Parser: Support for Iterative Statements

Prof. Naga Kandasamy
ECE Department, Drexel University

This assignment is due November 7, 2021, by 11:59 pm. You may work on it in a team of up to two people. One submission per group will suffice. Please submit original work.

(10 points) Extend the parser to support the `for` statement.

```
for (int i = 0; i < 100; i = i + 1) {
    // Statements
}

int j = 0;
int n = 10;
for (j = n; j > 0; j = j - 1) {
    // Statements
}

int j = 0;
int n = 100;
for (j = 0; j < n; j = j + 1) {
    // Statements
}
```

Follow the grammar shown below to guide your parser:

CompoundStmt	$\rightarrow$	LCURLY *statementList* RCURLY
statementList	$\rightarrow$	*Statement StatementList*
	$\mid$	*Statement*
Statement	$\rightarrow$	*AssignmentStmt*
	$\mid$	*IfStmt*
	$\mid$	*ForStmt*
	$\mid$	*FunctionCall*
	$\mid$	*ReturnStmt*
	$\mid$	ϵ
ForStmt	$\rightarrow$	`for` LPAREN *AssignmentStmt* SEMI *RelationalExpr* SEMI *AssignmentStmt* RPAREN *CompundStmt*

Complete the *parseForStatement()* method in the provided parser code to achieve the desired functionality. Please read through the *parseIfStatement()* method and structure your solution along those lines.

Use the following class definition for the *forStatement* node provided in the *parser.hh* file.

```
// Class definition for loop
class ForStatement : public Statement
{
    protected:
    std::shared_ptr<Statement> start;               // Initializer statement
    std::shared_ptr<Condition> end;                 // Condition statement
    std::shared_ptr<Statement> step;                // Increment statment
    std::vector<std::shared_ptr<Statement>> block;  // Code block
    // Local variables declared within loop body
    std::unordered_map<std::string, ValueType::Type> block_local_vars;

    public:
    // Constructors
    ForStatement(std::unique_ptr<Statement> &_start,
                 std::unique_ptr<Condition> &_end,
                 std::unique_ptr<Statement> &_step,
                 std::vector<std::shared_ptr<Statement>> &_block,
                 std::unordered_map<std::string,
                 ValueType::Type> &_block_local_vars)
    {
        type = StatementType::FOR_STATEMENT;

        start = std::move(_start);
        end = std::move(_end);
        step = std::move(_step);
        block = std::move(_block);

        block_local_vars = _block_local_vars;
    }

    ForStatement(const ForStatement &_for)
                 : start(std::move(_for.start))
                 , end(std::move(_for.end))
                 , step(std::move(_for.step))
                 , block(std::move(_for.block))
                 , block_local_vars(_for.block_local_vars)
    {}

    auto getStart() { return start.get(); }
    auto getEnd() { return end.get(); }
    auto getStep() { return step.get(); }
    auto &getBlock() { return block; }
    auto getBlockVars() { return &block_local_vars; }

    void printStatement() override;
};
```

Use the *series_sum_using_for.txt* and *factorial_using_for.txt* test files to validate the correctness of your implementation by printing out the generated parse trees.

(10 points) Extend the parser code to support the `while` statement. For example,

```
int n = 10;
int i = 0;

while (i < n) {
    // Statements
}
```

Follow the grammar shown below to guide your parser:

CompoundStmt	$\rightarrow$	LCURLY *statementList* RCURLY
statementList	$\rightarrow$	*Statement StatementList*
	$\mid$	*Statement*
Statement	$\rightarrow$	*AssignmentStmt*
	$\mid$	*IfStmt*
	$\mid$	*ForStmt*
	$\mid$	*WhileStmt*
	$\mid$	*FunctionCall*
	$\mid$	*ReturnStmt*
	$\mid$	ϵ
WhileStmt	$\rightarrow$	`while` LPAREN *RelationalExpr* RPAREN *CompoundStmt*

Complete the following steps to achieve the desired functionality:

- Define a new class called *whileStatement* within *parser.hh* which represents the `while` node. Also, define *parseWhileStatement()* as a new method within the *Parser* class.
- Implement the *parseWhileStatement()* method within *parser.cc*.
- Call the method from *parseStatement()* to parse a while statement.

  ```
  if (cur_token.isTokenWhile()) {
      auto code = parseWhileStatement(cur_func_name);
      codes.push_back(std::move(code));
      return;
  }
  ```

  Note: you may have to update the lexer code to add a new token for the `while` keyword along with the *isTokenWhile()* method.
- Add a *printStatment()* method to print out contents of the `while` node when traversing the AST.

Use the *series_sum_using_while.txt* and *factorial_using_while.txt* test files to validate the correctness of your implementation by printing out the generated parse trees.

Submission Instructions

Submit source files for both the lexer and parser as a single zip file. Source files for the lexer should be within the *lexer* directory and files for the parser within the *parser* directory. Run `make clean` to remove the executable and object files from your project directory. We must be able to build your executable from source and don't require any pre-compiled executables or intermediate object files.