P1 report

Ruoyu Wu

September 19, 2022

Implementation in parseExpr

The challenges in implementing parseExpr lay in transforming left-recursive to right-recursive and making sure that the AST is rightmost derivation. In the following, I will use `parseTerm` and `parseTermPrime` to illustrate how I solve these challenges.

$$Term ::= Term * Value \mid Term/Value \mid Term\%Value \mid Value$$

The original left-recursive grammar above can be first transformed to the right-recursive grammar below:

$$\begin{aligned} Term &::= Value\ TermPrime \\ TermPrime &::= *Value\ TermPrime \mid /Value\ TermPrime \mid \%Value\ TermPrime \mid \epsilon \end{aligned}$$

In `parseTerm`, we first invoke `parseValue` to parse a value, then we check if there is a TermPrime follows the value by invoking `parseTermPrime`. We use to (rightmost) parsed TermPrime as the retVal, to make sure the AST is rightmost derivation.

In `parseTermPrime`, if we find a valid binary operator (i.e., *, %), we expect there is a TermPrime following. We use the binary operator to construct an `ASTBinaryMathOp`, and set its `lhs` as the passed argument. Then, we invoke `parseValue` to parse the rhs. If we cannot parse the rhs, we throw an `OperandMissing` exception. After successfully setting the lhs and rhs, we use `finalizeOp` to complete construction. According to right-recursive grammar shown above, we invoke the `parseTermPrime` to recursively parse the TermPrime. Note that we set the retVal as the TermPrime on the right, to make sure that the generated AST is rightmost derivation.

The implementations of other parse function (e.g., `parseNumExpr`, `parseRelExpr`...) are similar to `parseTerm`, so they are omitted here.

Implementation in parseStmt

In this section, I will use `parseCompoundStmt` to demonstrate how I implement parseStmt, following its grammar shown below:

$$CompoundStmt ::= DeclarationsStatements$$

First, we expect that it begins with a '{' by invoking `peekAndConsume(Token::LBrace)`. If so, we consume it and assume that it is a compound statement and go ahead. Then, we construct a `ASTCompoundStmt` by making a shared pointer. We first keep parsing declarations by using `parseDecl()` and put them into compoundStmt, and then keep parsing statements by using `parseStmt()` and put them into compundStmt. At the end, since there must be a '}' to end the `ASTCompoundStmt`, we use `matchToken(Token::RBrace)` to make sure that it ends with a right brace. If it does not end with a right brace, `matchToken` will throw out an exception. We return the retVal as the created compound statement after we successfully parse everything.

The implementations of other parse function (e.g., `parseReturnStmt`, `parseIfStmt`...) are similar to `parseCompoundStmt`, so they are omitted here.