Introduction to Compiler Construction

Assignment 4 (Scanning and Parsing with JavaCC) Discussion

Add support for multiline comments in which all characters between `/*` and `*/` are ignored

Using the rules for single-line comments as a model, define the rules for multiline comments

Testing

```
>_ ~/workspace/j--
$ ant
$ ./bin/j-- -t javacc_frontend/MultilineComment.java
3         : "import" = import
3         : <IDENTIFIER> = java
...
19        : "}" = }
21        : <EOF> = <end of file>
```

Compare your output with the reference output in `javacc_frontend/MultilineComment.tokens`

Add support for the following reserved words in *j--*:

`break` `case` `continue` `default` `do` `double` `for` `long` `switch`

Define these reserved words along with the ones already defined

Testing

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -t javacc_frontend/Keywords.java
1         : "abstract" = abstract
2         : "boolean" = boolean
...
33        : "while" = while
33        : <EOF> =
```

Compare your output with the reference output in `javacc_frontend/Keywords.tokens`

Add support for the following operators:

```
/     %     ?     :     -=    *=
/=    %=    !=    >=    <     ||
```

Define these operators along with the ones already defined

Testing

```
>_ ~/workspace/j--

$ ant
$ ./bin/javaccj-- -t javacc_frontend/Operators1.java
1         : "="  = =
2         : ":"  = :
...
24        : "*=" = *=
24        : <EOF> =
```

Compare your output with the reference output in `javacc_frontend/Operators1.tokens`

Add support for long and double literals

Regular expressions for int, long, and double literals

```
DIGITS          ::= ( "0"..."9" ) { "0"..."9" }
INT_LITERAL     ::= DIGITS
LONG_LITERAL    ::= INT_LITERAL ( "l" | "L" )
EXPONENT        ::= ( "e" | "E" ) [ ( "+" | "-" ) ] DIGITS
SUFFIX          ::= "d" | "D"
DOUBLE_LITERAL ::= DIGITS "." [ DIGITS ] [ EXPONENT ] [ SUFFIX ] // part 1
                 | "." DIGITS [ EXPONENT ] [ SUFFIX ]            // part 2
                 | DIGITS EXPONENT [ SUFFIX ]                    // part 3
                 | DIGITS SUFFIX                                 // part 4
```

Define the regular expressions for `INT_LITERAL` and `LONG_LITERAL` under literals

Testing int and long literals

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -t javacc_frontend/IntLiterals.java
1         : <INT_LITERAL> = 0
2         : <INT_LITERAL> = 9
...
5         : <INT_LITERAL> = 1234567890
6         : <EOF> = <end of file>
$ ./bin/javaccj-- -t javacc_frontend/LongLiterals.java
1         : <LONG_LITERAL> = 1l
2         : <LONG_LITERAL> = 9L
...
6         : <LONG_LITERAL> = 1234567890L
7         : <EOF> = <end of file>
```

Compare your output with the reference output in `javacc_frontend/IntLiterals.tokens` and `javacc_frontend/LongLiterals.tokens`

Define `#EXPONENT`, `#SUFFIX`, and `DOUBLE_LITERAL` (all four parts) under literals

Testing double literals

```
>_ ~/workspace/j--

$ ./bin/javaccj-- -t javacc_frontend/DoubleLiterals1.java
1        : <DOUBLE_LITERAL> = 0.
2        : <DOUBLE_LITERAL> = 1.
...
74       : <DOUBLE_LITERAL> = 123456789.e-135D
75       : <EOF> = <end of file>
$ ./bin/javaccj-- -t javacc_frontend/DoubleLiterals2.java
1        : <DOUBLE_LITERAL> = .0
2        : <DOUBLE_LITERAL> = .1
...
32       : <DOUBLE_LITERAL> = .098765e-135
33       : <EOF> = <end of file>
$ ./bin/javaccj-- -t javacc_frontend/DoubleLiterals3.java
1        : <DOUBLE_LITERAL> = 0e2
2        : <DOUBLE_LITERAL> = 9e9
...
21       : <DOUBLE_LITERAL> = 246e-13D
22       : <EOF> = <end of file>
$ ./bin/javaccj-- -t javacc_frontend/DoubleLiterals4.java
1        : <DOUBLE_LITERAL> = 0d
2        : <DOUBLE_LITERAL> = 0D
...
6        : <DOUBLE_LITERAL> = 0987654321D
7        : <EOF> = <end of file>
```

Compare your output with the reference output in `javacc_frontend/DoubleLiterals*.tokens`

Add support for the following operators

`/` `%` `-=` `*=` `/=` `%=` `!=` `>=` `<` `||` `+` (unary) `++` `--`

Modify `multiplicativeExpression()` to parse the `/` and `%` operators, using `JDivideOp` and `JRemainderOp` in `JBinaryExpression` as the corresponding AST representations

Modify `assignmentExpression()` to parse the `-=`, `*=`, `/=`, and `%=` operators, using `JMinusAssignOp`, `JStarAssignOp`, `JDivAssignOp`, and `JRemAssignOp` in `JAssignment` as the corresponding AST representations

Modify `equalityExpression()` to parse the `!=` operator, using `JNotEqualOp` in `JBooleanBinaryExpression` as the corresponding AST representation

Modify `relationalExpression()` to parse the `>=` and `<` operators, using `JGreaterEqualOp` and `JLessThanOp` in `JComparisonExpression` as the corresponding AST representations

Add `conditionalOrExpression()` to parse the `||` operator, using `JLogicalOrOp` in `JBooleanBinaryExpression` as the corresponding AST representation; modify `conditionalExpression()` in `Parser` to now call `conditionalOrExpression()`

Modify `unaryExpression()` to parse the pre `+` (unary) operator, using `JUnaryPlusOp` and `JUnaryExpression` as the corresponding AST representation

Modify `unaryExpression()` to parse the pre `--` operator, using `JPreDecrementOp` and `JUnaryExpression` as the corresponding AST representation

Modify `postfixExpression()` to parse the post `++` operator, using `JPostIncrementOp` and `JUnaryExpression` as the corresponding AST representation

Testing

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -p javacc_frontend0perators2.java
```

Compare your output with the reference output in `javacc_frontend/Operators2.ast`

Add support for the `long` and `double` basic types

Modify the following methods to support longs and doubles

- `basicType()`
- `literal()` (use `JLiteralLong` and `JLiteralDouble` as the AST representations for a long and double literal respectively)

Testing

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -p javacc_frontend/Factorial.java
$ ./bin/javaccj-- -p javacc_frontend/Quadratic.java
```

Compare your output with the reference output in `javacc_frontend/Factorial.ast` and `javacc_frontend/Quadratic.ast`

Add support for conditional expression (`e ? e1 : e2`)

Define a method `private JExpression conditionalExpression()` to parse a conditional expression

```
conditionalExpression ::= conditionalAndExpression [ QUESTION expression COLON conditionalExpression ]
```

The method should return an object of type `JConditionalExpression`

Modify `assignmentExpression()` to call `conditionalExpression()`

Testing

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -p javacc_frontend/ConditionalExpression.java
```

Compare your output with the reference output in `javacc_frontend/ConditionalExpression.ast`

Add support for a do statement in *j--*

Modify `statement()` to parse a do statement

```
statement ::= ...
            | DO statement WHILE parExpression SEMI
            | ...
```

The method should return an object of type `JDoStatement` for a do statement

Testing

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -p javacc_frontend/DoStatement.java
```

Compare your output with the reference output in `javacc_frontend/DoStatement.ast`

Add support for a for statement

Make the following changes to support a for statement

- Add `ArrayList<JStatement> forInit()` to parse the `forInit` part
  - If looking at a statement expression (use `LOOKAHEAD`), then return a list of statement expressions
  - Otherwise, return a list containing a single `JVariableDeclaration` object encapsulating the variable declarators (see `localVariableDeclarationStatement()` for how to construct that object)
- Add `ArrayList<JStatement> forUpdate()` to parse the `forUpdate` part
- Modify `statement()` to parse a for statement, using `JForStatement` as the AST representation for a for statement

Testing

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -p javacc_frontend/ForStatement.java
```

Compare your output with the reference output in `javacc_frontend/ForStatement.ast`

Add support for a break statement

Modify `statement()` to parse a break statement, using `JBreakStatement` as the AST representation

```
>_ ~/workspace/j--

$ ant
$ ./bin/javaccj-- -p javacc_frontend/BreakStatement.java
```

Compare your output with the reference output in `javacc_frontend/BreakStatement.ast`

Add support for a continue statement

Modify `statement()` to parse a continue statement, using `JContinueStatement` as the AST representation

```
>_ ~/workspace/j--
```

```
$ ant
$ ./bin/javaccj-- -p javacc_frontend/ContinueStatement.java
```

Compare your output with the reference output in `javacc_frontend/ContinueStatement.ast`

Add support for a switch statement

Make the following changes to support a switch statement

- Add `SwitchStatementGroup switchBlockStatementGroup()` to parse the `switchBlockStatementGroup` part
  - After parsing a `switchLabel`, parse zero or more `switchLabel` (use `LOOKAHEAD`)
  - Parse zero or more `blockStatement`
- Add `JExpression switchLabel()` to parse the `switchLabel` part, which must return an expression for a case and `null` for default
- Modify `statement()` to parse a switch statement, using `JSwitchStatement` as the AST representation for a switch statement
  - After parsing `SWITCH parExpression LCURLY`, parse zero or more `switchBlockStatementGroup` until you see an `RCURLY` or `EOF`, and then scan an `RCURLY`

```
>_ ~/workspace/j--
$ ant
$ ./bin/javaccj-- -p javacc_frontend/SwitchStatement.java
```

Compare your output with the reference output in `javacc_frontend/SwitchStatement.ast`