExpr AndTerm' | ε

parseWhileStmt

The idea here was to match the left and right parenthesis if the while keyword matched and in between them; I would parse the condition using the built in parseExpr(). If the condition was null or failed to parse, then I would throw an exception that matched the error tests. Otherwise, I would parse for the loop body then construct the while statement AST if condition and body are present.

parseExprStmt and parseNullStmt

These were simple either parse the expr using the built in parseExpr() and if it exists parse the semicolon to then create the AST or just peekAndConsume the semicolon for the null stmt.

parseIfStmt

First, I checked for the if keyword, then matched the parenthesis around the condition. If the condition did not parse correctly then I would throw an invalid condition message. After that I parsed the then statement and optionally looked for an else case. If there is an else, it gets populated and the if statement AST gets created with all 3. Otherwise, just condition and then statement are incorporated into the if statement AST.

parseAddrOfArrayFactor

For this, first check if there is an addr token. If there isn't a identifier after the addr token, then report an error. Otherwise, you get the identifier, consume the token, match the brackets for the array and parse the expression in between the brackets. If the expression is parsed incorrectly, then we throw an error message saying how the required subscript is missing. After this we can safely create the ASTNode for ArraySub and AddrOfArray.

parseStmt and parseFactor

The order of the if and else if were kind of arbitrary except for the parseNullStmt, which is the last case in parseStmt as a failsafe.