Introduction to Compiler Construction

Assignment 1 (Supporting Simple Operations) Discussion

Problem 1 (Understanding the JVM)

Complete the implementation of `GenIsPrime.java` such that it uses `CLEmitter` to generate `IsPrime.class` for the `IsPrime.java` program given in the assignment writeup

Generating JVM code for the `boolean isPrime(int n)` method

- Add the method header
- Use the following pseudocode for testing and returning *true* if *n* is prime, and *false* otherwise

```
    if n >= 2 jump to A
    return 0 # n is not prime
A:  i = 2
B:  if i > n / i jump to D
    if n % i != 0 jump to C
    return 0 # n is not prime
C:  i = i + 1
    jump to B
D:  return 1 # n is prime
```

Testing

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

```
$ ./bin/clemitter simpleops/GenIsPrime.java
$ java IsPrime 42
false
$ java IsPrime 31
true
```

Problem 2 (Understanding the Marvin Machine)

Implement the function `boolean isPrime(int n)` in `IsPrime.marv` such that it returns 1 if *n* is prime, and 0 otherwise

The function receives the input *n* in register `r0` and must store the return value in register `r13`

Use the following pseudocode to implement the function

```
    set r1 to 2
    if r0 < r1 jump to B # n is not prime
A:  set r2 to r1 * r1
    if r2 > r0 jump to C # n is prime
    set r2 to r0 % r1
    if r2 = 0 jump to B  # n is not prime
    increment r1 by 1
    jump to A
B:  set r13 to 0 and jump to caller
C:  set r13 to 1 and jump to caller
```

Testing

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

```
$ python3 simpleops/marvin.py simpleops/IsPrime.marv
42
0
$ python3 simpleops/marvin.py simpleops/IsPrime.marv
31
1
```

Problem 3 (Arithmetic Operators)

Implement the division `/`, remainder `%`, and unary plus `+` operators in *j--*

Scanning

- Define tokens `DIV` and `REM` in `TokenInfo.TokenKind`
- Scan the tokens in `Scanner.getNextToken()`

Parsing

- Modify `multiplicativeExpression()` in `Parser` to parse the `/` and `%` operators

  ```
  multiplicativeExpression ::= unaryExpression { ( DIV | REM | STAR ) unaryExpression }
  ```

  The method should return an object of type `JDivideOp` for `/` and `JRemainderOp` for `%`

- Modify `unaryExpression()` in `Parser` to parse the unary `+` operator

  ```
  unaryExpression ::= ...
                    | ( MINUS | PLUS ) unaryExpression
                    | ...
  ```

  The method should return an object of type `JUnaryPlusOp` for unary `+`

Problem 3 (Arithmetic Operators)

Semantic analysis

- In `JDivideOp.analyze()` and `JRemainderOp.analyze()`
  - Analyze `lhs` and `rhs` and make sure they are both ints
  - Set the type of the expression to int
- In `JUnaryPlusOp.analyze()`
  - Analyze `operand` and make sure it is an int
  - Set the type of the expression to int

Code generation

- In `JDivideOp.codegen()` and `JRemainderOp.codegen()`
  - Generate code for `lhs` and `rhs`
  - Add an appropriate instruction for the operation
- In `JUnaryPlusOp.codegen()`
  - Generate code for `operand`

Problem 3 (Arithmetic Operators)

Testing

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

```
$ ant
$ ./bin/j-- simpleops/Division.java
$ java Division 60 13
4
$ ./bin/j-- simpleops/Remainder.java
$ java Remainder 60 13
8
$ ./bin/j-- simpleops/UnaryPlus.java
$ java UnaryPlus 60
60
```

Problem 4 (Conditional Expression)

Add support for conditional expression (`e ? e1 : e2`) in *j--*

Scanning

- Define tokens `COLON` and `QUESTION` in `TokenInfo.TokenKind`
- Scan the two tokens in `Scanner.getNextToken()`

Parsing

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

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

  The method should return an object of type `JConditionalExpression`

- Modify `assignmentExpression()` in `Parser` to call `conditionalExpression()`

  ```
  assignmentExpression ::= conditionalExpression [ ( ASSIGN | PLUS_ASSIGN ) assignmentExpression ]
  ```

Problem 4 (Conditional Expression)

Semantic analysis

- In `JConditionalExpression.analyze()`
  - Analyze `condition` and make sure it is a boolean
  - Analyze `thenPart` and `elsePart` and make sure they have the same type
  - Set the type of the conditional expression to that of `thenPart` (or `elsePart`)

Code generation

- In `JConditionalExpression.codegen()`
  - Create labels `elseLabel` and `endLabel`
  - Use the 3-argument `codegen()` method to generate code for `condition`, branching to `elseLabel` when `condition` is `false`
  - Generate code for `thenPart`
  - Add an instruction to jump to `endLabel`
  - Emit label `elseLabel`
  - Generate code for `elsePart`
  - Emit label `endLabel`

Problem 4 (Conditional Expression)

Testing

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

$ ant
$ ./bin/j-- simpleops/ConditionalExpression.java
$ java ConditionalExpression
Tails
$ java ConditionalExpression
Tails
$ java ConditionalExpression
Heads
```

Problem 5 (Do Statement)

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

Scanning

- Define a token `DO` in the `TokenInfo.TokenKind`
- Add the token to the table of reserved words in `Scanner`

Parsing

- Modify `statement()` in `Parser` to parse a do statement

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

The method should return an object of type `JDoStatement`

Problem 5 (Do Statement)

Semantic analysis

- In `JDoStatement.analyze()`
  - Analyze `body`
  - Analyze `condition` and make sure it is a boolean

Code generation

- In `JDoStatement.codegen()`
  - Create label `topLabel`
  - Add label `topLabel`
  - Generate code for `body`
  - Use the 3-argument `codegen()` method to generate code for `condition`, branching to `topLabel` when `condition` is `true`

Testing

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

$ ant
$ ./bin/j-- simpleops/DoStatement.java
$ java DoStatement 100
5050
```