1. Compilation

1.1 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 an equivalent *target program* in a lower level language such as machine code, which can be executed directly by a computer. This translation is illustrated in Figure 1.1.

Figure 1.1 Compilation

By equivalent, we mean *semantics preserving*: the translation should have the same behavior as the original. This process of translation is called *compilation*.

1.1.1 Programming Languages

A *programming language* is an artificial language, in which a programmer (usually a person) writes a program to control the behavior of a machine, particularly a computer. Of course, a program has an audience other than the computer whose behavior it means to control; other programmers may read a program to understand how it works, or why it causes unexpected behavior. So, it must be designed so as to allow the programmer to precisely specify what the computer is to do in a way that both the computer and other programmers can understand.

Examples of programming languages are Java, C, C++, C# and Ruby. There are hundreds, if not thousands, of different programming languages. But at any one time, a much smaller number are in popular use.

Like a natural language, a programming language is specified in three steps:

1. The tokens, or lexemes are described. Examples are the keyword `if`, the operator `+`, constants such as `4` and `'c'`, and the identifier `foo`. Tokens in a programming language are like words in a natural language.
2. One describes the syntax of programs and language constructs such as classes, methods, statements and expressions. This is very much like the syntax of a natural language but much less flexible.
3. One specifies the meaning, or semantics, of the various constructs. The semantics of various constructs is usually described in English.

1-1

Some programming languages, like Java, also specify various static type rules, which a program and its constructs must obey. These additional rules are usually specified as part of the semantics.

Programming language designers go to great lengths to precisely specify the structure of tokens, the syntax and the semantics. The tokens and the syntax are often described using formal notations, e.g. regular expressions and context-free grammars. The semantics is usually described in a natural language such as English[1]. A good example of a programming language specification is the *Java Language Specification (*Gosling et al, 2005).

1.1.2 Machine Languages

A computer's *machine language* or, equivalently, its *instruction set* is designed so as to be easily interpreted by the computer itself. A machine language program consists of a sequence of instructions and operands, usually organized so that each instruction and each operand occupies one or more bytes and so is easily accessed and interpreted. On the other hand, people are not expected to read a machine code program[2]. A machine's instruction set and its behavior is often referred to as its *architecture*.

Examples of machine languages are the instruction sets for both the Intel i386 family of architectures, and the MIPS computer. The Intel i386 is known as a complex instruction set computer (CISC) because many of its instructions are both powerful and complex. The MIPS is known as a reduced instruction set computer (RISC) because its instructions are relatively simple; it may often require several RISC instructions to carry out the same operation as a single CISC instruction. RISC machines often have at least 32 registers, while CISC machines often have as few as 8 registers. Fetching data from, and storing data in, registers is much faster than accessing memory locations because registers are part of the computer processing unit (CPU) that does the actual computation. For this reason, a compiler tries to keep as many variables and partial results in registers as possible.

Another example is the machine language for Oracle's Java Virtual Machine (JVM) architecture. The JVM is said to be *virtual* not because it doesn't exist but because it is not necessarily implemented in hardware[3]; rather it is implemented as a software program. We discuss the implementation of the JVM in greater detail in Chapter 7. But as compiler writers, we are interested in its instruction set rather than its implementation.

---

[1] Although formal notations have been proposed for describing both the type rules and semantics of programming languages, these are not popularly used.
[2] But one can. Tools often exist for displaying the machine code in mnemonic form, which is more readable than a sequence of binary byte values. The Java toolset provides `javap` for displaying the contents of class files.
[3] Although Oracle has experimented with designing a JVM implemented in hardware, it never took off. Computers designed for implementing particular programming languages rarely succeed.

1-2

Hence the compiler: the compiler transforms a program written in the high-level programming language into a semantically equivalent machine code program.

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.

Compilation is often contrasted with *interpretation*, where the high-level language program is executed directly. That is, 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, Forth, and many versions of LISP.
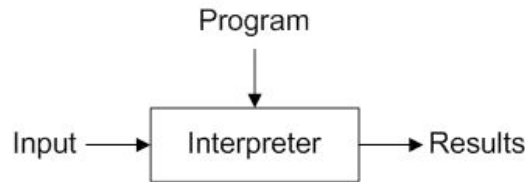

Figure 1.2 Interpretation

One might ask, "Why not interpret all programs directly?" There are two answers.

First is performance. Native machine code programs run faster than interpreted high-level language programs. To see why this is so, consider what an interpreter must do with each statement it executes: it must parse and analyze the statement to decode its meaning *every time* it executes that statement; a limited form of compilation is taking place for every execution of every statement. It is much better to translate all statements in a program to *native code* just once, and execute that.[4]

Second is secrecy. Companies often want to protect their investment in the programs that

---

[4]   Not necessarily always; studies have shown that just-in-time compilation, where a method is translated the first time it is invoked, and then cached, or hotspot compilation, where only code that is interpreted several times is compiled and cached, can provide better performance. Even so, in both of these techniques, programs are partially compiled to some intermediate form such as Oracle's Java Virtual Machine (JVM), or Microsoft's Common Language Runtime (CLR). The intermediate forms are smaller and space can play a role in run-time performance. We discuss just-in-time compilation and hotspot compilation in Chapter 9.

1-3

they have paid programmers to write. It is more difficult (albeit not impossible) to discern the meaning of machine code programs than of high-level language programs.

But, compilation is not always suitable. The overhead of interpretation doesn't always justify writing (or, buying) a compiler. An example is the UNIX Shell (or Windows shell) programming language. Programs written in shell script have a simple syntax and so are easy to interpret; moreover, they are not executed often enough to warrant compilation. And, as we have stated, compilation maps names to addresses; some dynamic programming languages (LISP is the class example, but there are a myriad of newer dynamic languages) depend on keeping names around at run time.

1.2 Why Should We Study Compilers?

So why study compilers? Haven't all the compilers been written? There are several reasons for studying compilers.

The Top Ten Reasons for Studying Compilers:

1. Compilers are larger programs than the ones you've written in your programming courses. It is good to work with a program that is like the size of the programs you will be working on when you graduate.
2. 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. It is fun to use all of these in a real program.
3. You learn about the language you are compiling (in our case, Java).
4. You learn a lot about the target machine (in our case, both the Java Virtual Machine and the MIPS computer).
5. Compilers are still being written for new languages and targeted to new computer architectures. Yes, there are still compiler-writing jobs out there.
6. Compilers are finding their way into all sorts of applications including games, phones and entertainment devices.
7. XML. Programs that process XML use compiler technology.
8. There is a mix of theory and practice, and each is relevant to the other.
9. The organization of a compiler is such that it can be written in stages, and each stage makes use of earlier stages. So, compiler writing is a case study in software engineering.
10. Compilers are programs. And writing programs is fun.

1.3 How Does a Compiler Work? The Phases of Compilation

A compiler is usually broken down into several phases – components, each of which performs a specific sub-task of compilation. At the very least, a compiler can be broken into a front end and a back end.
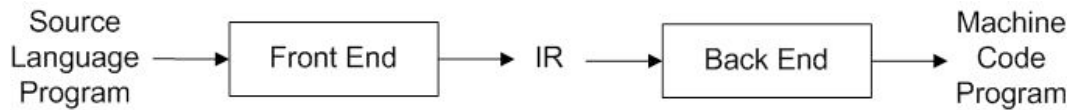
1-4

Figure 1.3 A Compiler: Analysis and Synthesis

The front end takes as input, a high-level language program, and produces as output a representation (another translation) of that program in some intermediate language that lies somewhere between the source language and the target language. We call this the intermediate representation (IR). The back end then takes this intermediate representation of the program as input, and produces the target machine language program.

1.3.1 The Front End

A compiler's front end,

- is that part of the compiler that analyzes the input program for determining its meaning, and so
- is source language dependent (and target machine, or target language independent); moreover it
- can be further decomposed into a sequence of analysis phases such as that illustrated in Figure 1.4.

Figure 1.4 The Front End: Analysis

The Scanner is responsible for breaking the input stream of characters to a stream of tokens: identifiers, literals, reserved words, (one-, two-, three-, and 4-character) operators and separators.

The Parser is responsible for taking this sequence of lexical tokens and parsing 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 is responsible for declaring names in a symbol table, looking up names as they are referenced for determining their types, assigning types to expressions, and checking the validity of types. Sometimes, a certain amount of storage analysis is also done, for example assigning addresses or offsets to variables (as we do in our *j--* compiler). When a programming language allows one to refer to a name that is declared later on in the program, the semantics phase must really involve at least two phases (or two passes over the program).

1-5

1.3.2 The Back End

A compiler's back end,

- is that part of the compiler that takes the IR and produces (synthesizes) a target machine program having the same meaning, and so
- is target language dependent (and source language independent); moreover it
- may be further decomposed into a sequence of synthesis phases such as that illustrated in Figure 1.5.
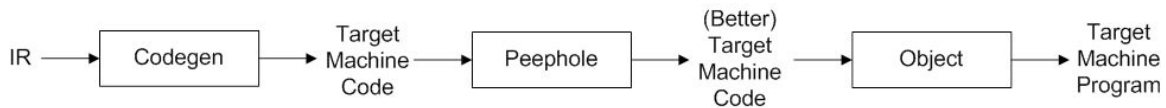

Figure 1.5 The Back End: Synthesis

The code generation phase is responsible for choosing what target machine instructions to generate. It makes use of information collected in earlier phases.

The peephole phase implements a *peephole optimizer*, which scans through the generated instructions looking locally for wasteful instruction sequences such as branches to branches and unnecessary load/store pairs (where a value is loaded onto a stack or into a register and then immediately stored back at the original location).

Finally, the object phase, links together any modules produced in code generation and it constructs a single machine code executable program.

1.3.3 The "Middle End"

Sometimes, a compiler will have an optimizer, which sits between the front end and the back end. Because of its location in the compiler architecture, we often call it the "middle end", with a little tongue-in-cheek.
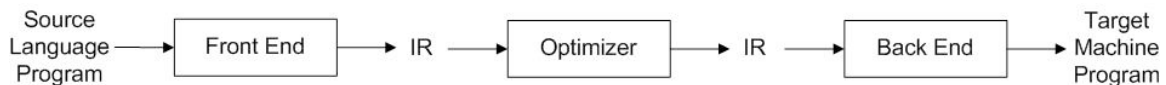

Figure 1.6 The "Middle End": Optimization

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:

- It might 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.

1-6

- From the basic block structure, one may then compute next-use information for determining the lifetimes of variables (how long a variable retains its value before it is redefined by assignment), and loop identification.
- Next-use information is useful for eliminating common sub-expressions and constant folding (for example, replacing `x + 5` by `9` when we know `x` has the value `4`). It may also be used for register allocation (deciding what variables or temporaries should be kept in registers and what values to "spill" from a register for making room for another).
- Loop information is useful for pulling loop-invariants out of loops and for strength reduction, for example, replacing multiplication operations by (equivalent but less expensive) addition operations.

An optimizer might consist of just one phase or several phases, depending on the optimizations performed. These and other possible optimizations are discussed more fully in Chapters 7 and 8.

1.3.4 Advantages to Decomposition

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

1. Decomposition reduces complexity. It is 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 re-use.[5] 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 retarget both compilers to the Oracle SPARC architecture. Figure 1.7 illustrates how this re-use gives us four compilers for the price of two.

---

[5]   This depends on a carefully designed IR. We can't count the number of times we've written front-ends with the intention of re-using them, only to have to re-write them for new customers (with that same intention!). Realistically, one ends up re-using designs more often than code.
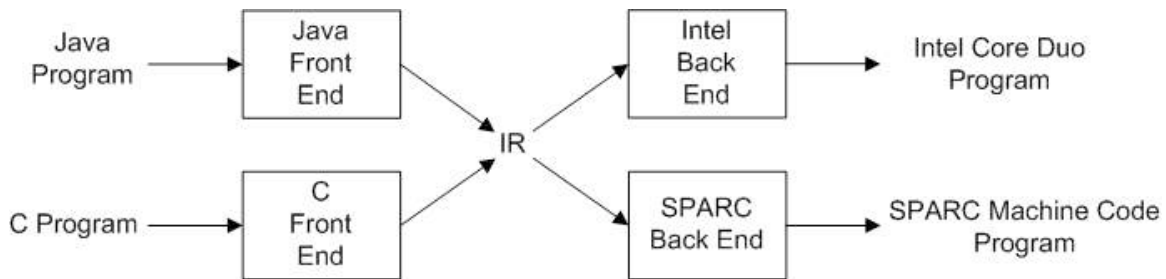
Figure 1.7 Re-use through Decomposition

Decomposition was certainly helpful to us, the authors, in writing the *j--* compiler as it allowed us better organize the program and to work concurrently on distinct parts of it.

1.3.5 Compiling to a Virtual Machine: New Boundaries

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

```
javac MyProgram.java
```

produces a .class file called **MyProgram.class**, that is, a byte-code[6] program suitable for execution on a Java Virtual Machine (Java VM). The source language is Java; the target machine is the JVM. To execute this .class file, one types

```
java MyProgram
```

which effectively interprets the JVM program. The JVM is an interpreter, which is implemented based on the observation that almost all programs spend most of their time in a small part of their code. The JVM monitors itself to identify these "hot spots" in the program it is interpreting and it compiles these critical methods to native code; this compilation is accompanied by a certain amount of in-lining: the replacement of method invocations by the method bodies. The native code is then executed, or interpreted on the native computer (an interpreter implemented in hardware[7]). Thus the JVM byte code might be considered an IR, with the Java "compiler" acting as the front-end, and the JVM acting as the back end, targeted to the native machine on which it is running.

The IR analogy to the byte-code makes even more sense in Microsoft's Common Language Runtime (CLR) architecture used in implementing its .Net tools. Microsoft has written compilers (or front ends) for Visual Basic, C++, C# and J++ (a variant of Java), all of which produce byte-code targeted for a common architecture (the CLR). Using a technique called *just-in-time (JIT) compilation*, the CLR compiles each method to native code, and caches that native code when that method is first invoked. Third parties have implemented other front-end compilers for other programming languages, taking

---

[6] "Byte-code" because the program is represented as a sequence of byte instructions and operands (and operands occupy several bytes).
[7] If not in microcode!

1-8

Draft, © 2008-2011 by Bill Campbell, Swami Iyer and Bahar Akbal-Delibaş

advantage of the existing JIT compilers.

In this textbook, we compile a (non-trivial) subset of Java, which we call *j--*. In the first instance, we target the Oracle JVM. So in a sense, this compiler is a front end. Nevertheless, our compiler implements many of those phases that are traditional to compilers and so it serves as a reasonable example for an introductory compilers course.

The experience in writing a compiler targeting the JVM is deficient in one respect: one doesn't learn about register allocation because the JVM is a stack-based architecture and has no registers.

1.3.6 Compiling JVM Code to a Register Architecture

To remedy this deficiency, we (beginning in Chapter 7) discuss 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.

One might ask, " Why don't we simply translate j-- programs to MIPS programs?" after all, C language programs are always translated to native machine code.

The strategy of providing an intermediate virtual machine code representation for one's programs has several advantages:

1. Byte code, such as JVM code or Microsoft's CLR code is quite compact. It takes up less space to store (and less memory to execute) and it is more amenable to transport over the Internet. This latter aspect of JVM code made Java applets possible and accounts for much of Java's initial success.
2. Much effort has been invested in making interpreters like the JVM and the CLR run quickly; their just-in-time compilers are highly optimized. One wanting a compiler for any source language need only write a front-end compiler that targets the virtual machine to take advantage of this optimization.
3. Implementers claim, and performance tests support, that hot-spot interpreters, which compile to native code only those portions of a program that execute frequently, actually run faster than programs that have been fully translated to native code. Caching behavior might account for this improved performance.

Indeed, the two most popular platforms (Oracle's Java platform and Microsoft's .NET architecture) follow the strategy of targeting virtual, stack-based, byte-code architecture in the first instance, and then employing either just-in-time compilation or HotSpot compilation for implementing these "interpreters".

1.4 An Overview of the j-- to JVM Compiler

Our source language, *j--*, is a proper subset of the Java programming language. It has about half the syntax of Java; that is, its grammar that describes the syntax is about half the size of that describing Java's syntax. But *j--* is a non-trivial, object-oriented

1-9

programming language, supporting classes, methods, fields, message expressions, and a variety of statements, expressions and primitive types. *J--* is more fully described in Appendix B.

Our *j--* compiler is organized in an object-oriented fashion. To be honest, most compilers are not organized in this way. Nor are they written in languages like Java, but in lower-level languages such as C and C++ (principally for better performance). As the previous section suggests, most compilers are written in a procedural style. Compiler writers have generally bucked the object-oriented organizational style and have relied on the more functional organization described in section 1.3.

Even so, we decided to structure our *j--* compiler on object-oriented principles. We chose Java as the implementation language because that is the language our students know best and the one (or one like it) in which you will program when you graduate. Also, you are likely to be programming in an object-oriented style.

It has many of the components of a traditional compiler, and its structure is not necessarily novel. Nevertheless, it serves our purposes:

- We learn about compilers.
- We learn about Java. *j--* is a non-trivial subset of Java. The *j--* compiler is written in Java.
- We work with a non-trivial object-oriented program.

1.4.1 j-- Compiler Organization
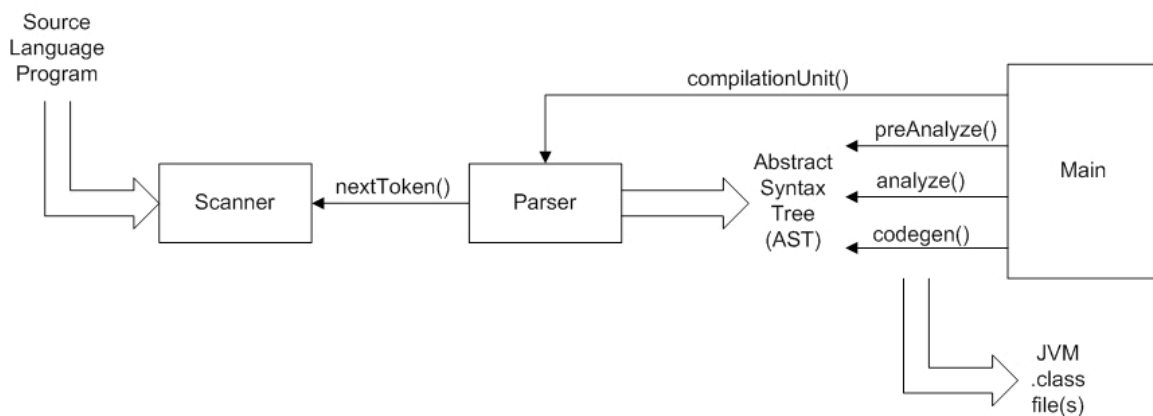
Our compiler's structure is illustrated in Figure 1.8.

Figure 1.8 The *j--* Compiler

The entry point to the *j--* compiler is **Main**[8]. It reads in a sequence of arguments, and then goes about creating a Scanner object, for scanning tokens, and a Parser object for parsing the input source language program and constructing an abstract syntax tree (AST).

Each node in the abstract syntax tree is an object of a specific type, reflecting the underlying linguistic component or operation. For example, an object of type **JCompilationUnit** sits at the root (the top) of the tree for representing the program being compiled. It has sub-trees representing the package name, list of imported types, and list of type (i.e. class) declarations. An object of type **JMultiplyOp** in the AST, for example, represents a multiplication operation. Its two sub-trees represent the two operands. At the leaves of the tree, one finds **JVariable** objects and objects representing constant literals.

Each type of node in the AST defines three methods, each of which performs a specific task on the node, and recursively on its sub-trees:
1. **preAnalyze(Context context)** is defined only for the types of nodes that appear near the top of the AST. Pre-analysis deals with declaring imported types, defined class names, and class member headers (method headers and fields). This is required since method bodies may make forward references to names declared later on in the input. The context argument is a string of **Context** (or subtypes of **Context**) objects representing the compile-time symbol table of declared names and their definitions.
2. **analyze(Context context)** is defined over all types of AST nodes. When invoked on a node, this method declares names in the symbol table (context), checks types (looking up the types of names in the symbol table), and converts local variables to offsets in a method's run-time local stack frame where the local variables reside.
3. **codegen(CLEmitter output)** is invoked for generating the Java Virtual Machine (JVM) code for that node, and is applied recursively for generating code for any sub-trees. The output argument is a **CLEmitter** object, an abstraction of the output .class file.

So, once **Main** has created the Scanner and Parser,

1. **Main** sends a **compilationUnit()** message to the Parser, causing it to parse the program by a technique known as recursive descent, and to produce an AST.
2. **Main** then sends the **preAnalyze()** message to the root node (an object of type **JCompilationUnit**) of the AST. **preAnalyze()** recursively descends the tree down to the class member headers for declaring the types and the class members in the symbol table, context.
3. **Main** then sends the **analyze()** message to the root **JCompilationUnit** node,

---

[8]   This and other classes related to the compiler are part of the **jminusminus** package under **$j/j--/src**.

and `analyze()` recursively descends the tree all the way down to its leaves, declaring names and checking types.
4. `Main` then sends the `codegen()` message to the root `JCompilationUnit` node, and `codegen()` recursively descends the tree all the way down to its leaves, generating JVM code. At the start of each class declaration, `codegen()` creates a new `CLEmitter` for representing a target .class file for that class; at the end of each class declaration, `codegen()` writes out the code to a .class file on the file system.
5. The compiler is then done with its work. If errors occur in any phase of the compilation process, the phase attempts to run to completion (finding any additional errors) and then the compilation process halts.

In the next sections, we briefly discuss how each phase does its work. As this is just an overview and a preview of what's to come in subsequent chapters, it is not important that one understand everything at this point. Indeed, if you understand just 15%, that is fine. The point to this overview is to let you know where stuff is. We have the rest of the text to understand how it all works!

1.4.2 The Scanner

The Scanner is a slave to the Parser. Its purpose is to scan tokens from the input stream of characters comprising the source language program. For example, consider 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!");
    }
}
```

The Scanner breaks the program text into atomic tokens. For example, it recognizes each of `import`, `java`, `.`, `lang`, `.`, `System`, `;` as being distinct tokens.

Some tokens, such as `java`, `HelloWorld`, and `main` are identifiers. The Scanner categorizes theses tokens as `IDENTIFIER` tokens. The Parser uses these category names to identify the kinds of incoming tokens. `IDENTIFIER` tokens carry along their images as attributes; for example the first `IDENTIFIER` in the above program has `java` as its image. Such attributes are used in semantic analysis.

Some tokens are reserved words, each having its unique name in the code. For example, `import`, `public` and `class` are reserved word tokens having the names `IMPORT`,

1-12

PUBLIC and **CLASS**. Operators and separators also have distinct names. For example, the separators, **. , ; , { , } , [** and **]** have the token names **DOT**, **SEMI**, **LCURLY**, **RCURLY**, **LBRACK**, and **RBRACK** respectively.

Others are literals; for example, the string literal "Hello, World!" comprises a single token. The Scanner calls this a **STRING_LITERAL**.

Comments are scanned over and ignored altogether. As important as some comments are to a person who is trying to understand a program[9], they are irrelevant to the compiler.

Now, the Scanner doesn't first break down the input program text into a sequence of tokens. Rather, it scans each token on demand; each time the Parser needs a subsequent token, it sends the **nextToken()** message to the Scanner, which then returns the token id and any image information.

The Scanner program is discussed in greater detail in Chapter 2.

1.4.3 The Parser

The parsing of a *j--* program and the construction of its AST is driven by the language's syntax, and so is said to be *syntax directed*. In the first instance, our Parser is handcrafted 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
```

This rule says that a compilation unit consists of
* an optional package clause (the brackets **[]** bracket optional clauses),
* followed by zero or more import statements (the curly brackets **{}** bracket clauses that may appear zero or more times),
* followed by zero or more type declarations (in *j--*, these are only class declarations),
* followed by an end of file (**EOF**).

The tokens **PACKAGE**, **SEMI**, **IMPORT** and **EOF** are returned by the Scanner.

To parse a compilation unit using the recursive descent technique, one would write a

---

[9]   But we know some who swear by the habit of stripping out all comments before reading a program for fear that those comments might be misleading. When programmers modify code, they often forget to update the accompanying comments.
1-13

method, call it **compilationUnit()**, which does the following.

1. If the next (the first, in this case) incoming token were **PACKAGE**, would scan it (advancing to the next token), invoke a separate method called **qualifiedIdentifier()** for parsing a qualified identifier, and then we must scan a **SEMI** (and announce a syntax error if the next token weren't a **SEMI**).
2. While the next incoming token is an **IMPORT**, scan it and invoke **qualifiedIdentifier()** for parsing the qualified identifier, and then we must again scan a **SEMI**. We save the (imported) qualified identifiers in a list.
3. While the next incoming token is *not* an **EOF**, invoke a method called **typeDeclaration()** for parsing the type declaration (in *j*-- this is only a class declaration), and we must scan a **SEMI**. We save all of the ASTs for the type declararations in a list.
4. We must scan the **EOF**.

Here's the Java code for **compilationUnit**, taken directly from **Parser**.

```
public JCompilationUnit compilationUnit(String classDir) {
    int line = scanner.token().line();
    TypeName packageName = null; // Default
    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(scanner.fileName(), line,
        packageName, imports, typeDeclarations);
}
```

In the Parser, **see()** is a Boolean method that looks to see whether or not its argument matches the next incoming token. Method **have()** is the same, but has the side-effect of scanning past the incoming token when it does match its argument. Method **mustBe()** requires that its argument match the next incoming token, and raises an error if it does not.

Of course, the method **typeDeclaration()** recursively invokes additional methods for parsing the **HelloWorld** class declaration; hence the technique's name: recursive

1-14

descent. Each of these parsing methods produces an AST constructed from some particular type of node. For example; at the end of `compilationUnit()` a `JCompilationUnit` node is created for encapsulating any package name (none here), the single import (having its own AST), and a single class declaration (an AST rooted at a `JClassDeclaration` node).

Parsing in general, and recursive descent in particular, are discussed more fully in Chapter 3.

1.4.4 The AST

An abstract syntax tree (AST) is just another representation of the source program. But it is a representation that is much more amenable to analysis. We know how to do tree-walking. And the AST makes explicit that syntactic structure, which is implicit in the original source language program. The AST produced for our `HelloWorld` program, from section 1.4.2 is illustrated in Figure 1.9.
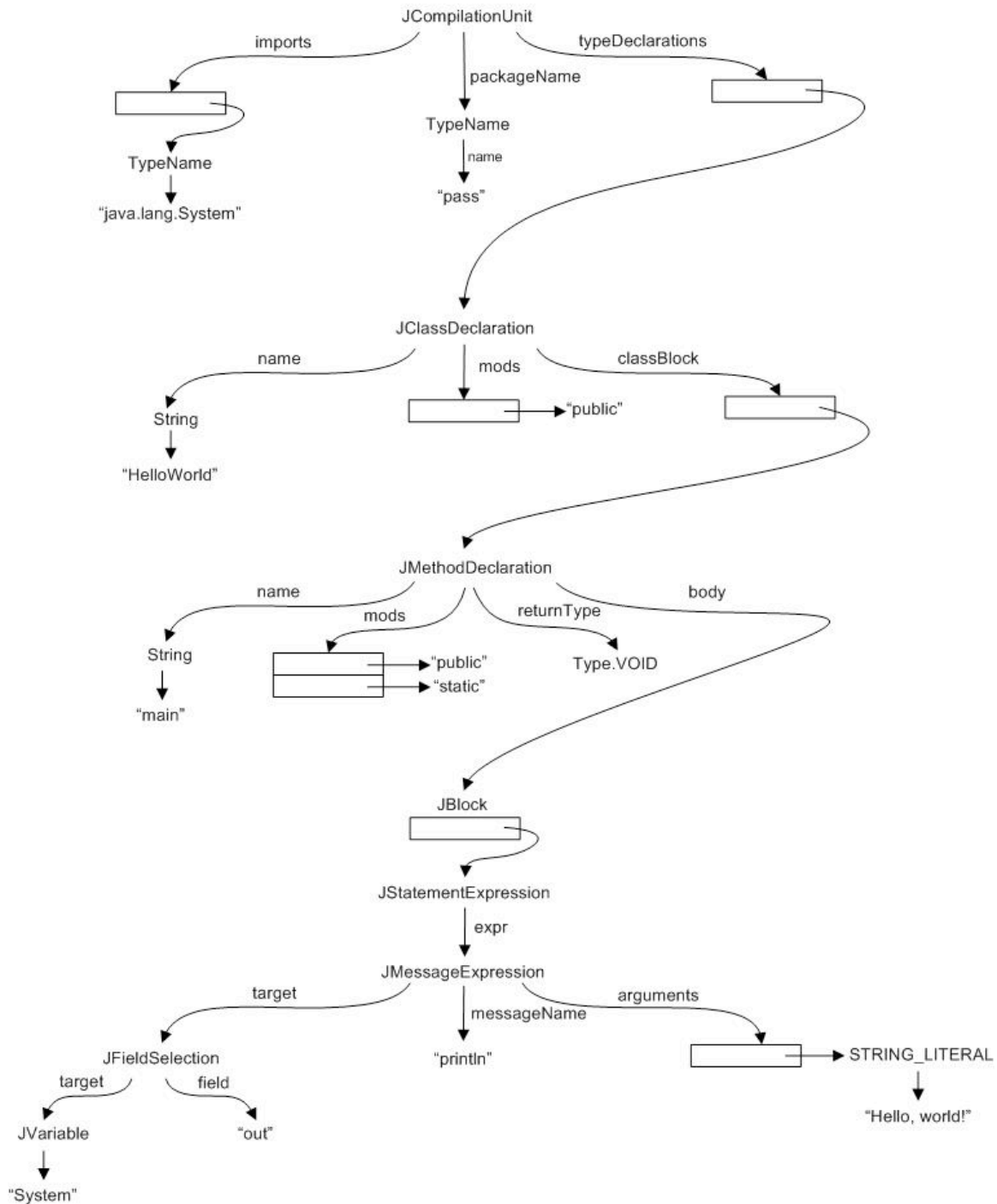
Figure 1.9 An AST for the HelloWorld Program

All classes in the *j--* compiler that are used to represent nodes in the AST extend the abstract class **JAST** and have names beginning with the letter **J**. Each of these classes implements the three methods required for compilation:

1. **preAnalyze()** for declaring types and class members in the symbol table;

Draft, © 2008-2011 by Bill Campbell, Swami Iyer and Bahar Akbal-Delibaş

2. `analyze()` for declaring local variables and typing all expressions; and
3. `codegen()` for generating code for each sub-tree.

We discuss these methods briefly below, and in greater detail later on in this book. But before doing that, we must first briefly discuss how we build a symbol table and use it for declaring (and looking up) names and their types.

1.4.5 Types

As in Java, *j--* names and values have types. A type indicates how something can behave. A `boolean` behaves differently from an `int`; a `Queue` behaves differently from a `Hashtable`. Because *j--* (like Java) is *statically typed*, its compiler must determine the types of all names and expressions. So we need a representation for types.

Now, Java already has a representation for its types: objects of type `java.lang.Class` from the Java API. Since *j--* is a subset of Java, why not use class `Class`? The argument is more compelling because *j--*'s semantics dictate that it may make use of classes from the Java API, so its type representation must be compatible with Java's.

But, since we want to define our own functionality for types, we encapsulate the `Class` objects within our own class called `Type`. Likewise, we encapsulate `java.lang.reflect.Method`, `java.lang.reflect.Constructor`, `java.lang.reflect.Field`, and `java.lang.reflect.Member` within our own classes, `Method`, `Constructor`, `Field` and `Member` respectively[10]. And we define a sufficiently rich set of operations on these representational classes.

There are places, for example in the Parser, where we want to denote a type by its name, before that types is known or defined. For this we introduce `TypeName` and (because we need arrays of these things) `ArrayTypeName`. During the analysis phase of compilation, these type denotations are *resolved*: they are looked up in the symbol table and replaced by the actual `Type`s they denote.

1.4.6 The Symbol Table

During semantic analysis, the compiler must construct and maintain a symbol table in which it declares names. Because *j--* (like Java) has a nested scope for declared names, this symbol table must behave like a pushdown stack.

In the *j--* compiler, this symbol table is represented as a singly linked list of `Context` objects, that is objects whose types extend the `Context` class. Each object in this list represents some area of scope and contains a mapping from names to definitions. Every

---

[10]   These private classes are defined in the `Type.java` file, together with the public class `Type`. In the code tree, we have chosen to put many private classes in the same file in which their associated public class is defined.

context object maintains three pointers: one to the object representing the surrounding context, one to the object representing the compilation unit context (at the root), and one to the enclosing class context.

For example, there is a **CompilationUnitContext** object for representing the scope comprising the program, that is, the entire compilation unit. There is a **ClassContext** object for representing the scope of a class declaration. The **ClassContext** has a reference to the defining class type; this is used to determine where we are (that is, in which class declaration the compiler is in) for settling issues such as accessibility. There is a **MethodContext** (a subclass of **LocalContext**) for representing the scopes of methods and, by extension, constructors. Finally, a **LocalContext** represents the scope of a block, including those blocks that enclose method bodies. Here, local variable names are declared and mapped to **LocalVariableDefn**s.

1.4.7 preAnalyze() and analyze()

PreAnalyze() is a first pass at type checking. Its purpose is to build 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 declared in those classes. This first pass is necessary for declaring names that may be referenced before they are defined.

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

The **analyze()** phase performs other important tasks:

- Type checking: **analyze()** computes the type for every expression, and it checks its type when a particular type is required.
- Accessibility: **analyze()** enforces the accessibility rules (expressed by the modifiers **public**, **protected**, and **private**) for both types and members.
- Member finding: **analyze()** finds members (messages in message expressions, based on signature, and fields in field selections) in types. Of course, only the compile-time member name is located; polymorphic messages are determined at run time.
- Tree rewriting: **analyze()** does a certain amount of AST (sub) tree rewriting. Implicit field selections (denoted by identifiers that are fields in the current class) are made explicit and field and variable initializations are rewritten as assignment statements after the names have been declared.

1.4.8 Stack Frames

Analyze() does a little storage allocation. It allocates positions in the method's current stack frame for formal parameters and (other) local variables.

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

1-18

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 frames for both a static method and for an instance method are illustrated in Figure 1.10.
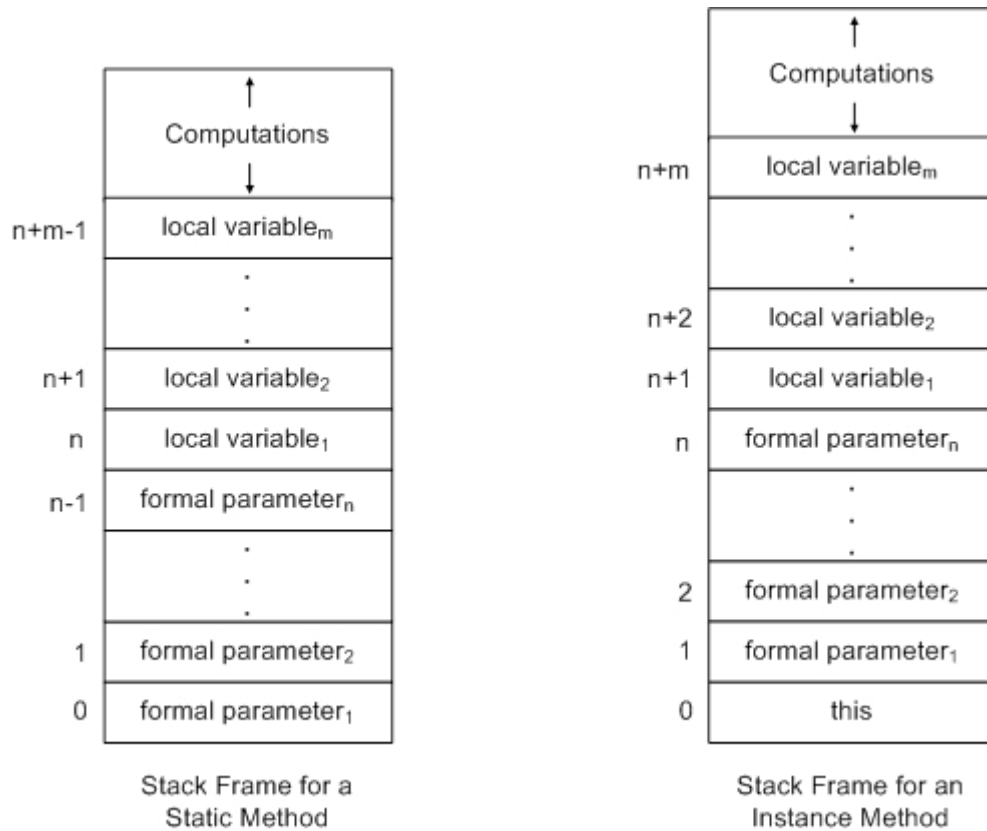

Figure 1.10 Run-time Stack Frames in the JVM

In both frames, locations are set aside for *n* formal parameters and *m* local variables; *n, m* or both may be 0. In the stack frame for a *static* method, these locations are allocated at offsets beginning at 0. But in the invocation of an instance method, the instance itself, i.e. **this**, must be passed as an argument, so in an *instance* method's stack frame, location 0 is set aside for **this**, and parameters and local variables are allocated offset positions starting at 1. The areas marked "computations" in the frames are memory locations set aside for run-time stack computations within the method invocation.

While the compiler cannot predict how many stack frames will be pushed onto the stack (that would be akin to solving the halting problem), it can compute the offsets of all formal parameters and local variables, and compute how much space the method will need for its computations.

1-19

1.4.9 codegen()

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**'s sending the **codegen()** message to the root of the AST, and **codegen()** recursively descends the AST, generating byte code.

The format of a JVM class file is rather arcane. For this reason, we have implemented a tool, **CLEmitter** (and its associated classes), to ease the generation of types (e.g. classes), members and code. **CLEmitter** may be considered an abstraction of the JVM class file; it hides many of the gory details. **CLEmitter** is described further in Appendix D.

Main creates a new **CLEmitter**. **JClassDeclaration** adds a new class, using **addClass()**. **JFieldDeclaration** writes out the fields using **addField()**. **JMethodDeclaration**s and **JConstructorDeclaration**s add themselves, using **addMethod()**, and then delegate their code generation to their bodies. It's not rocket science.

The code for **JMethodDeclaration.codegen()** illustrates what these **codegen()** methods look like:

```
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);
    }
}
```

In general we generate only the class headers, members, and their instructions and operands. **CLEmitter** takes care of the rest. For example, here is the result of executing

```
javap HelloWorld
```

where **javap** is a Java tool which disassembles **HelloWorld.class**:

```
public class HelloWorld extends java.lang.Object
{
  public HelloWorld();
  Code:
   Stack=1, Locals=1, Args_size=1
   0: aload_0
   1: invokespecial   #8; //Method java/lang/Object."<init>":()V
   4: return
```

1-20

Draft, © 2008-2011 by Bill Campbell, Swami Iyer and Bahar Akbal-Delibaş

```
  public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=1, Args_size=1
   0: getstatic   #17; //Field java/lang/System.out:
                       // Ljava/io/PrintStream;
   3:   ldc       #19; //String Hello, World!
   5:   invokevirtual  #25; //Method java/io/PrintStream.println:
                           //(Ljava/lang/String;)V
   8:   return
}
```

We have shown only the instructions; tables such as the constant table have been left out of our illustration.

In general, **CLEmitter** does the following for us:

- It builds the constant table and generates references that the JVM can use to reference names and constants; one need only generate the instructions and their operands, using names and literals.
- It computes branch offsets and addresses; the user can use mnemonic labels.
- It computes the argument and local variable counts and the stack space a method requires to do computation.
- Constructs the complete class file.

The **CLEmitter** is discussed in more detail in Appendix D. JVM code generation is discussed more fully in Chapter 5.

1.5 The j-- Compiler Source Tree

The zip file **j--.zip** containing the *j*-- distribution can be downloaded from the following website:

http://www.cs.umb.edu/j--

The zip file may be unzipped into any directory of your choosing. Throughout this book, we refer to this directory as **$j**.

For a detailed description of what is in the software bundle, how to setup the compiler for command-line execution, how to setup, run, and debug the software in Eclipse[11], and how to add *j*-- test programs to the test framework, see Appendix A.

$j/j--/src/jminusminus contains the source files for the compiler, where **jminusminus** is a package. These include:

---

[11]   An open source IDE. http://www.eclipse.org, which supports ant and junit.

- **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 backend code that is used by *j--* for creating JVM bytecode; 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 machine's symbolic assembly language);
- **j--.jj**, the input file to JavaCC[12] 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; and
- other Java files providing representation for types and the symbol table.

$j/j--/bin/j-- is a script to run the compiler. It 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.
  -d <dir> Specify where to place generated class 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!
```

---

[12]   A scanner and parser generator for Java. http://javacc.dev.java.net/

1.5.1 Enhancing j--

Although *j--* is a subset of Java, it provides an elaborate framework with which one may add new Java constructs to *j--*. This will be the objective of many of the exercises in this book. In fact, with what we know so far about *j--*, we are already in a position to start enhancing the language by adding new albeit simple constructs to it.

As an illustrative example, we will add the division[13] operator to *j--*. This involves modifying the lexical analyzer 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.

In adding new language features to *j--*, we advocate the use of the Extreme Programming[14] (XP) paradigm, which emphasizes writing tests before writing code. We will do exactly this with the implementation of the division operator.

1.5.2 Writing Tests

Writing tests for new language constructs using the *j--* test framework involves,

- writing *pass* tests, which are *j--* programs that can successfully be compiled using the *j--* compiler;
- writing JUnit test cases that would run these pass tests;
- adding the JUnit test cases to the *j--* test suite;
- and finally, writing *fail* tests, which are erroneous *j--* programs. Compiling a fail test using *j--* should result in the compiler's reporting the errors and gracefully terminating without producing any .class files for the erroneous program.

We first write a *pass* test `Division.java` for the division operator, which simply has a method `divide()` that accepts two arguments *x* and *y*, and returns the result of dividing *x* by *y*. We place this file under the `$j/j--/tests/pass` folder; `pass` is a package.

```
// Division.java

package pass;

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

---

[13]   We only handle integer division since *j--* supports only `int`s as numeric types.
[14]   http://www.extremeprogramming.org/

1-23

Next, we write a JUnit test case `DivisionTest.java`, with a method `testDivide()` that tests the `divide()` method in `Division.java` with various arguments. We place this file under `$j/j--/tests/junit` folder; `junit` is a package.

```java
package junit;

import junit.framework.TestCase;
import pass.Division;

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);
    }
}
```

Now that we have a test case for the division operator, we must register it with the *j*-- test suite by making the following entry in the `suite()` method of `junit.JMinusMinusTestRunner`.

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

j-- supports only `int` as a numeric type, so the division operator can operate only on `int`s. The compiler should thus report an error if the operands have incorrect types; to test this we add the following *fail* test `Division.java` and place it under the `$j/j--/tests/fail` folder; `fail` is a package.

```java
// Division.java

package fail;

import java.lang.System;
```

1-24

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

1.5.3 Changes to Lexical and Syntactic Grammars

Appendix B specifies both the lexical and the syntactic grammars for the *j--* language; the former describes how individual tokens are composed and the latter describes how these tokens are put together to form language constructs. Chapters 3 and 4 describe such grammars in great detail.

For every language construct that is newly added to *j--*, we strongly recommend that these files be modified accordingly so that they accurately describe the modified syntax of the language. Though these files are for human consumption only, it is a good practice to keep them up to date.

For the division operator we are trying to add to *j--*, we add the following token describing the operator to `$j/j--/lexicalgrammar` under the *operators* section.

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

where `DIV` is the kind of the token and "`/`" is its image (string representation).

Since the division operator is a multiplicative operator, we add it to the grammar rule describing multiplicative expressions in the `$j/j--/grammar` file.

```
multiplicativeExpression ::= unaryExpression // level¹⁵ 2
                             {(STAR | DIV) unaryExpression}
```

Next, we discuss the changes in the *j--* codebase to get the compiler to support the division operation.

1.5.4 Changes to Scanner¹⁶

Before changing `Scanner.java`, we must register `DIV` as new token, so we add the following to the `TokenKind` enumeration in `TokenInfo.java` file.

```
enum TokenKind
{
```

---

[15]   Indicates operator precedence.
[16]   Here we only discuss the changes to the hand-written scanner. Scanners can also be generated; this is discussed in Chapter 2.

1-25

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

The `Scanner` method that actually recognizes and returns tokens in the input is `getNextToken()`. Currently, `getNextToken()` does not recognize "/" as an operator, and reports an error when it encounters a single / in the source program. In order to recognize the operator, we simply replace the `getNextToken()` code

```
if (ch == '/') {
    nextCh();
    if (ch == '/') {
        // CharReader maps all new lines to '\n'
        while (ch != '\n' && ch != EOFCH) {
            nextCh();
        }
    }
    else {
        reportScannerError(
            "Operator / is not supported in j--.");
    }
}
```

with the following.

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

1.5.5 Changes to Parser[17]

We first need to define a new AST node to represent the division expression. Since the operator is a multiplicative operator like "*", we can model the AST for the division expression based on the one (`JMultiplyOp`) for "*". We call the new AST node `JDivideOp`, and since division expression is a binary expression (one with two operands), we define it in `JBinaryExpression.java` as follows.

---

[17] Here we only discuss the changes to the hand-written parser. Parsers can also be generated; this is discussed in Chapter 3.

1-26

```
/**
 * The AST node for a division (/) expression.
 */

class JDivideOp extends JBinaryExpression
{
    /**
     * Construct an AST node for a division expression
     * given its line number, and the lhs and rhs operands.
     *
     * @param line line in which the division expression
     * occurs in the source file.
     * @param lhs lhs operand.
     * @param rhs rhs operand.
     */

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

    // TODO

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

    // TODO

    public void codegen(CLEmitter output)
    {

    }
}
```

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

```
/**
 * Parse a multiplicative expression.
 *
 * <pre>
 *   multiplicativeExpression ::= unaryExpression  // level 2
 *                                 {(STAR | DIV)
 *                                   unaryExpression}
 * </pre>
 *
 * @return an AST for a multiplicativeExpression.
 */

private JExpression multiplicativeExpression()
```

1-27

```
{
    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;
}
```

1.5.6 Semantic Analysis and Code Generation

Since `int` is the only numeric type supported in *j--*, analyzing the division operator is trivial. It involves analyzing its two operands, making sure each type is `int`, and setting the resulting expression's type to `int`. We thus implement `analyze()` in the `JDivideOp` AST as follows.

```
/**
 * Analyzing the / operation involves analyzing its operands,
 * checking types, and determining the result type.
 *
 * @param context in which names are resolved.
 * @return the analyzed (and possibly rewritten) AST subtree.
 */

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;
}
```

Generating code for the division operator is also trivial. It simply involves generating (through delegation) code for its operands and emitting the JVM (`IDIV`) instruction for the (integer) division of two numbers. Hence the following implementation for `codegen()` in `JDivideOp`.

```
/**
 * Generating code for the / operation involves generating code
```
1-28

```
 * for the two operands, and then the division instruction.
 *
 * @param output the code emitter (basically an abstraction for
 * producing the .class file).
 */

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

The **IDIV** instruction is a zero-argument instruction. The operands that it operates on have to be loaded on the operand stack prior to executing the instruction.

1.5.7 Testing the Changes

Finally, we need to test the addition of the new (division operator) construct to *j--*. This can be done at the command prompt by running:

```
ant
```

which compiles our tests using the hand-written scanner and parser, and then tests them. The results of compiling and running the tests are written to the console (**STDOUT**).

Alternatively, one could compile and run the tests from within Eclipse; Appendix A describes how.

1.6 The Organization of this Book

This book is organized like a compiler. You may think of this first chapter as the main program, the driver if you like. It gives the overall structure of compilers in general, and of our *j--* compiler in particular.

In Chapter 2 we talk about the scanning of tokens, that is, lexical analysis.

In Chapter 3 we discuss context-free grammars and parsing. We first address the recursive descent parsing technique, which is the method **Parser** uses to parse *j--*. We then go on to examine the LL and LR parsing strategies, both of which are used in various compilers today.

In Chapter 4 we discuss type checking, or semantic analysis. There are two passes required for this in the *j--* compiler and we will discuss both of them. We shall also discuss the use of attribute grammars for declaratively specifying semantic analysis.

In Chapter 5 we discuss JVM code generation. Again we address the peculiarities of code generation in our *j--* compiler, and then some other more general issues in code

1-29

generation.

In Chapter 6 we will discuss extensions to *j--*, that is, how one might add new constructs to the language.

In Chapter 7 we discuss translating JVM code to instructions native to a MIPS computer; MIPS is a register-based RISC architecture. We will discuss what is generally called optimization: a process by which the compiler produces better (that is, faster and smaller) target programs. Although our compiler has no optimizations, a general introduction to them is important.

In Chapter 8 register allocation is the principal challenge.

In Chapter 9 we will talk about a few celebrity compilers such as the Oracle Java compiler, and Microsoft's C# compiler.

Appendix A gives instructions on setting up a *j--* development environment.

Appendix B contains the lexical and syntactic grammar for *j--*.

Appendix C contains the lexical and syntactic grammar for Java.

Appendix D describes the `CLEmitter` interface and also provides a group-wise summary of the JVM instruction set.

Appendix E describes James Larus's SPIM simulator for the MIPS family of computers and how to write *j--* programs that target SPIM.

Further Readings

The *Java* programming language is fully described in (Gosling et al, 2005). The Java Virtual Machine is described in (Lindholm and Yellin, 1999).

If you want to be a compiler writer, read (Aho, Lam, Sethi and Ullman, 2007), commonly referred to as the "Dragon Book." Other classic texts include (Appel, 2002), (Cooper and Torczon, 2003), (Allen and Kennedy, 2001), and (Muchnick, 1997).

A reasonable introduction to testing is (Whittaker, 2002). Testing using the JUnit framework is nicely described in (Link, 2003) and (Rainsberger, 2004). A good introduction to extreme programming, where development is driven by tests, is (Beck, 1999).

Exercises

    1.1. We suggest you use either Emacs or Eclipse for working with the *j--* compiler.

In any case, you will want to get the *j--* code tree onto your own machine. If you choose to use Eclipse, then do the following.

a. Download Eclipse and install it on your own computer. You can get Eclipse from http://www.eclipse.org.
b. Download the *j--* distribution from http://www.cs.umb.edu/j--/.
c. Follow the directions in Appendix A for importing the *j--* code tree as a project into Eclipse.

1.2. Now is a good time to begin browsing through the code for the *j--* compiler. Locate and browse through each of the following classes.

```
a. Main
b. Scanner
c. Parser
d. JCompilationUnit
e. JClassDeclaration
f. JMethodDeclaration
g. JVariableDeclaration
h. JBlock
i. JMessageExpression
j. JVariable
k. JLiteralString
```

The remaining exercises may be thought of as optional. Some students (and their professors) may choose to go directly to Chapter 2.

*Exercises 1.3 through 1.9 require studying the compiler in its entirety, if only cursorily, and then making slight modifications to it. Notice that, in these exercises, many of the operators have different levels of precedence, just as * has a different level of precedence in j-- than does +. These levels of precedence are captured in the Java grammar (in Appendix C); for example, the parser uses one method to parse expressions involving * and /, and another to parse expressions involving + and −.*

1.3. To start, follow the process outlined in section 1.5 to implement the Java remainder operator **%**.

1.4. Implement the Java shift operators, **<<**, **>>**, and **>>>**.

1.5. Implement the Java bitwise inclusive or operator, **|**.

1.6. Implement the Java bitwise exclusive or operator, **^**.

1.7. Implement the Java bitwise and operator, **&**.

1.8. Implement the Java unary bitwise complement operator, **~** and the Java unary **+**

1-31

operator. What code is generated for the latter?

1.9. Write tests for all of the exercises (1.3 through 1.8) done above. Put these tests where they belong in the code tree and modify the JUnit framework in the code tree for making sure they are invoked.

Exercise 1.10 through 1.16 are exercises in j-- programming. J-- is a subset of Java and is described in Appendix B.

1.10.   Write a *j*-- program `Fibonacci.java` that accepts a number *n* as input and outputs the *n*th Fibonacci number.

1.11.   Write a *j*-- program `GCD.java` that accepts two numbers *a* and *b* as input, and outputs the Greatest Common Divisor (GCD) of *a* and *b*. Hint: Use the Euclidean algorithm[18].

1.12.   Write a *j*-- program `Primes.java` that accepts a number *n* as input, and outputs the all the prime numbers that are less than or equal to *n*. Hint: Use the Sieve of Eratosthenes algorithm[19]. For example:

```
java Primes 11
```

should output:

```
2 3 5 7 11
```

1.13.   Write a *j*-- program `Date.java` that accepts a date in "yyyy-mm-dd" format as input, and outputs the date in "Month Day, Year" format. For example:

```
java Date 1879-03-14[20]
```

should output:

```
March 14, 1879
```

1.14.   Write a *j*-- program `Palindrome.java` that accepts a string as input, and outputs the string if it is a palindrome (a string that reads the same in either direction), and outputs nothing if it isn't. The program should be case-insensitive to the input. For example:

```
java Palindrome Malayalam[21]
```

---

[18] http://en.wikipedia.org/wiki/Euclidean_algorithm
[19] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
[20] Day Albert Einstein was born
[21] Language spoken in Kerala, a southern Indian state

should output:

```
Malayalam
```

1.15.   Suggest enhancements to the *j--* language that would simplify the implementation of the programs described in the previous exercises (1.10 through 1.14).

1.16.   For each of the *j--* programs described in exercises 1.10 through 1.14, write a JUnit test case and integrate it with the *j--* test framework (Appendix C describes how this can be done).

Exercises 1.17 through 1.25 give the reader practice in reading JVM code and using CLEmitter for producing JVM code (in the form of .class files). The JVM and the CLEmitter are described in Appendix D.

1.17.   Disassemble (Appendix A describes how this can be done) a Java class (say `java.util.ArrayList`), study the output, and list the following:
- Major and minor version
- Size of the constant pool table
- Super class
- Interfaces
- Field names, their access modifiers, type descriptors, and their attributes (just names)
- Method names, their access modifiers, descriptors, exceptions thrown, and their method and code attributes (just names)
- Class attributes (just names)

1.18.   Compile `$j/j--/tests/pass/HelloWorld.java`[22] using the *j--* compiler and Sun's *javac* compiler. Disassemble the class file produced by each and compare the output. What differences do you see?

1.19.   Disassemble the class file produced by the *j--* compiler for `$j/j--/tests/pass/Series.java`, save the output in `Series.bytecode`. Add a single-line `(// …)` comment for each JVM instruction in `Series.bytecode` explaining what the instruction does.

1.20.   Write down the following class names in internal form:

```
java.lang.Thread
java.util.ArrayList
```

---

[22] `$j` is the directory containing the *j--* source tree.

1-33

```
java.io.FileNotFoundException
jminusminus.Parser
Employee
```

1.21.   Write down the method descriptor for each of the following constructors/method declarations:

```
public Employee(String name) ...
public Coordinates(float latitude, float longitude) ...
public Object get(String key) ...
public void put(String key, Object o) ...
public static int[] sort(int[] n, boolean ascending) ...
public int[][] transpose(int[][] matrix) ...
```

1.22.   Write a program `GenGCD.java` that produces, using `CLEmitter`, a `GCD.class` file with the following methods:

```
// Returns the Greatest Common Divisor (GCD) of a and b.
public static int compute(int a, int b)
{
    …
}
```

Running GCD as follows:

```
java GCD 42 84
```

should output:

```
42
```

Modify `GenGCD.java` to handle `java.lang.NumberFormatException` that `Integer.parseInt()` raises if the argument is not an integer, and in the handler, print an appropriate error message to `STDERR`.

1.23.   Write a program `GenPrimality.java` that produces, using `CLEmitter`, `Primality.class`, `Primality1.class`, `Primality2.class`, and `Primality3.class` files, where `Primality.class` is an interface with the following method:

```
// Returns true if the specified number is prime, false
// otherwise.
public boolean isPrime(int n);
```

and `Primality1.class`, `Primality2.class`, and `Primality3.class` are three

1-34

different implementations of the **interface**. Write a *j--* program
TestPrimality.java that has a test driver for the three implementations.

1.24.   Write a program **GenWC.java** that produces, using **CLEmitter**, a
WC.class which emulates the UNIX command **wc** that displays the number of
lines, words, and bytes contained in each input file.

1.25.   Write a program **GenGravity.java** that produces, using **CLEmitter**, a
Gravity.class file which computes the acceleration *g* due to gravity at a point
on the surface of a massive body. The program should accept the mass *M* of the
body, and the distance *r* of the point from body's center as input. Use the
following formula for computing *g*:

$$g = GMr^{-2}$$

where $G = 6.67 \times 10^{-11}$ Nm^2kg^{-2}, is the universal gravitational constant.

1-36