Compilation

Outline

1. Compilers
2. Why Study Compilers?
3. Phases of Compilation
4. The *j--* Compiler
5. Adding New Constructs to *j--*

Compilers

Compilers

A compiler translates a source language program into a target language program

Compilers

A compiler translates a source language program into a target language program

Examples of source language: C, Java

Compilers

A compiler translates a source language program into a target language program

source language program (high-level) $\longrightarrow$ $\boxed{\text{compiler}}$ $\longrightarrow$ target language program (low-level)

Examples of source language: C, Java

Examples of target language: MIPS instructions, JVM instructions (aka bytecode)

Compilers

Compilers

A programming language specification consists of:

A programming language specification consists of:

- Syntax of tokens

A programming language specification consists of:

- Syntax of tokens
- Syntax of constructs such as classes, methods, statements, and expressions

A programming language specification consists of:

- Syntax of tokens
- Syntax of constructs such as classes, methods, statements, and expressions
- Semantics (ie, meaning) of the constructs

Compilers

Compilers

A machine's instruction set along with its behavior is referred to as its architecture

A machine's instruction set along with its behavior is referred to as its architecture

Examples of machine architectures:

Compilers

A machine's instruction set along with its behavior is referred to as its architecture

Examples of machine architectures:

- Intel i386: a complex instruction set computer (CISC)

A machine's instruction set along with its behavior is referred to as its architecture

Examples of machine architectures:

- Intel i386: a complex instruction set computer (CISC)
- MIPS: a reduced instruction set computer (RISC)

A machine's instruction set along with its behavior is referred to as its architecture

Examples of machine architectures:

- Intel i386: a complex instruction set computer (CISC)
- MIPS: a reduced instruction set computer (RISC)
- Java Virtual Machine (JVM): a virtual machine

Compilers

Compilers

An interpreter executes a source language program directly

Compilers

An interpreter executes a source language program directly

Examples of interpreters: Bash, Python

Why Study Compilers?

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Compilers make use of all those things you have learned about earlier

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Compilers make use of all those things you have learned about earlier

You learn a lot about the source language (in our case, Java)

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Compilers make use of all those things you have learned about earlier

You learn a lot about the source language (in our case, Java)

You learn a lot about the target machine (in our case, JVM and MIPS)

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Compilers make use of all those things you have learned about earlier

You learn a lot about the source language (in our case, Java)

You learn a lot about the target machine (in our case, JVM and MIPS)

Compilers are still being written for new languages and targeted to new architectures

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Compilers make use of all those things you have learned about earlier

You learn a lot about the source language (in our case, Java)

You learn a lot about the target machine (in our case, JVM and MIPS)

Compilers are still being written for new languages and targeted to new architectures

There is a good mix of theory and practice

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Compilers make use of all those things you have learned about earlier

You learn a lot about the source language (in our case, Java)

You learn a lot about the target machine (in our case, JVM and MIPS)

Compilers are still being written for new languages and targeted to new architectures

There is a good mix of theory and practice

Compiler writing is a case study in software engineering

Why Study Compilers?

Compilers are larger programs than the ones you have written so far

Compilers make use of all those things you have learned about earlier

You learn a lot about the source language (in our case, Java)

You learn a lot about the target machine (in our case, JVM and MIPS)

Compilers are still being written for new languages and targeted to new architectures

There is a good mix of theory and practice

Compiler writing is a case study in software engineering

Compilers are programs and writing programs is fun

Phases of Compilation

Phases of Compilation

A compiler can be broken into a front end and a back end

source language program $\longrightarrow$ $\boxed{\text{front end}}$ $\longrightarrow$ IR† $\longrightarrow$ $\boxed{\text{back end}}$ $\longrightarrow$ target language program

mesh language

† Intermediate Representation

Phases of Compilation

Phases of Compilation

The front end can be decomposed into a sequence of analysis phases

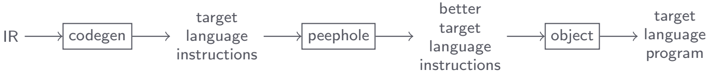
$$\text{source language program} \longrightarrow \boxed{\text{scanner}} \longrightarrow \text{tokens} \longrightarrow \boxed{\text{parser}} \longrightarrow \text{AST}^{\dagger} \longrightarrow \boxed{\text{semantics}} \longrightarrow \text{IR}$$

† Abstract Syntax Tree

Phases of Compilation

Phases of Compilation

The back end can be decomposed into a sequence of synthesis phases



Phases of Compilation

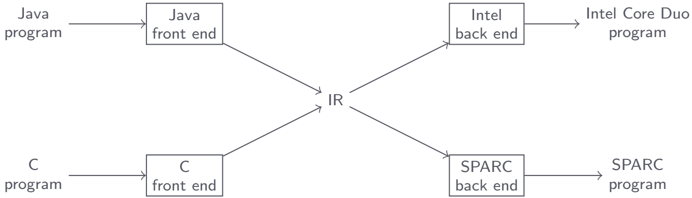
Phases of Compilation

A compiler sometimes has an optimizer between the front end and the back end

Phases of Compilation

Phases of Compilation

Separating the front end from the back end enables code re-use



The j-- Compiler

The j-- Compiler

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

The j-- Compiler

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

Compiling a *j--* program `$j/j--/tests/jvm/HelloWorld.java` for the JVM

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

The j-- Compiler

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

Compiling a *j--* program `$j/j--/tests/jvm/HelloWorld.java` for the JVM

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

Running the JVM program `HelloWorld.class`

```
>_ ~/workspace/j--
$ java HelloWorld
```

The j-- Compiler

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

Compiling a *j--* program `$j/j--/tests/jvm/HelloWorld.java` for the JVM

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

Running the JVM program `HelloWorld.class`

```
>_ ~/workspace/j--
$ java HelloWorld
```

Compiling a *j--* program `$j/j--/tests/spim/HelloWorld.java` for the MIPS machine

```
>_ ~/workspace/j--
$ bash ./bin/j-- -s naive tests/spim/HelloWorld.java
```

The j-- Compiler

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

Compiling a *j--* program `$j/j--/tests/jvm/HelloWorld.java` for the JVM

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

Running the JVM program `HelloWorld.class`

```
>_ ~/workspace/j--
$ java HelloWorld
```

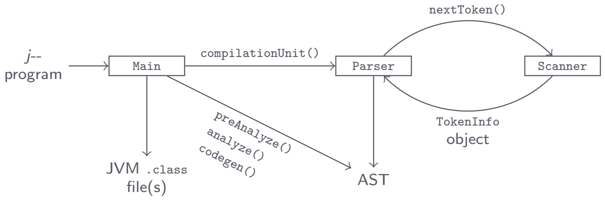
Compiling a *j--* program `$j/j--/tests/spim/HelloWorld.java` for the MIPS machine

```
>_ ~/workspace/j--
$ bash ./bin/j-- -s naive tests/spim/HelloWorld.java
```

Running the MIPS program `HelloWorld.s`

```
>_ ~/workspace/j--
$ spim -f HelloWorld.s
```

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



The j-- Compiler

The j-- Compiler

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

The j-- Compiler

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

For example, the following program

HelloWorld.java

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
//
// Writes to standard output the message "Hello, World".

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 j-- Compiler

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

For example, the following program

HelloWorld.java

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
//
// Writes to standard output the message "Hello, World".

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`, `{`, `...`, `;`, `}`, `}`

`import`, `public`, etc are reserved words with distinct names IMPORT and PUBLIC, etc

The j-- Compiler

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

For example, the following program

HelloWorld.java

```
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
//
// Writes to standard output the message "Hello, World".

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`, `{`, `. . .`, `;`, `}`, `}`

`import`, `public`, etc are reserved words with distinct names IMPORT and PUBLIC, etc

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

The j-- Compiler

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

For example, the following program

HelloWorld.java

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
//
// Writes to standard output the message "Hello, World".

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`, `{`, `...`, `;`, `}`, `}`

`import`, `public`, etc are reserved words with distinct names IMPORT and PUBLIC, etc

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

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

The j-- Compiler

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

For example, the following program

HelloWorld.java

```java
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
//
// Writes to standard output the message "Hello, World".

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`, `{`, `. . .`, `;`, `}`, `}`

`import`, `public`, etc are reserved words with distinct names `IMPORT` and `PUBLIC`, etc

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

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

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

The j-- Compiler

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

In the first instance, the parser is hand-crafted from the grammar, to parse programs using the recursive descent algorithm

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

In the first instance, the parser is hand-crafted from the grammar, to parse programs using the recursive descent algorithm

Grammar rules describing a compilation unit and a qualified identifier

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

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

The j-- Compiler

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<TypeName>();
    while (have(IMPORT)) {
        imports.add(qualifiedIdentifier());
        mustBe(SEMI);
    }
    ArrayList<JAST> typeDeclarations = new ArrayList<JAST>();
    while (!see(EOF)) {
        JAST typeDeclaration = typeDeclaration();
        if (typeDeclaration != null) {
            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);
}
```

Have(PACKAGE) are we look at Package

The j-- Compiler

```
{
    "JCompilationUnit:5":
    {
        "source": "tests/jvm/HelloWorld.java",
        "imports": ["java.lang.System"],
        "JClassDeclaration:7":
        {
            "modifiers": ["public"],
            "name": "HelloWorld",
            "super": "java.lang.Object",
            "JMethodDeclaration:9":
            {
                "name": "main",
                "returnType": "void",
                "modifiers": ["public", "static"],
                "parameters": [["args", "String[]"]],
                "JBlock:9":
                {
                    "JStatementExpression:10":
                    {
                        "JMessageExpression:10":
                        {
                            "ambiguousPart": "System.out", "name": "println",
                            "Argument":
                            {
                                "JLiteralString:10":
                                {
                                    "type": "", "value": "Hello, World"
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

pasar don’ t know type and “system.out”

The j-- Compiler

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:

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

Types in *j--* are represented using:

- `Type` (wraps `java.lang.Class`)

The j-- Compiler

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

Types in *j--* are represented using:

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

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

Types in *j--* are represented using:

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

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

Types in *j--* are represented using:

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

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

Types in *j*-- are represented using:

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

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

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 area 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 area 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 area 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 area 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 area 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 j-- Compiler

The `preAnalyze()` method builds the part of the symbol table close to the top of the AST, declaring imported types, types introduced by class declarations, and their members

The `preAnalyze()` method builds the part of the symbol table close to the top of the AST, declaring imported types, types introduced by class declarations, and their members

The `analyze()` method builds the rest of the symbol table, decorating the AST with type information

The `preAnalyze()` method builds the part of the symbol table close to the top of the AST, declaring imported types, types introduced by class declarations, and their members

The `analyze()` method builds the rest of the symbol table, decorating 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 part of the symbol table close to the top of the AST, declaring imported types, types introduced by class declarations, and their members

The `analyze()` method builds the rest of the symbol table, decorating 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 j-- Compiler

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:

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

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

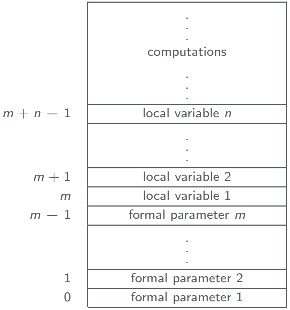
The JVM is a stack machine — all computations are carried out atop the run-time stack

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

The j-- Compiler

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



The j-- Compiler

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

	⋮ computations ⋮
$m + n$	local variable n
	⋮
$m + 2$	local variable 2
$m + 1$	local variable 1
m	formal parameter m
	⋮
2	formal parameter 2
1	formal parameter 1
0	`this`

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

```java
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

```java
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!

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!

`CLEmitter` provides an abstraction for the JVM class file

GenFactorial.java

```
import java.util.ArrayList;

import jminusminus.CLEmitter;

import static jminusminus.CLConstants.*;

/**
 * This class programatically generates the class file for the following Java application:
 *
 * <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 {
```

GenFactorial.java

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

        // public static void main(String[] args) {
        modifiers.clear();
        modifiers.add("public");
        modifiers.add("static");
        e.addMethod(modifiers, "main", "([Ljava/lang/String;)V", null, true);

        // int n = Integer.parseInt(args[0]);
        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);

        // int result = factorial(n);
        e.addNoArgInstruction(ILOAD_1);
        e.addMemberAccessInstruction(INVOKESTATIC, "Factorial", "factorial", "(I)I");
        e.addNoArgInstruction(ISTORE_2);

        // System.out.println(n + "! = " + result);

        // Get System.out on stack
        e.addMemberAccessInstruction(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");

        // Create an intance (say sb) of StringBuffer on stack for string concatenations
        //    sb = new StringBuffer();
        e.addReferenceInstruction(NEW, "java/lang/StringBuffer");
        e.addNoArgInstruction(DUP);
        e.addMemberAccessInstruction(INVOKESPECIAL, "java/lang/StringBuffer", "<init>", "()V");

        // sb.append(n);
        e.addNoArgInstruction(ILOAD_1);
```

GenFactorial.java

```
        e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer", "append",
                "(I)Ljava/lang/StringBuffer;");

        // sb.append("!=");
        e.addLDCInstruction("! = ");
        e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer", "append",
                "(Ljava/lang/String;)Ljava/lang/StringBuffer;");

        // sb.append(result);
        e.addNoArgInstruction(ILOAD_2);
        e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer", "append",
                "(I)Ljava/lang/StringBuffer;");

        // System.out.println(sb.toString());
        e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/lang/StringBuffer",
                "toString", "()Ljava/lang/String;");
        e.addMemberAccessInstruction(INVOKEVIRTUAL, "java/io/PrintStream", "println",
                "(Ljava/lang/String;)V");

        // return;
        e.addNoArgInstruction(RETURN);

        // private static int factorial(int n) {
        modifiers.clear();
        modifiers.add("private");
        modifiers.add("static");
        e.addMethod(modifiers, "factorial", "(I)I", null, true);

        // if (n > 1) branch to "Recurse"
        e.addNoArgInstruction(ILOAD_0);
        e.addNoArgInstruction(ICONST_1);
        e.addBranchInstruction(IF_ICMPGT, "Recurse");

        // Base case: return 1;
        e.addNoArgInstruction(ICONST_1);
```

GenFactorial.java

```
        e.addNoArgInstruction(IRETURN);

        // Recursive case: return n * factorial(n - 1);
        e.addLabel("Recurse");
        e.addNoArgInstruction(ILOAD_0);
        e.addNoArgInstruction(ILOAD_0);
        e.addNoArgInstruction(ICONST_1);
        e.addNoArgInstruction(ISUB);
        e.addMemberAccessInstruction(INVOKESTATIC, "Factorial", "factorial", "(I)I");
        e.addNoArgInstruction(IMUL);
        e.addNoArgInstruction(IRETURN);

        // Write Factorial.class to file system
        e.write();
    }
}
```

The j-- Compiler

Compile `GenFactorial.java`

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

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

The j-- Compiler

Compile `GenFactorial.java`

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

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

Run `Factorial.class`

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

$ java Factorial 5
5! = 120
```

The j-- Compiler

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

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

Example (code generation for a method declaration)

JMethodDeclaration.java

```java
public void codegen(CLEmitter output) {
    output.addMethod(mods, name, descriptor, null, false);
    if (body != null) {
        body.codegen(output);
    }
    if (returnType == Type.VOID) {
        output.addNoArgInstruction(RETURN);
    }
}
```

The j-- Compiler

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped into any directory (referred to as `$j`) of your choosing

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped into any directory (referred to as `$j`) of your choosing

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

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped into any directory (referred to as `$j`) of your choosing

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

- `Main.java`, the driver program

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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`)

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode
- `N*.java` files for translating JVM bytecode into MIPS code

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode
- `N*.java` files for translating JVM bytecode into MIPS code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode
- `N*.java` files for translating JVM bytecode into MIPS code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- `JavaCCMain.java`, the driver program that uses the generated scanner and parser

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode
- `N*.java` files for translating JVM bytecode into MIPS code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- `JavaCCMain.java`, the driver program that uses the generated scanner and parser
- Other supporting Java files

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode
- `N*.java` files for translating JVM bytecode into MIPS code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- `JavaCCMain.java`, the driver program that uses the generated scanner and parser
- Other supporting Java files

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

The j-- Compiler

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode
- `N*.java` files for translating JVM bytecode into MIPS code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- `JavaCCMain.java`, the driver program that uses the generated 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

The zip file `j--.zip` for the base *j--* compiler may be unzipped 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 creating JVM bytecode
- `N*.java` files for translating JVM bytecode into MIPS code
- `j--.jj`, the JavaCC specification file for generating a scanner and parser
- `JavaCCMain.java`, the driver program that uses the generated 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

Usage syntax for the *j--* compiler (`$j/j--/bin/j--`)

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

$ bash ./bin/j--
Usage: j-- <options> <source file>
Where possible options include:
  -t  Only tokenize input and print tokens to STDOUT
  -p  Only parse input and print AST to STDOUT
  -pa Only parse and pre-analyze input and print AST to STDOUT
  -a  Only parse, pre-analyze, and analyze input and print AST to STDOUT
  -s  <naive|linear|graph> Generate SPIM code
  -r  <num> Physical registers (1-18) available for allocation; default = 8
  -d  <dir> Specify where to place output files; default = .
```

Usage syntax for the *j--* compiler (`$j/j--/bin/j--`)

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

$ bash ./bin/j--
Usage: j-- <options> <source file>
Where possible options include:
  -t  Only tokenize input and print tokens to STDOUT
  -p  Only parse input and print AST to STDOUT
  -pa Only parse and pre-analyze input and print AST to STDOUT
  -a  Only parse, pre-analyze, and analyze input and print AST to STDOUT
  -s  <naive|linear|graph> Generate SPIM code
  -r  <num> Physical registers (1-18) available for allocation; default = 8
  -d  <dir> Specify where to place output files; default = .
```

For example, to just tokenize the *j--* program `$j/j--/tests/jvm/HelloWorld.java`, run

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

$ bash ./bin/j-- -t tests/jvm/HelloWorld.java
```

The j-- Compiler

Usage syntax for the *j--* compiler (`$j/j--/bin/j--`)

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

```
$ bash ./bin/j--
Usage: j-- <options> <source file>
Where possible options include:
  -t  Only tokenize input and print tokens to STDOUT
  -p  Only parse input and print AST to STDOUT
  -pa Only parse and pre-analyze input and print AST to STDOUT
  -a  Only parse, pre-analyze, and analyze input and print AST to STDOUT
  -s  <naive|linear|graph> Generate SPIM code
  -r  <num> Physical registers (1-18) available for allocation; default = 8
  -d  <dir> Specify where to place output files; default = .
```

For example, to just tokenize the *j--* program `$j/j--/tests/jvm/HelloWorld.java`, run

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

```
$ bash ./bin/j-- -t tests/jvm/HelloWorld.java
```

And to compile the program for the JVM, run

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

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

Adding New Constructs to j--

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j*-- language

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j*-- language

For example, to add the division operator (`/`) to *j*--, we must:

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j*-- language

For example, to add the division operator (`/`) to *j*--, we must:

- Modify the (lexical and syntactic) grammar and semantics files

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j--* language

For example, to add the division operator (/) to *j--*, we must:

- Modify the (lexical and syntactic) grammar and semantics files
- Modify the scanner

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j*-- language

For example, to add the division operator (`/`) to *j*--, we must:

- Modify the (lexical and syntactic) grammar and semantics files
- Modify the scanner
- Modify the parser

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j*-- language

For example, to add the division operator (`/`) to *j*--, we must:

- Modify the (lexical and syntactic) grammar and semantics files
- Modify the scanner
- Modify the parser
- Implement type checking (aka semantic analysis)

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j*-- language

For example, to add the division operator (/) to *j*--, we must:

- Modify the (lexical and syntactic) grammar and semantics files
- Modify the scanner
- Modify the parser
- Implement type checking (aka semantic analysis)
- Implement code generation

Adding New Constructs to j--

j-- provides an elaborate framework for adding new Java constructs to the *j*-- language

For example, to add the division operator (`/`) to *j*--, we must:

- Modify the (lexical and syntactic) grammar and semantics files
- Modify the scanner
- Modify the parser
- Implement type checking (aka semantic analysis)
- Implement code generation
- Test the changes

Adding New Constructs to j--

Adding New Constructs to j--

lexicalgrammar

```
DIV          ::= "/"
```

Adding New Constructs to j--

lexicalgrammar

```
DIV          ::= "/"
```

grammar

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

Adding New Constructs to j--

lexicalgrammar

```
DIV          ::= "/"
```

grammar

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

semantics

```
JBinaryExpression:
- JDivideOp
  - lhs and rhs must be integers.
```

Adding New Constructs to j--

Adding New Constructs to j--

TokenInfo.java

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

Adding New Constructs to j--

TokenInfo.java

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

Scanner.java

```
            if (ch == '/') {
                nextCh();
                if (ch == '/') {
                    // CharReader maps all new lines to '\n'.
                    while (ch != '\n' && ch != EOFCH) {
                        nextCh();
                    }
                } else {
                    return new TokenInfo(DIV, line);
                }
            }
```

Adding New Constructs to j--

Adding New Constructs to j--

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
    }
}
```

Adding New Constructs to j--

Adding New Constructs to j--

Parser.java

```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(DIV)) {
                lhs = new JDivideOp(line, lhs, unaryExpression());
            }
            else {
                more = false;
            }
        }
        return lhs;
    }
```

Adding New Constructs to j--

Adding New Constructs to j--

JBinaryExpression.java

```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);
    }
}
```

Adding New Constructs to j--

Adding New Constructs to j--

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);
    }
}
```

Adding New Constructs to j--

Adding New Constructs to j--

To compile the changes to the *j*-- compiler, go to `$j/j--`, and run

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

Adding New Constructs to j--

To compile the changes to the *j--* compiler, go to `$j/j--`, and run

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

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

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

Adding New Constructs to j--

To compile the changes to the *j--* compiler, go to `$j/j--`, and run

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

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

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

To run the test program (`Division.class`), run

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