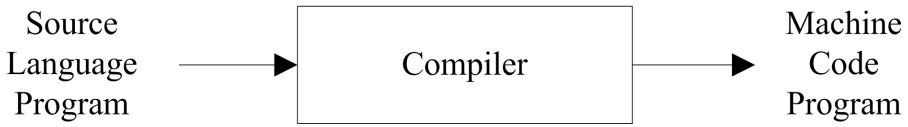
Compilation

Outline

Compilers

A compiler is a program that translates a source program written in a high-level programming language such as Java, C# or C, into a target program in a lower level language such as machine code



A programming language is an artificial language in which a programmer writes a program to control the behavior of a computer

Like a natural language, in a programming language, one describes

- The tokens (aka lexemes)
- The syntax of programs and language constructs such as classes, methods, statements and expressions
- The meaning (aka semantics) of the various constructs

Compilers

A computer's machine language (ie, its instruction set) is designed so as to be easily interpreted by the computer itself

A machine's instruction set and its behavior is often referred to as its architecture

Examples of architectures

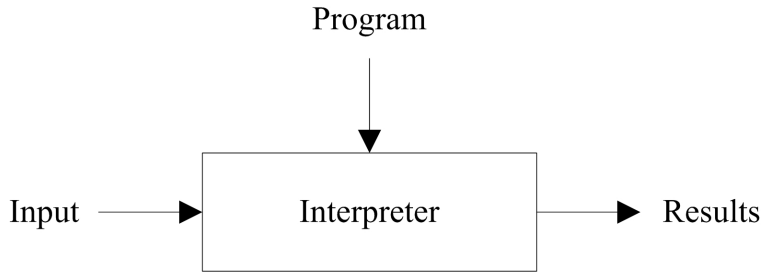
- Intel i386, a complex instruction set computer (CISC) with powerful and complex instructions
- MIPS, as a reduced instruction set computer (RISC) with relatively simple instructions
- Java Virtual Machine (JVM), a virtual machine not necessarily implemented in hardware

Traditionally, a compiler analyzes the input program to produce (or synthesize) the output program

- mapping names to memory addresses, stack frame offsets, and registers,
- generating a linear sequence of machine code instructions, and
- detecting any errors in the program that can be detected in compilation

Compilers

An interpreter executes a high-level language program directly, ie, the high-level program is first loaded into the interpreter, and then executed



Examples of programming languages whose programs may be interpreted directly are the UNIX shell languages, such as bash and csh, Python, and many versions of LISP

Why Study Compilers?

Compilers are larger programs than the ones you have written in your programming courses

Compilers make use of all those things you have learned about earlier: arrays, lists, queues, stacks, trees, graphs, maps, regular expressions and finite state automata, context-free grammars and parsers, recursion and patterns

You learn a lot about the language you are compiling (in our case, Java)

You learn a lot about the target machine (in our case, both the Java Virtual Machine and the MIPS computer)

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

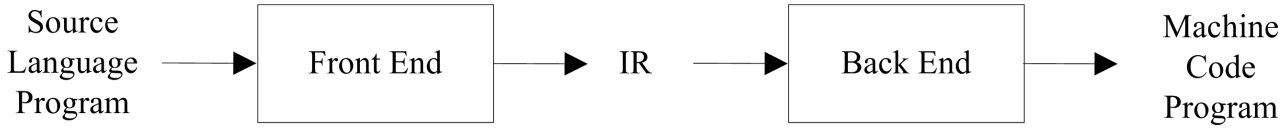
There is a good mix of theory and practice, and each is relevant to the other

The organization of a compiler is such that it can be written in stages, and each stage makes use of earlier stages; compiler writing is a case study in software engineering

Compilers are programs and writing programs is fun

The Phases of Compilation

At the very least, a compiler can be broken into a front end and a back end



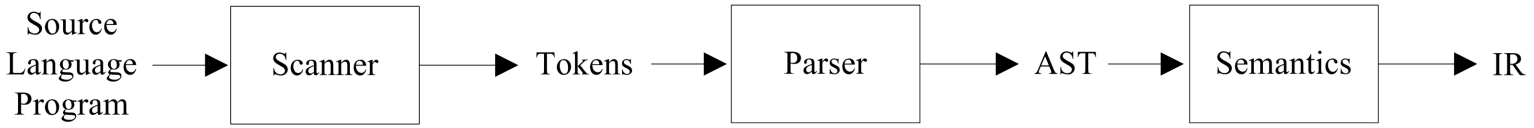
The front end takes as input, a high-level language program, and produces as output an intermediate representation (IR) of that program

The back end then takes the IR of the program as input, and produces the target machine language program

The Phases of Compilation

A compiler's front end analyzes the input program to determine its meaning, and so is source language dependent and target language independent

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



The scanner breaks the input stream of characters to a stream of tokens: identifiers, literals, reserved words, operators, and separators

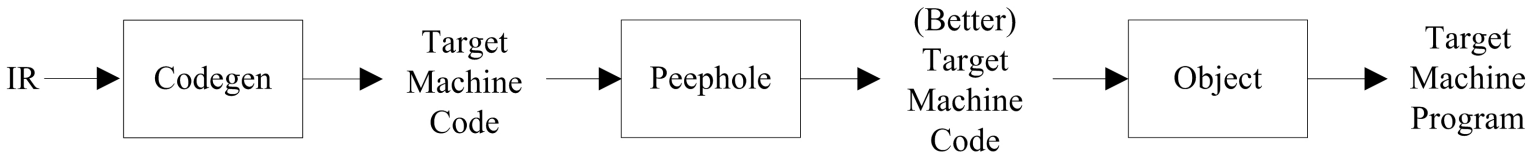
The parser takes a sequence of tokens and parses it against a grammar to produce an abstract syntax tree (AST), which makes the syntax that is implicit in the source program, explicit

The semantics phase does semantic analysis: declares names in a symbol table, looks up names as they are referenced to determine their types, assigns types to expressions, and checks the validity of types; sometimes, a certain amount of storage analysis (such as assigning addresses or offsets to variables) is also done

The Phases of Compilation

A compiler's back end takes the IR and produces (synthesizes) a target machine program having the same meaning, and so is target language dependent and source language independent

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



The code generation phase chooses what target machine instructions to generate, making use of information collected in earlier phases

The peephole phase scans through the generated instructions looking locally for wasteful instruction sequences such as branches to branches and unnecessary load/store pairs

The object phase links together any modules produced in code generation and it constructs a single machine code executable program

The Phases of Compilation

Sometimes, a compiler will have an optimizer ("middle end"), which sits between the front end and the back end

The purpose of the optimizer is both to improve the IR program and to collect information that the back end may use for producing better code

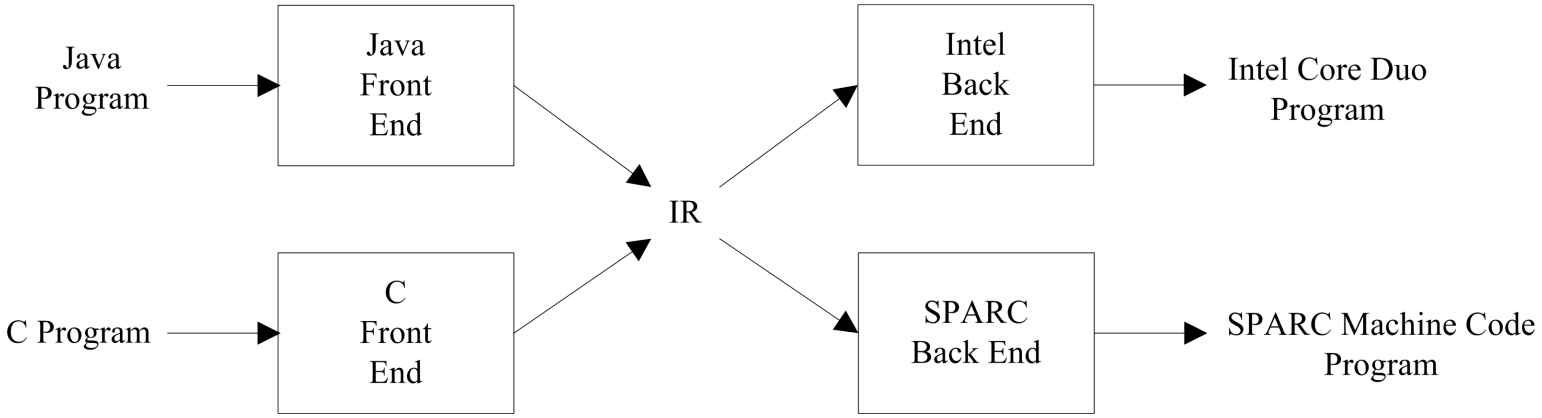
The optimizer might do any number of the following

- Organize the program into what are called basic blocks: blocks of code from which there are no branches out and into which there are no branches
- From the basic block structure, one may then compute next-use information for determining the lifetimes of variables, and gather loop information
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` if we know `x` has the value `4`)
- Loop information is useful for pulling loop invariants out of loops and for strength reduction (for example, replacing multiplication operations by addition operations)

The Phases of Compilation

There are several advantages to separating the front end from the back end:

1. Decomposition reduces complexity; it's easier to understand (and implement) the smaller programs
2. Decomposition makes it possible for several individuals or teams to work concurrently on separate parts, reducing the overall implementation time
3. Decomposition permits a certain amount of code re-use; for example, once one has written a front end for Java and a back end for the Intel Core Duo, one need only write a new C front end to get a C compiler, and one need only write a single SPARC back end to re-target both compilers to the SPARC architecture



The Phases of Compilation

The Java compiler, the program invoked when one types, for example

```
$ javac MyProgram.java
```

produces a `MyProgram.class` file, a byte code program suitable for execution on the JVM

To execute the `MyProgram.class` file, one types

```
$ java MyProgram
```

which effectively interprets the JVM program

In this course, we'll study the implementation of a compiler called *j--* to compile a (non-trivial) subset of Java, which we also call *j--*

In the first instance, we'll target the JVM

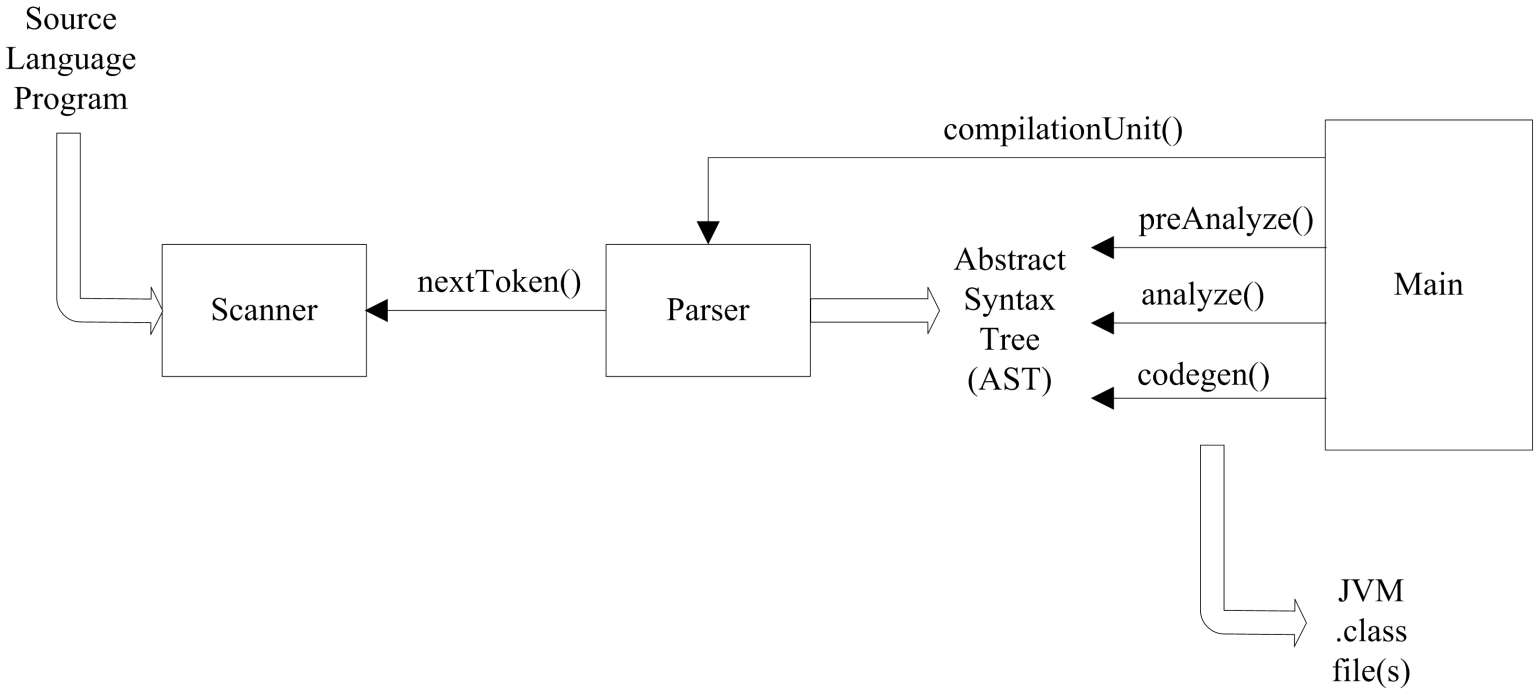
Targeting the JVM is deficient in one respect: one does not learn about register allocation because the JVM is a stack-based architecture and has no registers

To remedy this deficiency, we'll also study the compilation of JVM code to code for the MIPS machine, which is a register-based architecture; in doing this, we face the challenge of mapping possibly many variables to a limited number of fast registers

Overview of the j-- to JVM Compiler

Our source language, *j--*, is a proper subset of the Java programming language; it is an object-oriented programming language, supporting classes, methods, fields, message expressions, and a variety of statements, expressions and primitive types

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



Overview of the j-- to JVM Compiler

The scanner supports the parser, scanning tokens from the input stream of characters comprising the source language program

For example, the scanner breaks down the following source language `HelloWorld` program

```
import java.lang.System;

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

into atomic tokens such as `import`, `java`, `.`, `lang`, `.`, `System`, `;`, and so on

Some tokens such as `java`, `HelloWorld`, and `main` are `IDENTIFIER` tokens, carrying along their images as attributes

Some tokens such as `import`, `public`, and `class` are reserved words having the unique names such as `IMPORT`, `PUBLIC`, and `CLASS`

Operators and separators such as `+` and `;` have distinct names such as `PLUS` and `SEMI`

Literals such as `"Hello, World!"` comprise a single token such as `STRING_LITERAL`

Comments are scanned and ignored altogether

Overview of the j-- to JVM Compiler

The parsing of a *j--* program and the construction of its abstract syntax tree (AST) is driven by the language's syntax, and so is said to be syntax directed

In the first instance, our parser is hand-crafted from the *j--* grammar, to parse *j--* programs by a technique known as recursive descent

For example, consider the following grammatical rule describing the syntax for a compilation unit

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

To parse a compilation unit using the recursive descent technique, one would write a method, call it `compilationUnit()`

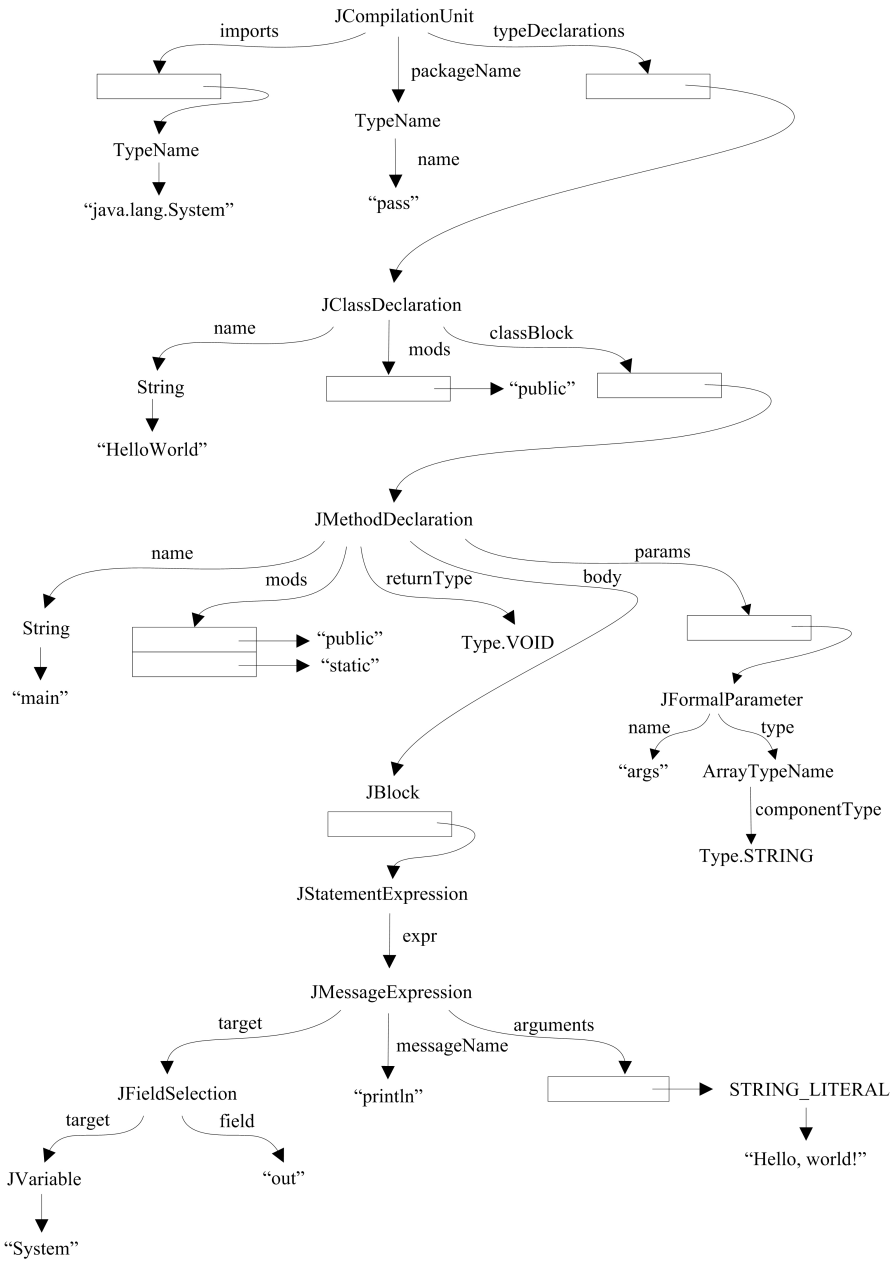
```
    public JCompilationUnit compilationUnit() {
        int line = scanner.token().line();
        TypeName packageName = null; // Default
        if (have(PACKAGE)) {
            packageName = qualifiedIdentifier();
            mustBe(SEMI);
        }
```

Overview of the j-- to JVM Compiler

```
        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(scanner.fileName(), line, packageName,
                imports, typeDeclarations);
    }
```

Overview of the j-- to JVM Compiler

An AST for the `HelloWorld` program



Overview of the j-- to JVM Compiler

As in Java, *j--* names and values have types, and since *j--* (like Java) is statically typed, its compiler 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`), and `Member` (wraps `java.lang.reflect.Member`)

There are places where we want to denote a type by its name, before its type is known or defined, and for this we use `TypeName` and (because we need array types) `ArrayTypeName`

The *j--* compiler maintains a symbol table (a singly-linked list of `Context` objects) in which it declares names; each object in the symbol table represents some area of scope and contains a mapping from names to definitions

A `CompilationUnitContext` object represents the scope comprising the program, ie, the entire compilation unit

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 and, by extension, constructors

Overview of the j-- to JVM Compiler

`preAnalyze()` is a first pass at type checking, which builds that part of the symbol table that is at the top of the AST, to declare both imported types and types introduced by class declarations, and to declare the members in those classes

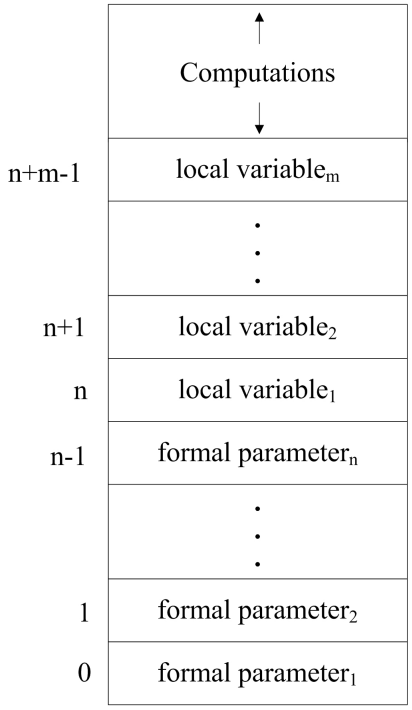
`analyze()` picks up where `preAnalyze()` left off and continues to build the symbol table, decorating the AST with type information and enforcing the *j--* type rules

`analyze()` also performs other important tasks such as type checking, accessibility, member finding, tree rewriting, and storage allocation

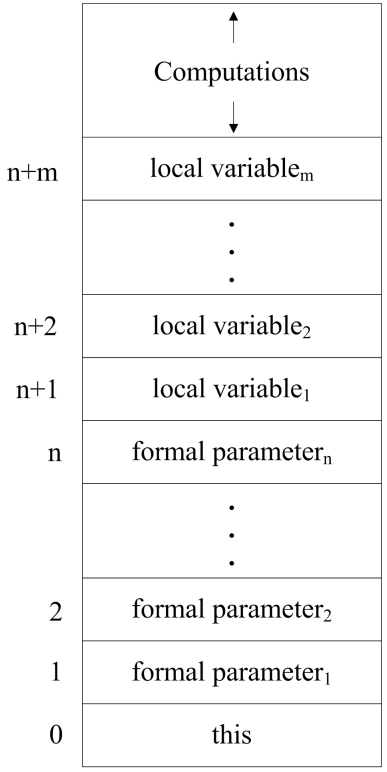
Overview of the j-- to JVM Compiler

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

Each time a method is invoked, the JVM allocates a stack frame, a contiguous block of memory locations on top of the run-time stack; the actual arguments substituted for formal parameters, the values of local variables, and temporary results are all given positions within this stack frame



Stack Frame for a Static Method



Stack Frame for an Instance Method

Overview of the j-- to JVM Compiler

The purpose of `codegen()` is to generate JVM byte code from the AST, based on information computed by `preAnalyze()` and `analyze()`

`codegen()` is invoked by `Main` on the root of the AST, and `codegen()` recursively descends the AST, generating byte code

`CLEmitter` provides an abstraction for the JVM class file, hiding many of the gory details

Here's the implementation of `codegen()` in `JMethodDeclaration` (AST representation for method declarations)

```
    public void codegen(CLEmitter output) {
        output.addMethod(mods, name, descriptor, null, false);
        if (body != null) {
            body.codegen(output);
        }

        // Add implicit RETURN
        if (returnType == Type.VOID) {
            output.addNoArgInstruction(RETURN);
        }
    }
```

The j-- Compiler Source Tree

The zip file `j--.zip` containing the *j*-- distribution may be unzipped into any directory of your choosing; we refer to this directory as `$j`

`$j/j--/src/jminusminus` contains the source files for the compiler, where `jminusminus` is a package, and these include

- `Main.java`, the driver program
- A hand-written scanner (`Scanner.java`) and parser (`Parser.java`)
- `J*.java` files defining classes representing the AST nodes
- `CL*.java` files supplying the back-end code that is used by *j*-- for creating JVM byte code; the most important file amongst these is `CLEmitter.java`, which provides the interface between the front end and back end of the compiler
- `S*.java` files that translate JVM code to SPIM files (SPIM is an interpreter for the MIPS machines symbolic assembly language)
- `j--.jj`, the input file to JavaCC containing the specification for generating (as opposed to hand-writing) a scanner and parser for the *j*-- language; `JavaCCMain`, the driver program that uses the scanner and parser produced by JavaCC
- Other Java files providing representation for types and the symbol table

The j-- Compiler Source Tree

`$j/j--/bin/j--` is a script to run the compiler and has the following command-line syntax

```
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> Max. physical registers (1-18) available for allocation; default = 8
  -d <dir> Specify where to place output files; default = .
```

For example, the *j--* program `$j/j--/tests/pass/HelloWorld.java` can be compiled using `j--` as follows

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

to produce a `HelloWorld.class` file under `pass` folder within the current directory, which can then be run as

```
$ java pass.HelloWorld
```

to produce as output

```
$ Hello, World!
```

The j-- Compiler Source Tree

j-- provides an elaborate framework with which one may add new Java constructs to the *j--* language

As an illustrative example, let's add the division operator to *j--*, which involves modifying the scanner to recognize `/` as a token, modifying the parser to be able to parse division expressions, implementing semantic analysis, and finally, code generation for the division operation

Writing tests

- Place the following file (`Division.java`) under `$j/j--/tests/pass`

```
package pass;

public class Division {
    public int divide(int x, int y) {
        return x / y ;
    }
}
```

Writing tests (contd.)

- Place the following file (`DivisionTest.java`) under `$j/j--/tests/junit`

```
public class DivisionTest extends TestCase {
    private Division division;

    protected void setUp() throws Exception {
        super.setUp();
        division = new Division();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testDivide() {
        this.assertEquals(division.divide(0, 42), 0);
        this.assertEquals(division.divide(42, 1), 42);
        this.assertEquals(division.divide(127, 3), 42);
    }
}
```

- Make the following entry in the `suite()` method of `junit.JMinusMinusTestRunner`

```
TestSuite suite = new TestSuite();
...
suite.addTestSuite(DivisionTest.class);
return suite;
```

Writing tests (contd.)

- Place the following file (`Division.java`) under `$j/j--/tests/fail`

```
package fail;

import java.lang.System;

public class Division {
    public static void main(String[] args) {
        System.out.println('a' / 42);
    }
}
```

Changes to lexical and syntactic grammars

- Add a line describing the division operator to `$j/j--/lexicalgrammar` under the operators section

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

- Add a rule for the division operator to the rule describing multiplicative expressions in the `$j/j--/grammar` file

```
multiplicativeExpression ::= unaryExpression // level 2
                               {(STAR | DIV) unaryExpression}
```

Changes to scanner

- Make the following change in `TokenInfo`

```
enum TokenKind {
    EOF (" < EOF >") ,
    ... ,
    STAR ("*") ,
    DIV ("/") ,
    ...
}
```

- Make the following change in `Scanner`

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

The j-- Compiler Source Tree

Changes to parser

- Define a class called `JDivideOp` in `JBinaryExpression.java` to represent an AST node for the division operator

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

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

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

- To parse expressions involving division operator, modify the `multiplicativeExpression()` method in `Parser.java` as follows

```
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;
}
```

The j-- Compiler Source Tree

Semantic analysis and code generation

- Implement `analyze()` in `JDivideOp` as follows

```
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;
}
```

- Implement `codegen()` in `JDivideOp` as follows

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

To test the changes, run the following command inside the `$j` directory

```
$ ant
```