Introduction to Compiler Construction

Compilation: Overview of *j--* to JVM Compiler

Outline

1. The *j*-- Compiler
2. Adding New Constructs to *j*--

The j-- Compiler

The j-- Compiler

j-- is a compiler for a subset of Java, with support for classes, methods, fields, statements, and expressions

The j-- Compiler

j-- is a compiler for a subset of Java, with support for classes, methods, fields, statements, and expressions

The base *j*-- compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The j-- Compiler

j-- is a compiler for a subset of Java, with support for classes, methods, fields, statements, and expressions

The base *j--* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/j--/src/jminusminus` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating JVM code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- Other supporting Java files

The j-- Compiler

j-- is a compiler for a subset of Java, with support for classes, methods, fields, statements, and expressions

The base *j--* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/j--/src/jminusminus` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating JVM code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- Other supporting Java files

The directory `$j/j--/bin` contains wrapper scripts

The j-- Compiler

j-- is a compiler for a subset of Java, with support for classes, methods, fields, statements, and expressions

The base *j--* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/j--/src/jminusminus` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating JVM code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- Other supporting Java files

The directory `$j/j--/bin` contains wrapper scripts

The directory `$j/j--/tests` contains test programs

The j-- Compiler

j-- is a compiler for a subset of Java, with support for classes, methods, fields, statements, and expressions

The base *j--* compiler may be downloaded and extracted into any directory (referred to as `$j`) of your choosing

The directory `$j/j--/src/jminusminus` contains

- `Main.java`, the driver program
- A hand-crafted scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files for generating JVM code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- Other supporting Java files

The directory `$j/j--/bin` contains wrapper scripts

The directory `$j/j--/tests` contains test programs

The file `$j/j--/build.xml` is the Ant build configuration file

The j-- Compiler

The j-- Compiler

To compile the compiler, run

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

$ ant
```

The j-- Compiler

To compile the compiler, run

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

$ ant
```

Usage syntax for the compiler

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

$ ./bin/j--
Usage: j-- <options> <source file>
Where possible options include:
  -t  Tokenize input and print tokens to STDOUT
  -p  Parse input and print AST to STDOUT
  -pa Pre-analyze input and print AST to STDOUT
  -a  Analyze input and print AST to STDOUT
  -d  <dir> Specify where to place output (.class) files; default = .
```

The j-- Compiler

The j-- Compiler

To compile a *j--* program `$j/j--/tests/HelloWorld.java` for the JVM, run

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

$ ./bin/j-- tests/HelloWorld.java
```

The j-- Compiler

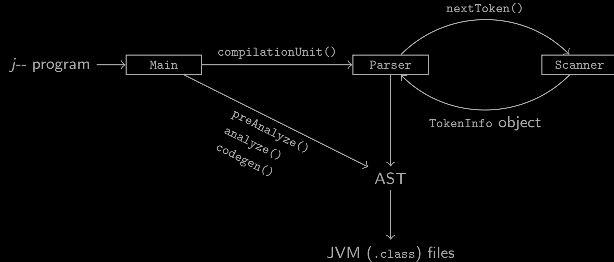
To compile a *j*-- program `$j/j--/tests/HelloWorld.java` for the JVM, run

```
>_ ~/workspace/j--
$ ./bin/j-- tests/HelloWorld.java
```

To run the generated JVM program `HelloWorld.class`, run

```
>_ ~/workspace/j--
$ java HelloWorld
Hello, World
```

The *j--* compiler is organized in an object-oriented fashion



The scanner breaks down a *j--* program into a sequence of tokens

The scanner breaks down a *j--* program into a sequence of tokens

Example: the following program

</> HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import java.lang.System;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

is broken down into `import`, `java`, `.`, `lang`, `.`, `System`,`;`, `public`, `class`, `HelloWorld`, `{`, `...`, `;`, `}`, `}`

The scanner breaks down a *j--* program into a sequence of tokens

Example: the following program

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import java.lang.System;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

is broken down into `import`, `java`, `.`, `lang`, `.`, `System`,`;`, `public`, `class`, `HelloWorld`, `{`, `...`, `;`, `}`, `}`

`.`, `;`, etc are separators with distinct names DOT, SEMI, etc

The scanner breaks down a *j--* program into a sequence of tokens

Example: the following program

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import java.lang.System;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

is broken down into `import`, `java`, `.`, `lang`, `.`, `System`,`;`, `public`, `class`, `HelloWorld`, `{`, `...`, `;`, `}`, `}`

`.`, `;`, etc are separators with distinct names DOT, SEMI, etc

`java`, `lang`, etc are IDENTIFIER tokens with the images `"java"`, `"lang"`, etc

The scanner breaks down a *j*-- program into a sequence of tokens

Example: the following program

</> HelloWorld.java

```
1  // Writes the message "Hello, World" to standard output.
2
3  import java.lang.System;
4
5  public class HelloWorld {
6      // Entry point.
7      public static void main(String[] args) {
8          System.out.println("Hello, World");
9      }
10 }
```

is broken down into `import`, `java`, `.`, `lang`, `.`, `System`,`;`, `public`, `class`, `HelloWorld`, `{`, `...`, `;`, `}`, `}`

`.`, `;`, etc are separators with distinct names `DOT`, `SEMI`, etc

`java`, `lang`, etc are `IDENTIFIER` tokens with the images `"java"`, `"lang"`, etc

`import`, `public`, etc are reserved words (aka keywords) with distinct names `IMPORT` and `PUBLIC`, etc

The scanner breaks down a *j--* program into a sequence of tokens

Example: the following program

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import java.lang.System;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

is broken down into `import`, `java`, `.`, `lang`, `.`, `System`,`;`, `public`, `class`, `HelloWorld`, `{`, `...`, `;`, `}`, `}`

`.`, `;`, etc are separators with distinct names `DOT`, `SEMI`, etc

`java`, `lang`, etc are `IDENTIFIER` tokens with the images `"java"`, `"lang"`, etc

`import`, `public`, etc are reserved words (aka keywords) with distinct names `IMPORT` and `PUBLIC`, etc

`"Hello, World"` is a `STRING_LITERAL` token with the image `"Hello, World"`

The parser validates the syntax of a *j--* program against the *j--* grammar and represents the program as an AST

The parser validates the syntax of a *j--* program against the *j--* grammar and represents the program as an AST

Grammar rules describing a compilation unit and a qualified identifier

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

The parser validates the syntax of a *j--* program against the *j--* grammar and represents the program as an AST

Grammar rules describing a compilation unit and a qualified identifier

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF

qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

A parser can be hand-crafted or generated

Parser.java

```java
    public JCompilationUnit compilationUnit() {
        int line = scanner.token().line();
        String fileName = scanner.fileName();
        TypeName packageName = null;
        if (have(PACKAGE)) {
            packageName = qualifiedIdentifier();
            mustBe(SEMI);
        }
        ArrayList<TypeName> imports = new ArrayList<>();
        while (have(IMPORT)) {
            imports.add(qualifiedIdentifier());
            mustBe(SEMI);
        }
        ArrayList<JAST> typeDeclarations = new ArrayList<>();
        while (!see(EOF)) {
            JAST typeDeclaration = typeDeclaration();
            typeDeclarations.add(typeDeclaration);
        }
        mustBe(EOF);
        return new JCompilationUnit(fileName, line, packageName, imports, typeDeclarations);
    }

    private TypeName qualifiedIdentifier() {
        int line = scanner.token().line();
        mustBe(IDENTIFIER);
        String qualifiedIdentifier = scanner.previousToken().image();
        while (have(DOT)) {
            mustBe(IDENTIFIER);
            qualifiedIdentifier += "." + scanner.previousToken().image();
        }
        return new TypeName(line, qualifiedIdentifier);
    }
```

The j-- Compiler · Parsing

```
{
    "JCompilationUnit:3":
    {
        "source": "tests/HelloWorld.java", "imports": ["java.lang.System"],
        "JClassDeclaration:5":
        {
            "modifiers": ["public"],
            "name": "HelloWorld",
            "super": "java.lang.Object",
            "JMethodDeclaration:7":
            {
                "modifiers": ["public", "static"],
                "returnType": "void",
                "name": "main",
                "parameters": [["args", "String[]"]],
                "JBlock:7":
                {
                    "JStatementExpression:8":
                    {
                        "JMessageExpression:8":
                        {
                            "ambiguousPart": "System.out", "name": "println",
                            "Argument":
                            {
                                "JLiteralString:8":
                                {
                                    "type": "",
                                    "value": "Hello, World"
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

j--, being statically typed, must determine the types of all names and expressions

j--, being statically typed, must determine the types of all names and expressions

Types in *j--* are represented using

- `Type` (wraps `java.lang.Class`)
- `Member` (wraps `java.lang.reflect.Member`)
- `Field` (wraps `java.lang.reflect.Field`)
- `Constructor` (wraps `java.lang.reflect.Constructor`)
- `Method` (wraps `java.lang.reflect.Method`)

j--, being statically typed, must determine the types of all names and expressions

Types in *j*-- are represented using

- `Type` (wraps `java.lang.Class`)
- `Member` (wraps `java.lang.reflect.Member`)
- `Field` (wraps `java.lang.reflect.Field`)
- `Constructor` (wraps `java.lang.reflect.Constructor`)
- `Method` (wraps `java.lang.reflect.Method`)

In some places *j*-- uses `TypeName` and `ArrayTypeName` to denote a type by its name, before the type is known

j--, being statically typed, must determine the types of all names and expressions

Types in *j--* are represented using

- `Type` (wraps `java.lang.Class`)
- `Member` (wraps `java.lang.reflect.Member`)
- `Field` (wraps `java.lang.reflect.Field`)
- `Constructor` (wraps `java.lang.reflect.Constructor`)
- `Method` (wraps `java.lang.reflect.Method`)

In some places *j--* uses `TypeName` and `ArrayTypeName` to denote a type by its name, before the type is known

An ambiguous expression such as `x.y.z` in `x.y.z.w()` is denoted as `AmbiguousName` by the parser and is reclassified during analysis

j-- maintains a singly-linked list of `Context` objects in which it declares names

j-- maintains a singly-linked list of `Context` objects in which it declares names

Each object in the list represents some region of scope and contains a symbol table that maps names to definitions

j-- maintains a singly-linked list of `Context` objects in which it declares names

Each object in the list represents some region of scope and contains a symbol table that maps names to definitions

A `CompilationUnitContext` object represents the scope comprising the program

j-- maintains a singly-linked list of `Context` objects in which it declares names

Each object in the list represents some region of scope and contains a symbol table that maps names to definitions

A `CompilationUnitContext` object represents the scope comprising the program

A `ClassContext` object represents the scope of a class declaration

j-- maintains a singly-linked list of `Context` objects in which it declares names

Each object in the list represents some region of scope and contains a symbol table that maps names to definitions

A `CompilationUnitContext` object represents the scope comprising the program

A `ClassContext` object represents the scope of a class declaration

A `LocalContext` object represents the scope of a block

j-- maintains a singly-linked list of `Context` objects in which it declares names

Each object in the list represents some region of scope and contains a symbol table that maps names to definitions

A `CompilationUnitContext` object represents the scope comprising the program

A `ClassContext` object represents the scope of a class declaration

A `LocalContext` object represents the scope of a block

A `MethodContext` (subclass of `LocalContext`) object represents the scopes of methods/constructors

The `preAnalyze()` method builds the symbol table at the top of the AST, declaring imported types, user-defined types, and their members

The `preAnalyze()` method builds the symbol table at the top of the AST, declaring imported types, user-defined types, and their members

The `analyze()` method builds the rest of the symbol tables and decorates the AST with type information

The `preAnalyze()` method builds the symbol table at the top of the AST, declaring imported types, user-defined types, and their members

The `analyze()` method builds the rest of the symbol tables and decorates the AST with type information

The `analyze()` method also does type checking, accessibility checking, member finding, tree rewriting, and storage allocation

The `preAnalyze()` method builds the symbol table at the top of the AST, declaring imported types, user-defined types, and their members

The `analyze()` method builds the rest of the symbol tables and decorates the AST with type information

The `analyze()` method also does type checking, accessibility checking, member finding, tree rewriting, and storage allocation

Example (analysis of a while-statement)

</> JWhileStatement.java

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

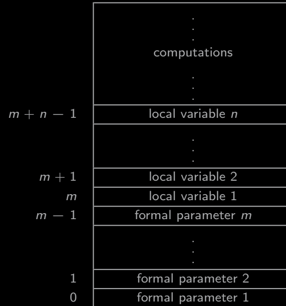
The JVM is a stack machine — all computations are carried out atop the run-time stack

The JVM is a stack machine — all computations are carried out atop the run-time stack

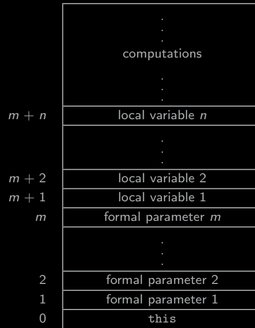
Each time a method is called, the JVM

- Allocates a stack frame (contiguous block of memory locations) on top of the stack
- Assigns positions on the frame for formal parameters and substitutes actual arguments for the parameters
- Assigns positions on the frame for values of local variables and temporary results

Stack frame for a static method call with m formal parameters and n local variables



Stack frame for an instance method call with m formal parameters and n local variables



A *j--* method

```
    public static int multiply(int x, int y) {
        int z = x * y;
        return z;
    }
```

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

Stack frame for the call `multiply(6, 7)`

2	z :
1	y : 7
0	x : 6

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

Stack frame for the call `multiply(6, 7)`

	6
2	z :
1	y : 7
0	x : 6

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

Stack frame for the call `multiply(6, 7)`

	7
	6
2	z :
1	y : 7
0	x : 6

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

Stack frame for the call `multiply(6, 7)`

	42
2	z :
1	y : 7
0	x : 6

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

Stack frame for the call `multiply(6, 7)`

2	z : 42
1	y : 7
0	x : 6

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

Stack frame for the call `multiply(6, 7)`

	42
2	z : 42
1	y : 7
0	x : 6

A *j--* method

```
public static int multiply(int x, int y) {
    int z = x * y;
    return z;
}
```

JVM code for the method

```
public static int multiply(int, int);
    stack=2, locals=3, args_size=2
       0: iload_0
       1: iload_1
       2: imul
       3: istore_2
       4: iload_2
       5: ireturn
```

Stack frame for the call `multiply(6, 7)`

poof!

GenFactorial.java 1/4

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

/**
 * This class programatically generates the class file for the following Java application using CLEmitter:
 *
 * <pre>
 * public class Factorial {
 *     public static void main(String[] args) {
 *         int n = Integer.parseInt(args[0]);
 *         int result = factorial(n);
 *         System.out.println(n + "! = " + result);
 *     }
 *
 *     private static int factorial(int n) {
 *         if (n <= 1) {
 *             return 1;
 *         }
 *         return n * factorial(n - 1);
 *     }
 * }
 * </pre>
 */
public class GenFactorial {
    public static void main(String[] args) {
        // Create a CLEmitter instance
        CLEmitter e = new CLEmitter(true);

        // Create an ArrayList instance to store modifiers
        ArrayList<String> modifiers = new ArrayList<String>();

        // public class Factorial {
        modifiers.add("public");
        e.addClass(modifiers, "Factorial", "java/lang/Object", null, true);
```

GenFactorial.java 2/4

```
36
37         // public static void main(String[] args) {
38         modifiers.clear();
39         modifiers.add("public");
40         modifiers.add("static");
41         e.addMethod(modifiers, "main", "([Ljava/lang/String;)V", null, true);
42
43         // int n = Integer.parseInt(args[0]);
44         e.addNoArgInstruction(ALOAD_0);
45         e.addNoArgInstruction(ICONST_0);
46         e.addNoArgInstruction(AALOAD);
47         e.addMemberAccessInstruction(INVOKESTATIC, "java/lang/Integer", "parseInt", "(Ljava/lang/String;)I");
48         e.addNoArgInstruction(ISTORE_1);
49
50         // int result = factorial(n);
51         e.addNoArgInstruction(ILOAD_1);
52         e.addMemberAccessInstruction(INVOKESTATIC, "Factorial", "factorial", "(I)I");
53         e.addNoArgInstruction(ISTORE_2);
54
55         // System.out.println(n + "! = " + result);
56
57         // Get System.out on stack
58         e.addMemberAccessInstruction(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
59
60         // Create an intance (say sb) of StringBuffer on stack for string concatenations
61         //    sb = new StringBuffer();
62         e.addReferenceInstruction(NEW, "java/lang/StringBuffer");
63         e.addNoArgInstruction(DUP);
64         e.addMemberAccessInstruction(INVOKESPECIAL, "java/lang/StringBuffer", "<init>", "()V");
65
66         // sb.append(n);
67         e.addNoArgInstruction(ILOAD_1);
68         e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer", "append", "(I)Ljava/lang/StringBuffer;");
69
70         // sb.append("!=");
```

GenFactorial.java

```
71          e.addLDCInstruction("! = ");
72          e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer", "append",
73                  "(Ljava/lang/String;)Ljava/lang/StringBuffer;");
74
75          // sb.append(result);
76          e.addNoArgInstruction(ILOAD_2);
77          e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer", "append", "(I)Ljava/lang/StringBuffer;");
78
79          // System.out.println(sb.toString());
80          e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer",
81                  "toString", "()Ljava/lang/String;");
82          e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");
83
84          // return;
85          e.addNoArgInstruction(RETURN);
86
87          // private static int factorial(int n) {
88          modifiers.clear();
89          modifiers.add("private");
90          modifiers.add("static");
91          e.addMethod(modifiers, "factorial", "(I)I", null, true);
92
93          // if (n > 1) branch to "Recurse"
94          e.addNoArgInstruction(ILOAD_0);
95          e.addNoArgInstruction(ICONST_1);
96          e.addBranchInstruction(IF_ICMPGT, "Recurse");
97
98          // Base case: return 1;
99          e.addNoArgInstruction(ICONST_1);
100         e.addNoArgInstruction(IRETURN);
101
102         // Recursive case: return n * factorial(n - 1);
103         e.addLabel("Recurse");
104         e.addNoArgInstruction(ILOAD_0);
105         e.addNoArgInstruction(ILOAD_0);
```

GenFactorial.java

```
106         e.addNoArgInstruction(ICONST_1);
107         e.addNoArgInstruction(ISUB);
108         e.addMemberAccessInstruction(INVOKESTATIC, "Factorial", "factorial", "(I)I");
109         e.addNoArgInstruction(IMUL);
110         e.addNoArgInstruction(IRETURN);
111 
112         // Write Factorial.class to file system
113         e.write();
114     }
115 }
```

To compile `GenFactorial.java`, run

```
>_ ~/workspace/j--
$ ./bin/clemitter tests/GenFactorial.java
```

To compile `GenFactorial.java`, run

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

$ ./bin/clemitter tests/GenFactorial.java
```

To run the generated JVM program `Factorial.class`, run

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

$ java Factorial 5
5! = 120
```

The `codegen()` method, starting at the root, recursively descends the AST, generating JVM code

Example (code generation for a while-statement)

</> JWhileStatement.java

```
1    public void codegen(CLEmitter output) {
2        String testLabel = output.createLabel();
3        String endLabel = output.createLabel();
4        output.addLabel(testLabel);
5        condition.codegen(output, endLabel, false);
6        body.codegen(output);
7        output.addBranchInstruction(GOTO, testLabel);
8        output.addLabel(endLabel);
9    }
```

Adding New Constructs to j--

Adding New Constructs to j--

To add a new construct (eg, `/` operator) to the *j--* language, we must

- Modify the scanner
- Modify the parser
- Implement type checking (aka semantic analysis)
- Implement code generation
- Test the changes

</> TokenInfo.java

```
enum TokenKind {
    DIV ("/"),
}
```

</> TokenInfo.java

```
enum TokenKind {
    DIV ("/"),
}
```

</> Scanner.java

```
1              if (ch == '/') {
2                  nextCh();
3                  if (ch == '/') {
4                      // CharReader maps all new lines to '\n'.
5                      while (ch != '\n' && ch != EOFCH) {
6                          nextCh();
7                      }
8                  } else {
9                      return new TokenInfo(DIV, line);
10                 }
11             }
```

</> JBinaryExpression.java

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

    public JExpression analyze (Context context) {
        // TODO
        return this;
    }

    public void codegen(CLEmitter output) {
        // TODO
    }
}
```

Parser.java

```
1      /**
2       * Parses a multiplicative expression and returns an AST for it.
3       *
4       * <pre>
5       *   multiplicativeExpression ::= unaryExpression { ( DIV | STAR ) unaryExpression }
6       * </pre>
7       *
8       * @return an AST for a multiplicative expression.
9       */
10     private JExpression multiplicativeExpression() {
11         int line = scanner.token().line();
12         boolean more = true;
13         JExpression lhs = unaryExpression();
14         while (more) {
15             if (have(STAR)) {
16                 lhs = new JMultiplyOp(line, lhs, unaryExpression());
17             }
18             else if (have(DIV)) {
19                 lhs = new JDivideOp(line, lhs, unaryExpression());
20             }
21             else {
22                 more = false;
23             }
24         }
25         return lhs;
26     }
```

Adding New Constructs to j-- · Example (Adding Support for Division)

</> JBinaryExpression.java

```
class JDivideOp extends JBinaryExpression {
    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(IDIV);
    }
}
```

</> Division.java

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

public class Division {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        System.out.println(a / b);
    }
}
```

To compile the changes, run

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

To compile the changes, run

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

To compile the test program using *j--*, run

```
>_ ~/workspace/j--
$ ./bin/j-- tests/Division.java
```

To compile the changes, run

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

To compile the test program using *j--*, run

```
>_ ~/workspace/j--
$ ./bin/j-- tests/Division.java
```

To run the generated JVM program `Division.class`, run

```
>_ ~/workspace/j--
$ java Division 42 6
7
```