Contents

1 **Compilation (Chapter 1)** 2

2 **Lexical Analysis (Chapter 2)** 4

3 **Parsing (Chapter 3)** 7

4 **Type Checking (Chapter 4)** 13

5 **JVM Code Generation (Chapter 5)** 14

6 **Translating JVM Code to MIPS Code (Chapter 6)** 17

7 **Register Allocation (Chapter 7)** 20

1 Compilation (Chapter 1)

Problems

Problem 1. What are the changes that you will need to make in the *j*-- code tree in order to support `@` as the remainder operator on primitive integers? For example, `17 @ 5 = 2`.

Problem 2. Write down the following class names in internal form:

- `java.util.ArrayList`
- `jminusminus.Parser`
- `Employee`

Problem 3. Write down the JVM type descriptor for each of the following field/constructor/method declarations:

- `private int N;`
- `private String s;`
- `public static final double PI = 3.141592653589793;`
- `public Employee(String name) ...`
- `public Coordinates(float latitude, float longitude) ...`
- `public Object get(String key) ...`
- `public void put(String key, Object o) ...`
- `public static int[] sort(int[] a, boolean ascending) ...`
- `public int[][] transpose(int[][] matrix) ...`

Problem 4. Write a program `GenSquare.java` that produces, using `CLEmitter`, a `Square.class` program, which accepts an integer n as command-line argument and prints the square of that number as output.

Solutions

Solution 1.

`$j/j--/lexicalgrammar`

```
...
REM ::= "@"
...
```

`$j/j--/grammar`

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

`$j/j--/src/jminusminus/TokenInfo.java`

```
enum TokenKind {
    ...
    REM("@"),
    ...
}
```

`$j/j--/src/jminusminus/Scanner.java`

```
...
        case '@':
            nextCh();
            return new TokenInfo(REM, line);
...
```

$j/j--/src/jminusminus/Parser.java

```
    private JExpression multiplicativeExpression() {
        int line = scanner.token().line();
        boolean more = true;
        JExpression lhs = unaryExpression();
        while (more) {
            if (have(STAR)) {
                lhs = new JMultiplyOp(line, lhs, unaryExpression());
            } else if (have(REM)) {
                lhs = new JRemainderOp(line, lhs, unaryExpression());
            } else {
                more = false;
            }
        }
        return lhs;
    }
```

$j/j--/src/jminusminus/JBinaryExpression.java

```
class JRemainderOp extends JBinaryExpression {
    public JRemainderOp(int line, JExpression lhs, JExpression rhs) {
        super(line, "%", lhs, rhs);
    }

    public JExpression analyze(Context context) {
        lhs = (JExpression) lhs.analyze(context);
        rhs = (JExpression) rhs.analyze(context);
        lhs.type().mustMatchExpected(line(), Type.INT);
        rhs.type().mustMatchExpected(line(), Type.INT);
        type = Type.INT;
        return this;
    }

    public void codegen(CLEmitter output) {
        lhs.codegen(output);
        rhs.codegen(output);
        output.addNoArgInstruction(IREM);
    }
}
```


Solution 2.

- java/util/ArrayList
- jminusminus/Parser
- Employee

Solution 3.

- I
- Ljava/lang/String;
- D
- (Ljava/lang/String;)V
- (FF)V
- (Ljava/lang/String;)Ljava/lang/Object;
- (Ljava/lang/String;Ljava/lang/Object;)V
- ([IZ)[I
- ([[I)[[I

Solution 4.

```
import jminusminus.CLEmitter;
import static jminusminus.CLConstants.*;
import java.util.ArrayList;

public class GenSquare {
    public static void main(String[] args) {
        CLEmitter e = new CLEmitter(true);
        ArrayList<String> accessFlags = new ArrayList<String>();

        accessFlags.add("public");
        e.addClass(accessFlags, "Square", "java/lang/Object", null, true);

        accessFlags.clear();
        accessFlags.add("public");
        accessFlags.add("static");
        e.addMethod(accessFlags, "main", "([Ljava/lang/String;)V", null, true);
        e.addNoArgInstruction(ALOAD_0);
        e.addNoArgInstruction(ICONST_0);
        e.addNoArgInstruction(AALOAD);
        e.addMemberAccessInstruction(INVOKESTATIC, "java/lang/Integer",
              "parseInt", "(Ljava/lang/String;)I");
        e.addNoArgInstruction(ISTORE_1);
        e.addMemberAccessInstruction(GETSTATIC, "java/lang/System", "out",
              "Ljava/io/PrintStream;");
        e.addNoArgInstruction(ILOAD_1);
        e.addNoArgInstruction(ILOAD_1);
        e.addNoArgInstruction(IMUL);
        e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/io/PrintStream",
              "println", "(I)V");
        e.addNoArgInstruction(RETURN);

        e.write();
    }
}
```


2 Lexical Analysis (Chapter 2)

PROBLEMS

Problem 1. Consider the following *j*-- program:

```
import java.lang.System;

public class Greetings {
    public static void main(String[] args) {
        System.out.println("Hi " + args[0] + "!");
    }
}
```

List the tokens in the program, along with their line numbers and their images.

Problem 2. Consider a language over the alphabet $\{a, b\}$ that consists of strings ending in ab.

a. Provide a regular expression for the language.

b. Draw a state transition diagram for the language.

Problem 3. Consider the regular expression $(a|b)*$ over the alphabet $\{a, b\}$.

a. Describe the language implied by the regular expression.

b. Use Thompson's construction to derive a non-deterministic finite state automaton (NFA) recognizing the same language.

c. Use powerset construction to derive an equivalent deterministic finite state automaton (DFA).

d. Use the partitioning method to derive an equivalent minimal DFA.

Problem 4. Suppose we wish to add support for the do-while statement in *j*--.

statement ::= block
| `if` parExpression statement [`else` statement]
| `while` parExpression statement
| `do` statement `while` parExpression ;
| `return` [expression] ;
| ;
| statementExpression ;

What changes will you need to make in the hand-written and JavaCC lexical analyzers in the *j*-- code tree in order to support the do-while statement?

Solutions

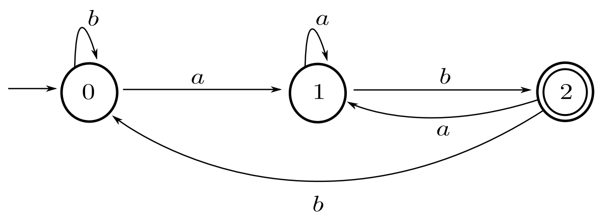
Solution 1.

```
1         : import = import
1         : <IDENTIFIER> = java
1         : . = .
1         : <IDENTIFIER> = lang
1         : . = .
1         : <IDENTIFIER> = System
1         : ; = ;
3         : public = public
3         : class = class
3         : <IDENTIFIER> = Greetings
3         : { = {
4         : public = public
4         : static = static
4         : void = void
4         : <IDENTIFIER> = main
4         : ( = (
4         : <IDENTIFIER> = String
4         : [ = [
4         : ] = ]
4         : <IDENTIFIER> = args
4         : ) = )
4         : { = {
5         : <IDENTIFIER> = System
5         : . = .
5         : <IDENTIFIER> = out
5         : . = .
5         : <IDENTIFIER> = println
5         : ( = (
5         : <STRING_LITERAL> = "Hi "
5         : + = +
5         : <IDENTIFIER> = args
5         : [ = [
5         : <INT_LITERAL> = 0
5         : ] = ]
5         : + = +
5         : <STRING_LITERAL> = "!"
5         : ) = )
5         : ; = ;
6         : } = }
7         : } = }
8         : <EOF> = <EOF>
```


Solution 2.

a. $(a|b) * ab$

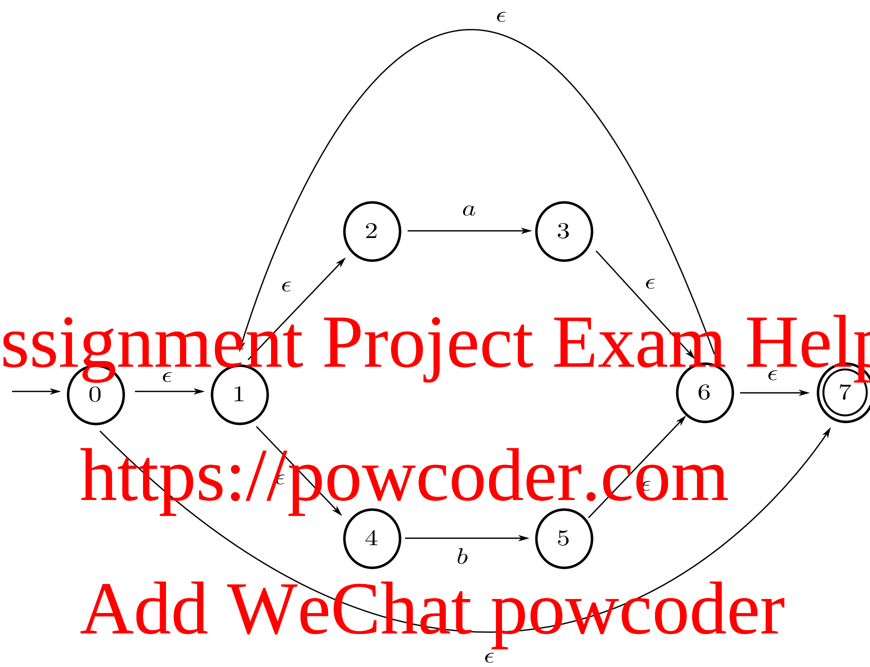
b. State transition diagram for the language:



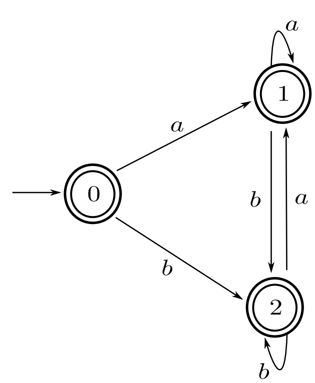
Solution 3.

a. The language consists of strings with any number of a's or b's.

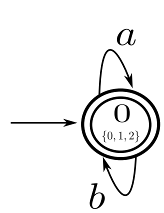
b. An NFA for the language:



c. A DFA for the language:



d. A minimal DFA for the language:



Solution 4.

`$j/j--/lexicalgrammar`

```
...
DO ::= "do"
...
```

`$j/j--/src/jminusminus/TokenInfo.java`

```
enum TokenKind {
    ...
    DO("do"),
    ...
}
```

`$j/j--/src/jminusminus/Scanner.java`

```
...
reserved.put(DO.image(), DO);
...
```

`$j/j--/src/jminusminus/j--.jj`

```
TOKEN:
{
...
| < CHAR: "do" >
...
}
```


3 Parsing (Chapter 3)

PROBLEMS

Problem 1. Consider the following grammar:

$$S ::= (L) \mid \texttt{a}$$
$$L ::= L\ S \mid \epsilon$$

a. What language does this grammar describe?

b. Show the derivation for the sentence `( a ( ) ( a ( a ) ) )`.

c. Derive an equivalent LL(1) grammar.

Problem 2. Show that the following grammar is ambiguous:

$$S ::= \texttt{a}\ S\ \texttt{b}\ S \mid \texttt{b}\ S\ \texttt{a}\ S \mid \epsilon$$

Problem 3. Consider the following context-free grammar:

$$S ::= A\ \texttt{a}$$
$$A ::= \texttt{b}\ \texttt{d}\ B \mid \texttt{e}\ B$$
$$B ::= \texttt{c}\ A \mid \texttt{d}\ B \mid \epsilon$$

a. Compute first and follow for S, A and B.

b. Construct an LL(1) parsing table for this grammar.

c. Show the steps in parsing `bdcea`.

Problem 4. Consider the following grammar:

$S ::= L \mathtt{=} R$
$S ::= R$
$L ::= \mathtt{*} R$
$L ::= \mathtt{i}$
$R ::= L$

a. Construct the canonical LR(1) collection.

b. Construct the Action and Goto tables.

c. Show the steps in the parse for `*i=i`.

Problem 5. Suppose we wish to add support for the do-while statement in *j--*.

statement ::= block
| `if` parExpression statement [`else` statement]
| `while` parExpression statement
| `do` statement `while` parExpression `;`
| `return` [expression] `;`
| `;`
| statementExpression `;`

What changes will you need to make in the hand-written and JavaCC parsers in the *j--* code tree in order to support the do-while statement?

Solutions

Solution 1. a. The grammar describes parenthesized strings, such as `()`, `(a)`, `(a(a))`, and so on.

b. A derivation for the sentence `( a ( ) ( a ( a ) ) )`:

$S \rightarrow \mathtt{(}\ L\ \mathtt{)}$
$\rightarrow \mathtt{(}\ LS\ \mathtt{)}$
$\rightarrow \mathtt{(}\ LSS\ \mathtt{)}$
$\rightarrow \mathtt{(}\ LSSS\ \mathtt{)}$
$\rightarrow \mathtt{(}\ SSS\ \mathtt{)}$
$\rightarrow \mathtt{(\ a}\ SS\ \mathtt{)}$
$\rightarrow \mathtt{(\ a\ (\ )}\ S\ \mathtt{)}$
$\rightarrow \mathtt{(\ a\ (\ )\ (}\ L\ \mathtt{)\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (}\ LS\ \mathtt{)\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (}\ LSS\ \mathtt{)\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (}\ SS\ \mathtt{)\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (\ a}\ S\ \mathtt{)\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (\ a\ (}\ L\ \mathtt{)\ )\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (\ a\ (}\ LS\ \mathtt{)\ )\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (\ a\ (}\ S\ \mathtt{)\ )\ )}$
$\rightarrow \mathtt{(\ a\ (\ )\ (\ a\ (\ a\ )\ )\ )}$

c. An equivalent LL(1) grammar:

$S \rightarrow \mathtt{(}\ L\ \mathtt{)}$
$S \rightarrow \mathtt{a}$
$L \rightarrow X'$
$X' \rightarrow SX'$
$X' \rightarrow \epsilon$

Solution 2. a. Two leftmost derivations (shown below) are possible for the sentence `abab`, and hence the grammar is ambiguous.

$$\begin{aligned} S &\to \texttt{a}S\texttt{b}S \\ &\to \texttt{ab}S\texttt{a}S\texttt{b}S \\ &\to \texttt{aba}S\texttt{b}S \\ &\to \texttt{abab}S \\ &\to \texttt{abab} \end{aligned}$$

$$\begin{aligned} S &\to \texttt{a}S\texttt{b}S \\ &\to \texttt{ab}S \\ &\to \texttt{aba}S\texttt{b}S \\ &\to \texttt{abab}S \\ &\to \texttt{abab} \end{aligned}$$

Solution 3. The grammar with numbered rules:

1. $S \to A\texttt{a}$
2. $A \to \texttt{bd}B$
3. $A \to \texttt{e}B$
4. $B \to \texttt{c}A$
5. $B \to \texttt{d}B$
6. $B \to \epsilon$

a. First and follow sets:

first(S) = {`b`, `e`}
first(A) = {`b`, `e`}
first(B) = {`c`, `d`, ϵ}

follow(S) = {`#`}
follow(A) = {`a`}
follow(B) = {`a`}

b. LL(1) parse table:

	a	b	c	d	e	#
S		1			1	
A		2			3	
B	6		4	5		

c. The steps in parsing `bcdea`:

Stack	Input	Output
#S	bdcea#	1
#aA	bdcea#	2
#aBdb	bdcea#	
#aBd	dcea#	
#aB	cea#	4
#aAc	cea#	
#aA	ea#	3
#aBe	ea#	
#aB	a#	6
#a	a#	
#	#	Accept!

Solution 4. Augmented grammar:

0. S' $\to S$
1. $S \to L \texttt{ = } R$
2. $S \to R$
3. $L \to * \ R$
4. $L \to \texttt{i}$
5. $R \to L$

a. The canonical LR(1) collection:

$s_0 = \{[S' \to \cdot S, \texttt{\#}], [S \to \cdot L\texttt{=}R, \texttt{\#}], [S \to \cdot R, \texttt{\#}], [L \to \cdot *R, \texttt{=/\#}], [L \to \cdot \texttt{i}, \texttt{=/\#}], [R \to \cdot L, \texttt{\#}]\}$

$\text{goto}(s_0, S) = \{[S' \to S\cdot, \texttt{\#}]\} = s_1$

$\text{goto}(s_0, L) = \{[S \to L \cdot \texttt{=}R, \texttt{\#}], [R \to L\cdot, \texttt{\#}]\} = s_2$

$\text{goto}(s_0, R) = \{[S \to R\cdot, \texttt{\#}]\} = s_3$

$\text{goto}(s_0, *) = \{[L \to * \cdot R, \texttt{=/\#}], [R \to \cdot L, \texttt{=/\#}], [L \to \cdot *R, \texttt{=/\#}], [L \to \cdot \texttt{i}, \texttt{=/\#}]\} = s_4$

$\text{goto}(s_0, \texttt{i}) = \{[L \to \texttt{i}\cdot, \texttt{=/\#}]\} = s_5$

$\text{goto}(s_2, \texttt{=}) = \{[S \to L\texttt{=} \cdot R, \texttt{\#}], [R \to \cdot L, \texttt{\#}], [L \to \cdot *R, \texttt{\#}], [L \to \cdot \texttt{i}, \texttt{\#}], \} = s_6$

$\text{goto}(s_4, L) = \{[R \to L\cdot, \texttt{=/\#}]\} = s_7$

$\text{goto}(s_4, R) = \{[L \to *R\cdot, \texttt{=/\#}]\} = s_8$

$\text{goto}(s_4, *) = \{[L \to * \cdot R, \texttt{=/\#}], [R \to \cdot L, \texttt{=/\#}], [L \to \cdot *R, \texttt{=/\#}], [L \to \cdot \texttt{i}, \texttt{=/\#}]\} = s_4$

$\text{goto}(s_4, \texttt{i}) = \{[L \to \texttt{i}\cdot, \texttt{=/\#}]\} = s_5$

$\text{goto}(s_6, L) = \{[R \to L\cdot, \texttt{\#}]\} = s_9$

$\text{goto}(s_6, R) = [S \to L\texttt{=}R\cdot, \texttt{\#}]\} = s_{10}$

$\text{goto}(s_6, *) = \{[L \to * \cdot R, \texttt{\#}], [R \to \cdot L, \texttt{\#}], [L \to \cdot *R, \texttt{\#}], [L \to \cdot \texttt{i}, \texttt{\#}]\} = s_{11}$

$\text{goto}(s_6, \texttt{i}) = \{[L \to \texttt{i}\cdot, \texttt{\#}]\} = s_{12}$

$\text{goto}(s_{11}, L) = \{[R \to L\cdot, \texttt{\#}]\} = s_9$

$\text{goto}(s_{11}, R) = [L \to *R\cdot, \texttt{\#}]\} = s_{13}$

$\text{goto}(s_{11}, *) = \{[L \to * \cdot R, \texttt{\#}], [R \to \cdot L, \texttt{\#}], [L \to \cdot *R, \texttt{\#}], [L \to \cdot \texttt{i}, \texttt{\#}]\} = s_{11}$

$\text{goto}(s_{11}, \texttt{i}) = \{[L \to \texttt{i}\cdot, \texttt{\#}]\} = s_{12}$

b. The Action and Goto tables:

	=	*	i	#	S	L	R
0		$s4$	$s5$		1	2	3
1				Accept			
2	$s6$			$r5$			
3				$r2$			
4		$s4$	$s5$			7	8
5	$r4$			$r4$			
6		$s11$	$s12$			9	10
7	$r5$			$r5$			
8	$r3$			$r3$			
9				$r5$			
10				$r1$			
11		$s11$	$s12$			9	13
12				$r4$			
13				$r3$			

c. The steps in parsing `*i=i`:

Stack	Input	Output
0	`*i=i#`	$s4$
0*4	`i=i#`	$s5$
0*4i5	`=i#`	$r4$
0*4L7	`=i#`	$r5$
0*4R8	`=i#`	$r3$
0L2	`=i#`	$s6$
0L2=6	`i#`	$s12$
0L2=6i12	`#`	$r4$
0L2=6L9	`#`	$r5$
0L2=6R10	`#`	$r1$
0S1	`#`	Accept!

Solution 5. `$j/j--/grammar`

```
statement ::= block
            | IF parExpression statement [ELSE statement]
            | WHILE parExpression statement
            | DO statement WHILE parExpression SEMI
            | RETURN [expression] SEMI
            | SEMI
            | statementExpression SEMI
```


`$j/j--/src/jminusminus/Parser.java`

```
    private JStatement statement() {
        int line = scanner.token().line();
        if (see(LCURLY)) {
            return block();
        } else if (have(IF)) {
            JExpression test = parExpression();
            JStatement consequent = statement();
            JStatement alternate = have(ELSE) ? statement() : null;
            return new JIfStatement(line, test, consequent, alternate);
        } else if (have(WHILE)) {
            JExpression test = parExpression();
            JStatement statement = statement();
            return new JWhileStatement(line, test, statement);
        } else if (have(DO)) {
            JStatement statement = statement();
            mustBe(WHILE);
            JExpression test = parExpression();
            mustBe(SEMI);
            return new JDoWhileStatement(line, statement, test);
        } else if (have(RETURN)) {
            if (have(SEMI)) {
                return new JReturnStatement(line, null);
            } else {
                JExpression expr = expression();
                mustBe(SEMI);
                return new JReturnStatement(line, expr);
            }
        } else if (have(SEMI)) {
            return new JEmptyStatement(line);
        } else { // Must be a statementExpression
            JStatement statement = statementExpression();
            mustBe(SEMI);
            return statement;
        }
    }
```

`$j/j--/src/jminusminus/JDoWhileStatement.java`

```
package jminusminus;

import static jminusminus.CLConstants.*;

class JDoWhileStatement extends JStatement {
    private JStatement body;
    private JExpression condition;

    public JDoWhileStatement(int line, JStatement body, JExpression condition) {
        super(line);
        this.body = body;
        this.condition = condition;
    }

    public JWhileStatement analyze(Context context) { return this; }

    public void codegen(CLEmitter output) { }

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("<JDoWhileStatement line=\"%d\">\n", line());
        p.indentRight();
        p.printf("<Body>\n");
        p.indentRight();
        body.writeToStdOut(p);
        p.indentLeft();
        p.printf("</Body>\n");
        p.printf("<TestExpression>\n");
        p.indentRight();
        condition.writeToStdOut(p);
        p.indentLeft();
        p.printf("</TestExpression>\n");
        p.indentLeft();
        p.printf("</JDoWhileStatement>\n");
    }
}
```

$j/j--/src/jminusminus/j--.jj

```
private JStatement statement(): {
    int line = 0;
    JStatement statement = null;
    JExpression test = null;
    JStatement consequent = null;
    JStatement alternate = null;
    JStatement body = null;
    JExpression expr = null;
}
{
    try {
        statement = block() |
        <IF> { line = token.beginLine; }
        test = parExpression()
        consequent = statement()

        // Even without the lookahead below, which is added to
        // suppress JavaCC warnings, dangling if-else problem is
        // resolved by binding the alternate to the closest
        // consequent.
        [
            LOOKAHEAD( <ELSE> )
            <ELSE> alternate = statement()
        ]
        { statement =
            new JIfStatement( line, test, consequent, alternate ); } |
        <WHILE> { line = token.beginLine; }
        test = parExpression()
        body = statement()
        { statement = new JWhileStatement( line, test, body ); } |
        <DO> { line = token.beginLine; }
```

```
        body = statement()
        <WHILE>
        test = parExpression()
        <SEMI>
        { statement = new JDoWhileStatement( line, body, test ); } |
        <RETURN> { line = token.beginLine; }
        [
            expr = expression()
        ]
        <SEMI>
        { statement = new JReturnStatement( line, expr ); } |
        <SEMI>
        { statement = new JEmptyStatement( line ); } |
        // Must be a statementExpression
        statement = statementExpression()
        <SEMI>
    }
    catch ( ParseException e ) {
        recoverFromError( new int[] { SEMI, EOF }, e );
    }
    { return statement; }
}
```

4 Type Checking (Chapter 4)

Problems

Problem 1. Consider the following *j*-- program:


```
package pass;

import java.lang.Integer;
import java.lang.System;

public class Sum {
    private static String MSG = "SUM = ";
    private int n;

    public Sum(int n) {
        this.n = n;
    }

    public int compute() {
        int sum = 0, i = n;
        while (i > 0) {
            sum += i--;
        }
        return sum;
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        Sum sum = new Sum(n);
        System.out.println(MSG + sum.compute());
    }
}
```

a. How does pre-analysis (`JCompilationUnit.preAnalyze()`) of the program work?

b. How does analysis (`JCompilationUnit.analyze()`) of the program work?

c. How are the declarations of the local variables `sum` and `i` handled in the `compute()` method?

d. How are offsets assigned to the parameters/variables in the program's constructor and the two methods?

e. How is the simple variable `n` resolved in the `main()` method?

f. How is the field selection `MSG` resolved in the `main()` method?

g. How are the message expressions `System.out.println(...)` and `sum.compute()` resolved in the `main()` method?

h. How is argument to `System.out.println()` analyzed in the `main()` method?

Problem 2. When can you cast an expression of type `Type1` to another type `Type2`?

Problem 3. Consider the following *j*-- program:

```java
public class Mystery {
    public int f(int x) {
        int y = x * x;
        return z;
    }
}
```

Is the program syntactically/semantically correct? If not, why and how does *j*-- figure it out?

Problem 4. How would you do semantic analysis for the do-while statement, ie, implement `analyze()` in `JDoWhileStatement.java`?

Solutions

Solution 1.

a. See section 4.4 of our text.

b. See section 4.5 of our text.

c. See section 4.5.2 of our text.

d. See section 4.5.2 of our text.

e. See section 4.5.3 of our text.

f. See section 4.5.4 of our text.

g. See section 4.5.4 of our text.

h. See section 4.5.5 of our text.

Solution 2. See section 4.5.6 of our text.

Solution 3. The program is syntactically correct, but semantically wrong since the variable `z` is not declared before use. During analysis of the return statement, the simple variable `z` is looked up in the chain of contexts, starting at the local context. The lookup is unsuccessful, and an error is reported that the variable has not been declared.

Solution 4.

```java
public JDoWhileStatement analyze(Context context) {
     body = (JStatement) body.analyze(context);
     condition = condition.analyze(context);
     condition.type().mustMatchExpected(line(), Type.BOOLEAN);
     return this;
}
```

5 JVM Code Generation (Chapter 5)

PROBLEMS

Problem 1. Reconsider the *j*-- program `Sum` from above:

```java
package pass;

import java.lang.Integer;
import java.lang.System;

public class Sum {
    private static String MSG = "SUM = ";
    private int n;

    public Sum(int n) {
        this.n = n;
    }

    public int compute() {
        int sum = 0, i = n;
        while (i > 0) {
            sum += i--;
        }
        return sum;
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        Sum sum = new Sum(n);
        System.out.println(MSG + sum.compute());
    }
}
```


How does JVM bytecode generation (`JCompilationUnit.codegen()`) for the program work?

Problem 2. Suppose `lhs` and `rhs` are boolean expressions. How does *j*-- generate code for the following statements?

a.
```java
boolean x = lhs && rhs;
```

b.
```java
if (lhs && rhs) {
    then_statement
} else {
    else_statement
}
```

c.
```java
while (lhs && rhs) {
    statement
}
```

Problem 3. Suppose `x` is an object and `y` is an integer field within.

a. What is the JVM bytecode generated for the following statement? How does the runtime stack evolve as the instructions are executed?

```java
++x.y;
```

b. If `z` is also an integer, what is the JVM bytecode generated for the following statement? How does the runtime stack evolve as the instructions are executed?

```java
z = ++x.y;
```

Problem 4. How is code generated for the expression `"The first perfect number is " + 6`?

Problem 5. How is code generated for casts?

Problem 6. How would you generate JVM bytecode for the do-while statement, ie, implement `codegen()` in `JDoWhileStatement.java`?

Solutions

Solution 1. Consult sections 5.2 – 5.6 of our text.

Solution 2.

a.
```
        lhs code
        branch to Target on false
        rhs code
        branch to Target on false
        push 1 on stack
        goto End
Target: push 0 on stack
End:    ...
```

b.
```
        lhs code
        branch to Target on false
        rhs code
        branch to Target on false
        then_statement code
        goto End
Target: else_statement code
End:    ...
```

c.
```
Test:   lhs code
        branch to Target on false
        rhs code
        branch to Target on false
        body code
        goto Test
Target: ...
```


Solution 3. We use table on slide 26 from the JVM Code Generation chapter.

a.
```
Bytecode:

aload x'
dup
getfield y
iconst_1
iadd
putfield y

Runtime stack (right to left is top to bottom):

| x |
| x | x
| x | y
| x | y | 1
| x | y+1
|
```

b.
```
Bytecode:

aload x
dup
getfield y
iconst_1
iadd
dup_x1
putfield y

Runtime stack (right to left is top to bottom):
```

```
| x |
| x | x
| x | y
| x | y | 1
| x | y+1
| y+1 | x | y+1
| y+1
```

Solution 4. Since the left-hand-side expression of `+` is a string, the operation denotes string concatenation, and is represented in the AST as a `JStringConcatenationOp` object. The `codegen()` method therein does the following:

1. Creates an empty string buffer, ie, a `StringBuffer` object, and initializes it.
2. Appends the string `"The first perfect number is "` to the buffer using `StringBuffer`'s `append(String x)` method.
3. Appends the integer value `6` to the buffer using `StringBuffer`'s `append(int x)` method.
4. Invokes the `toString()` method on the buffer to produce a string on the runtime stack.

Solution 5. Analysis determines both the validity of a cast and the necessary converter, which encapsulates the code generated for the particular cast. Each Converter implements a method `codegen()`, which generates any code necessary to the cast. Code is first generated for the expression being cast, and then for the cast, using the appropriate converter.

Solution 6.

```java
public void codegen(CLEmitter output) {
    String bodyStart = output.createLabel();
    output.addLabel(bodyStar);
    body.codegen(output);
    condition.codegen(output, bodyStart, true);
}
```


6 Translating JVM Code to MIPS Code (Chapter 6)

PROBLEMS

Problem 1. Consider the following *j*-- program `SpimSum.java`:

```java
import spim.SPIM;

public class SpimSum {
    public static int compute(int n) {
        int sum = 0, i = n;
        while (i > 0) {
            sum += i--;
        }
        return sum;
    }

    public static void main(String[] args) {
        int result = SpimSum.compute(100);
        SPIM.printInt(result);
        SPIM.printChar('\n');
    }
}
```

The JVM bytecode for the `SpimSum.compute()` method are listed below, with linebreaks denoting basic blocks.

```
public static int compute(int);
  Code:
    stack=2, locals=3, args_size=1
       0: iconst_0
       1: istore_1
       2: iload_0
```

```
     3: istore_2

     4: iload_2
     5: iconst_0
     6: if_icmple      19

     9: iload_1
    10: iload_2
    11: iinc           2, -1
    14: iadd
    15: istore_1
    16: goto           4

    19: iload_1
    20: ireturn
```

The HIR instructions for the method are listed below.

```
B0 succ: B1
Locals: I0
I0: LDLOC 0

B1 [0, 3] dom: B0 pred: B0  succ: B2
Locals: I0 I3 I0
I3: 0

B2 [LH] [4, 6] dom: B1 pred: B1 B3  succ: B3 B4
Locals: I0 I5 I6
I5: [ I3 I11 ]
I6: [ I0 I10 ]
I7: 0
8: if I6 <= I7 then B4 else B3

B3 [LT] [9, 16] dom: B2 pred: B2 succ: B2
Locals: I0 I11 I10
I9: -1
I10: I6 + I9
I11: I5 + I6
12: goto B2

B4 [19, 20] dom: B2 pred: B2
Locals: I0 I5 I6
I13: ireturn I5
```


a. Draw the HIR flow graph for `SpimSum.compute()`.

b. Suppse that the HIR to LIR translation procedure assigns virtual registers `V32`, `V33`, `V34`, `V37`, and `38` to HIR instructions `I3` `I5`, `I6`, `I10`, and `I11` respectively. How are the Phi functions `I5` and `I6` in block `B2` resolved?

Problem 2. What optimization techniques would you use to improve each of the following code snippets, and how?

a.
```java
static int f() {
    return 42;
}

static int g(int x) {
    return f() * x;
}
```

b.
```java
static int f() {
    int x = 28;
    int y = 42
    int z = x + y * 10;
    return z;
}
```

c.
```java
static int f(int x) {
    int y = x * x * x;
    return x * x * x;
}
```

d.
```java
static int f(int[][] a) {
    int sum = 0;
    int i = 0;
    while (i <= a.length - 1) {
        int j = 0;
        while (j < a[0].length - 1) {
            sum += a[i][j];
            j = j + 1;
        }
        i = i + 1;
    }
    return sum;
}
```

f.
```java
static int f(int[][] a, int[][] b, int[][] c) {
    ...
    c[i][j] = a[i][j] + b[i][j];
    ...
}
```


where `a`, `b`, and `c` have the same dimensions.

g.
```java
static int f(SomeObject o) {
    return o.x * o.y * o.z;
}
```


where `x`, `y`, and `z` are integer fields in `o`.

Solutions

Solution 1.

a.

b. The Phi functions `I5` and `I6` in block `B2` are resolved by adding `move` instructions at the end of `B2`'s predecessors `B1` and `B3`, as follows:

```
B1:
  move V32 V33
  move $a0 V34

B3:
  move V37 V33
  move V38 V34
```

Solution 2. a. Inlining

```java
static int g(int x) {
    return 42 * x;
}
```

b. Constant folding and constant propagation

```java
static int f() {
    return 448;
}
```

c. Common subexpression elimination

```
static int f(int x) {
    int y = x * x * x;
    return y;
}
```

d. Lifting loop invariant code

```
static int f(int[][] a) {
    int sum = 0;
    int i = 0;
    while (i <= a.length - 1) {
        int j = 0;
        int[] a_ = a[i];
        while (j < a_.length - 1) {
            sum += a_[j];
            j = j + 1;
        }
        i = i + 1;
    }
    return sum;
}
```

e. Array bounds check elimination. Since `a`, `b`, and `c` have the same dimensions, perform the array bounds check, ie, check if the indices `i` and `j` are within bounds, just once instead of three times.

f. Null check elimination. Perform null check on the object `c` just once instead of three times.

7 Register Allocation (Chapter 7)

Problems

Problem 1. The LIR instructions for the `compute()` method from the `SpimSum` program above are listed below.


```
B0

B1
0: LDC [0] [V32|I]
5: MOVE [V32|I] [V33|I]
10: MOVE $a0 [V34|I]

B2
15: LDC [0] [V35|I]
20: BRANCH [LE] [V34|I] [V35|I] B4

B3
25: LDC [-1] [V36|I]
30: ADD [V34|I] [V36|I] [V37|I]
35: ADD [V33|I] [V34|I] [V38|I]
40: MOVE [V37|I] [V34|I]
45: MOVE [V38|I] [V33|I]
50: BRANCH B2

B4
55: MOVE [V33|I] $v0
60: RETURN $v0
```

a. Compute the liveUse and liveDef sets (local liveness information) for each basic block in the method.

b. Compute the liveIn and liveOut sets (global liveness information) for each basic block in the method.

c. Compute the liveness interval for each virtual register in the method's LIR, with ranges and use positions.

Problem 2. Using the liveness intervals for the virtual registers in the LIR for the `SpimSum.compute()` method

a. Build an interference graph G for the method.

b. Represent G as an adjacency matrix.

c. Represent G as an adjacency list.

d. Is G 2-colorable? If so, give a register allocation using two physical registers r_1 and r_2.

e. Is G 3-colorable? If so, give a register allocation using three physical registers r_1, r_2, and r_3.

Solutions

Solution 1.

a.
```
B0
  liveUse:
  liveDef:

B1
  liveUse: $a0
  liveDef: V32, V33, V34

B2
  liveUse: $a0, V34
  liveDef: V35

B3
  liveUse: V33, V34
  liveDef: V33, V34, V36, V37, V38

B4
  liveUse: V33
  liveDef: $v0
```


b.
```
B4
  liveIn:  V33
  liveOut:

B3
  liveIn:  V33, V34
  liveOut: V33

B2
  liveIn:  $a0, V34
  liveOut: V33, V34

B1
  liveIn:  $a0
  liveOut: $a0, V34

B0
  liveIn:
  liveOut: $a0
```

c.
```
  v0: [55, 60]
  a0: [0, 10]
 V32: [0, 5]
 V33: [5, 35] [45, 55]
 V34: [10, 50]
 V35: [15, 20]
 V36: [25, 30]
 V37: [30, 40]
 V38: [35, 45]
```

Solution 2.

a.

b.

c.

d.