Chapter 1

Language Applications Cracked Open

In this first part of the book, we're going to learn how to recognize computer languages. (A *language* is just a set of valid sentences.) Every language application we look at will have a parser (recognizer) component, unless it's a pure code generator.

We can't just jump straight into the patterns, though. We need to see how everything fits together first. In this chapter, we'll get an architectural overview and then tour the patterns at our disposal. Finally, we'll look at the guts of some sample language applications to see how they work and how they use patterns.

1.1 The Big Picture

Language applications can be very complicated beasts, so we need to break them down into bite-sized components. The components fit together into a multistage pipeline that analyzes or manipulates an input stream. The pipeline gradually converts an input sentence (valid input sequence) to a handy internal data structure or translates it to a sentence in another language.

We can see the overall data flow within the pipeline in Figure 1.1, on the next page. The basic idea is that a reader recognizes input and builds an*intermediate representation* (IR) that feeds the rest of the application. At the opposite end, a generator emits output based upon the IR and what the application learned in the intermediate stages. The intermediate stages form the *semantic analyzer* component. Loosely speaking,

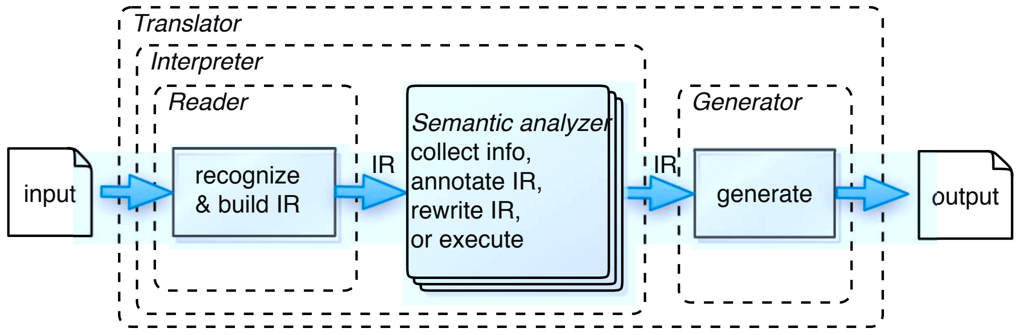


Figure 1.1: The multistage pipeline of a language application

semantic analysis figures out what the input means (anything beyond syntax is called the*semantics*).

The kind of application we're building dictates the stages of the pipeline and how we hook them together. There are four broad application categories:

- *Reader*: A reader builds a data structure from one or more input streams. The input streams are usually text but can be binary data as well. Examples include configuration file readers, program analysis tools such as a method cross-reference tool, and class file loaders.
- *Generator*: A generator walks an internal data structure and emits output. Examples include object-to-relational database mapping tools, object serializers, source code generators, and web page generators.
- *Translator* or *Rewriter*: A translator reads text or binary input and emits output conforming to the same or a different language. It is essentially a combined reader and generator. Examples include translators from extinct programming languages to modern languages, wiki to HTML translators, refactorers, profilers that instrument code, log file report generators, pretty printers, and macro preprocessors. Some translators, such as assemblers and compilers, are so common that they warrant their own subcategories.
- *Interpreter*: An interpreter reads, decodes, and executes instructions. Interpreters range from simple calculators and POP protocol servers all the way up to programming language implementations such as those for Java, Ruby, and Python.

1.2 A Tour of the Patterns

This section is a road map of this book's 31 language implementation patterns. Don't worry if this quick tour is hard to digest at first. The fog will clear as we go through the book and get acquainted with the patterns.

Parsing Input Sentences

Reader components use the patterns discussed in Chapter 2, *Basic Parsing Patterns*, on page 37 and Chapter 3, *Enhanced Parsing Patterns*, on page 65 to *parse* (recognize) input structures. There are five alternative parsing patterns between the two chapters. Some languages are tougher to parse than others, and so we need parsers of varying strength. The trade-off is that the stronger parsing patterns are more complicated and sometimes a bit slower.

We'll also explore a little about grammars (formal language specifications) and figure out exactly how parsers recognize languages. Pattern 1, *Mapping Grammars to Recursive-Descent Recognizers*, on page 45 shows us how to convert grammars to hand-built parsers. ANTLR[1] (or any similar parser generator) can do this conversion automatically for us, but it's a good idea to familiarize ourselves with the underlying patterns.

The most basic reader component combines Pattern 2, *LL(1) Recursive-Descent Lexer*, on page 49 together with Pattern 3, *LL(1) Recursive-Descent Parser*, on page 54 to recognize sentences. More complicated languages will need a stronger parser, though. We can increase the recognition strength of a parser by allowing it to look at more of the input at once (Pattern 4, *LL(k) Recursive-Descent Parser*, on page 59).

When things get really hairy, we can only distinguish sentences by looking at an entire sentence or phrase (subsentence) using Pattern 5, *Backtracking Parser*, on page 71.

Backtracking's strength comes at the cost of slow execution speed. With some tinkering, however, we can dramatically improve its efficiency. We just need to save and reuse some partial parsing results with Pattern 6, *Memoizing Parser*, on page 78.

For the ultimate parsing power, we can resort to Pattern 7, *Predicated Parser*, on page 84. A predicated parser can alter the normal parsing flow based upon run-time information. For example, input `T(i)` can mean

1. http://www.antlr.org

different things depending on how we defined T previously. A predicate parser can look up T in a dictionary to see what it is.

Besides tracking input symbols like T, a parser can execute actions to perform a transformation or do some analysis. This approach is usually too simplistic for most applications, though. We'll need to make multiple passes over the input. These passes are the stages of the pipeline beyond the reader component.

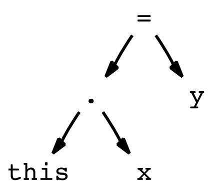
Constructing Trees

Rather than repeatedly parsing the input text in every stage, we'll construct an IR. The IR is a highly processed version of the input text that's easy to traverse. The nodes or elements of the IR are also ideal places to squirrel away information for use by later stages. In Chapter 4, *Building Intermediate Form Trees*, on page 88, we'll discuss why we build trees and how they encode essential information from the input.

The nature of an application dictates what kind of data structure we use for the IR. Compilers require a highly specialized IR that is very low level (elements of the IR correspond very closely with machine instructions). Because we're not focusing on compilers in this book, though, we'll generally use a higher-level tree structure.

The first tree pattern we'll look at is Pattern 8, *Parse Tree*, on page 105. Parse trees are pretty "noisy," though. They include a record of the rules used to recognize the input, not just the input itself. Parse trees are useful primarily for building syntax-highlighting editors. For implementing source code analyzers, translators, and the like, we'll build*abstract syntax trees* (ASTs) because they are easier to work with.

An AST has a node for every important token and uses operators as subtree roots. For example, the AST for assignment statement this.x=y; is as follows:



The AST implementation pattern you pick depends on how you plan on traversing the AST (Chapter 4, *Building Intermediate Form Trees*, on page 88 discusses AST construction in detail).

Pattern 9, *Homogeneous AST*, on page 109 is as simple as you can get. It uses a single object type to represent every node in the tree. Homogeneous nodes also have to represent specific children by position within a list rather than with named node fields. We call that a *normalized child list*.

If we need to store different data depending on the kind of tree node, we need to introduce multiple node types with Pattern 10, *Normalized Heterogeneous AST*, on page 111. For example, we might want different node types for addition operator nodes and variable reference nodes. When building heterogeneous node types, it's common practice to track children with fields rather than lists (Pattern 11, *Irregular Heterogeneous AST*, on page 114).

Walking Trees

Once we've got an appropriate representation of our input in memory, we can start extracting information or performing transformations.

To do that, we need to traverse the IR (AST, in our case). There are two basic approaches to tree walking. Either we embed methods within each node class (Pattern 12, *Embedded Heterogeneous Tree Walker*, on page 128) or we encapsulate those methods in an external visitor (Pattern 13, *External Tree Visitor*, on page 131). The external visitor is nice because it allows us to alter tree-walking behavior without modifying node classes.

Rather than build external visitors manually, though, we can automate visitor construction just like we can automate parser construction. To recognize tree structures, we'll use Pattern 14, *Tree Grammar*, on page 134 or Pattern 15, *Tree Pattern Matcher*, on page 138. A tree grammar describes the entire structure of all valid trees, whereas a tree pattern matcher lets us focus on just those subtrees we care about. You'll use one or more of these tree walkers to implement the next stages in the pipeline.

Figuring Out What the Input Means

Before we can generate output, we need to analyze the input to extract bits of information relevant to generation (semantic analysis). Language analysis is rooted in a fundamental question: for a given symbol reference x, what is it? Depending on the application, we might need to know whether it's a variable or method, what type it is, or where it's defined. To answer these questions, we need to track all input symbols using one of the *symbol tables* in Chapter 6, *Tracking and Identifying Program Symbols*, on page 146 or Chapter 7, *Managing Symbol Tables for Data Aggregates*, on page 170. A symbol table is just a dictionary that maps symbols to their definitions.

The semantic rules of your language dictate which symbol table pattern to use. There are four common kinds of scoping rules: languages with a single scope, nested scopes, C-style **struct** scopes, and class scopes. You'll find the associated implementations in Pattern 16, *Symbol Table for Monolithic Scope*, on page 156, Pattern 17, *Symbol Table for Nested Scopes*, on page 161, Pattern 18, *Symbol Table for Data Aggregates*, on page 176, andPattern 19, *Symbol Table for Classes*, on page 182.

Languages such as Java, C#, and C++ have a ton of semantic compile-time rules. Most of these rules deal with type compatibility between operators or assignment statements. For example, we can't multiply a string by a class name. Chapter 8, *Enforcing Static Typing Rules*, on page 196 describes how to compute the types of all expressions and then check operations and assignments for type compatibility. For non-object-oriented languages like C, we'd apply Pattern 22, *Enforcing Static Type Safety*, on page 216. For object-oriented languages like C++ or Java, we'd apply Pattern 23, *Enforcing Polymorphic Type Safety*, on page 223. To make these patterns easier to absorb, we'll break out some of the necessary infrastructure in Pattern 20, *Computing Static Expression Types*, on page 199 and Pattern 21, *Automatic Type Promotion*, on page 208.

If you're building a reader like a configuration file reader or Java .class file reader, your application pipeline would be complete at this point. To build an interpreter or translator, though, we have to add more stages.

Interpreting Input Sentences

Interpreters execute instructions stored in the IR but usually need other data structures too, like a symbol table. Chapter 9, *Building High-Level Interpreters*, on page 232 describes the most common interpreter implementation patterns, including Pattern 24, *Syntax-Directed Interpreter*, on page 238, Pattern 25, *Tree-Based Interpreter*, on page 243, Pattern 27, *Stack-Based Bytecode Interpreter*, on page 272, and Pattern 28, *Register-Based Bytecode Interpreter*, on page 280. From a capability standpoint, the interpreter patterns are equivalent (or could be made equally powerful). The differences between them lie in the instruction set, execution efficiency, interactivity, ease-of-use, and ease of implementation.

Translating One Language to Another

Rather than interpreting a computer language, we can translate programs to another language (at the extreme, compilers translate high-level programs down to machine code). The final component of any translator is a generator that emits structured text or binary. The output is a function of the input and the results of semantic analysis. For simple translations, we can combine the reader and generator into a single pass using Pattern 29, *Syntax-Directed Translator*, on page 307. Generally, though, we need to decouple the order in which we compute output phrases from the order in which we emit output phrases. For example, imagine reversing the statements of a program. We can't generate the first output statement until we've read the final input statement. To decouple input and output order, we'll use a model-driven approach. (See Chapter 11, *Translating Computer Languages*, on page 290.)

Because generator output always conforms to a language, it makes sense to use a formal language tool to emit structured text. What we need is an "unparser" called a *template engine*. There are many excellent template engines out there but, for our sample implementations, we'll use StringTemplate.[2] (See Chapter 12, *Generating DSLs with Templates*, on page 323.)

So, that's how patterns fit into the overall language implementation pipeline. Before getting into them, though, it's worth investigating the architecture of some common language applications. It'll help keep everything in perspective as you read the patterns chapters.

1.3 Dissecting a Few Applications

Language applications are a bit like fractals. As you zoom in on their architecture diagrams, you see that their pipeline stages are themselves multistage pipelines. For example, though we see compilers as black boxes, they are actually deeply nested pipelines. They are so complicated that we have to break them down into lots of simpler components. Even the individual top-level components are pipelines. Digging deeper,

2. http://www.stringtemplate.org

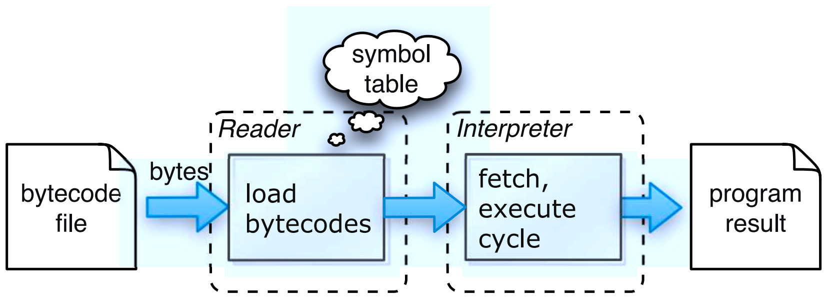


Figure 1.2: Bytecode interpreter pipeline

the same data structures and algorithms pop up across applications and stages.

This section dissects a few language applications to expose their architectures. We'll look at a bytecode interpreter, a bug finder (source code analyzer), and a C/C++ compiler. The goal is to emphasize the architectural similarity between applications and even between the stages in a single application. The more you know about existing language applications, the easier it'll be to design your own. Let's start with the simplest architecture.

Bytecode Interpreter

An *interpreter* is a program that executes other programs. In effect, an interpreter simulates a hardware processor in software, which is why we call them *virtual machines*. An interpreter's instruction set is typically pretty low level but higher level than raw machine code. We call the instructions*bytecodes* because we can represent each instruction with a unique integer code from 0..255 (a byte's range).

We can see the basic architecture of a bytecode interpreter in Figure 1.2. A reader loads the bytecodes from a file before the interpreter can start execution. To execute a program, the interpreter uses a *fetch-decode-execute cycle*. Like a real processor, the interpreter has an instruction pointer that tracks which instruction to execute next. Some instructions move data around, some move the instruction pointer (branches and calls), and some emit output (which is how we get the program result). There are a lot of implementation details, but this gives you the basic idea.

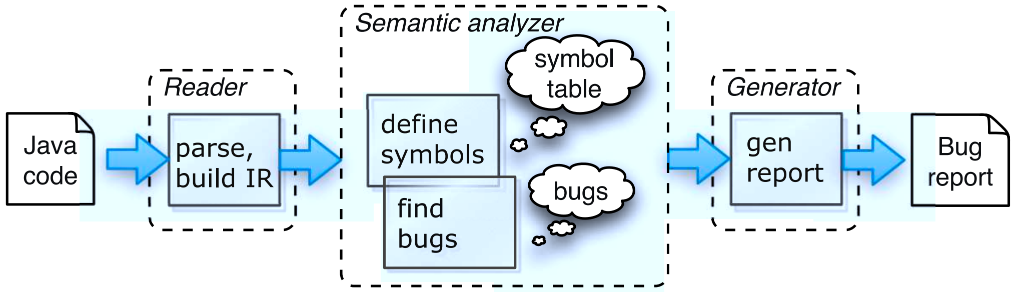


Figure 1.3: Source-level bug finder pipeline

Languages with bytecode interpreter implementations include Java, Lua,[3] Python, Ruby, C#, and Smalltalk.[4] Lua uses Pattern 28, *Register-Based Bytecode Interpreter*, on page 280, but the others use Pattern 27, *Stack-Based Bytecode Interpreter*, on page 272. Prior to version 1.9, Ruby used something akin to Pattern 25, *Tree-Based Interpreter*, on page 243.

Java Bug Finder

Let's move all the way up to the source code level now and crack open a Java bug finder application. To keep things simple, we'll look for just one kind of bug called *self-assignment*. Self-assignment is when we assign a variable to itself. For example, the setX() method in the following Point class has a useless self-assignment because this.x and x refer to the same field x:

```
class Point {
    int x,y;
    void setX(int y) { this.x = x; } // oops! Meant setX(int x)
    void setY(int y) { this.y = y; }
}
```

The best way to design a language application is to start with the end in mind. First, figure out what information you need in order to generate the output. That tells you what the final stage before the generator computes. Then figure out what that stage needs and so on all the way back to the reader.

3. http://www.lua.org
4. http://en.wikipedia.org/wiki/Smalltalk_programming_language

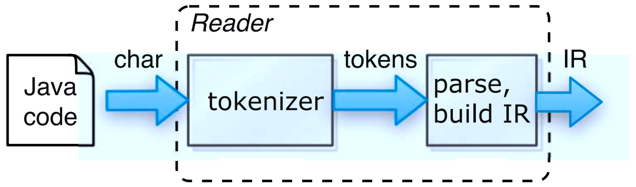


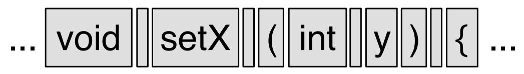
Figure 1.4: Pipeline that recognizes Java code and builds an IR

For our bug finder, we need to generate a report showing all self-assignments. To do that, we need to find all assignments of the form `this.x = x` and flag those that assign to themselves. To do that, we need to figure out (*resolve*) to which entity `this.x` and `x` refer. That means we need to track all symbol definitions using a symbol table like Pattern 19, *Symbol Table for Classes*, on page 182. We can see the pipeline for our bug finder in Figure 1.3, on the previous page.

Now that we've identified the stages, let's walk the information flow forward. The parser reads the Java code and builds an intermediate representation that feeds the semantic analysis phases. To parse Java, we can use Pattern 2, *LL(1) Recursive-Descent Lexer*, on page 49, Pattern 4, *LL(k) Recursive-Descent Parser*, on page 59, Pattern 5, *Backtracking Parser*, on page 71, and Pattern 6, *Memoizing Parser*, on page 78. We can get away with building a simple IR: Pattern 9, *Homogeneous AST*, on page 109.

The semantic analyzer in our case needs to make two passes over the IR. The first pass defines all the symbols encountered during the walk. The second pass looks for assignment patterns whose left-side and right-side resolve to the same field. To find symbol definitions and assignment tree patterns, we can use Pattern 15, *Tree Pattern Matcher*, on page 138. Once we have a list of self-assignments, we can generate a report.

Let's zoom in a little on the reader (see Figure 1.4). Most text readers use a two-stage process. The first stage breaks up the character stream into vocabulary symbols called*tokens*. The parser feeds off these tokens to check syntax. In our case, the tokenizer (or*lexer*) yields a stream of vocabulary symbols like this:



As the parser checks the syntax, it builds the IR. We have to build an IR in this case because we make multiple passes over the input. Retokenizing and reparsing the text input for every pass is inefficient and makes it harder to pass information between stages. Multiple passes also support forward references. For example, we want to be able to see field x even if it's defined after method setX(). By defining all symbols first, before trying to resolve them, our bug-finding stage sees x easily.

Now let's jump to the final stage and zoom in on the generator. Since we have a list of bugs (presumably a list of Bug objects), our generator can use a simple **for** loop to print out the bugs. For more complicated reports, though, we'll want to use a template. For example, if we assume that Bug has fields file, line, and fieldname, then we can use the following two StringTemplate template definitions to generate a report (we'll explore template syntax in Chapter 12, *Generating DSLs with Templates*, on page 323).

```
report(bugs) ::= "<bugs:bug()>" // apply template bug to each bug object
bug(b) ::= "bug: <b.file>:<b.line> self assignment to <b.fieldname>"
```

All we have to do is pass the list of Bug objects to the **report** template as attribute bugs, and StringTemplate does the rest.

There's another way to implement this bug finder. Instead of doing all the work to read Java source code and populate a symbol table, we can leverage the functionality of the javac Java compiler, as we'll see next.

Java Bug Finder Part Deux

The Java compiler generates .class files that contain serialized versions of a symbol table and AST. We can use Byte Code Engineering Library (BCEL)[5] or another class file reader to load .class files instead of building a source code reader (the fine tool FindBugs[6] uses this approach). We can see the pipeline for this approach in Figure 1.5, on the following page.

The overall architecture is roughly the same as before. We have just short-circuited the pipeline a little bit. We don't need a source code parser, and we don't need to build a symbol table. The Java compiler has already resolved all symbols and generated bytecode that refers to unique program entities. To find self-assignment bugs, all we have to

5. http://jakarta.apache.org/bcel/
6. http://findbugs.sourceforge.net/

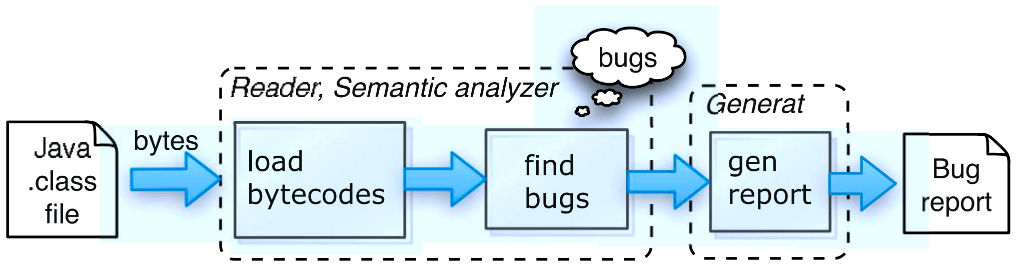


Figure 1.5: Java bug finder pipeline feeding off .class files

do is look for a particular bytecode sequence. Here is the bytecode for method setX():

```
0:   aload_0       // push 'this' onto the stack
1:   aload_0       // push 'this' onto the stack
2:   getfield #2; // push field this.x onto the stack
5:   putfield #2; // store top of stack (this.x) into field this.x
8:   return
```

The #2 operand is an offset into a symbol table and uniquely identifies the x (field) symbol. In this case, the bytecode clearly gets and puts the same field. If this.x referred to a different field than x, we'd see different symbol numbers as operands of getfield and putfield.

Now, let's look at the compilation process that feeds this bug finder. javac is a compiler just like a traditional C compiler. The only difference is that a C compiler translates programs down to instructions that run natively on a particular CPU.

C Compiler

A C compiler looks like one big program because we use a single command to launch it (via **cc** or **gcc** on UNIX machines). Although the actual C compiler is the most complicated component, the C compilation process has lots of players.

Before we can get to actual compilation, we have to preprocess C files to handle includes and macros. The preprocessor spits out pure C code with some line number directives understood by the compiler. The compiler munches on that for a while and then spits out assembly code (text-based human-readable machine code). A separate assembler translates the assembly code to binary machine code. With a few command-line options, we can expose this pipeline.

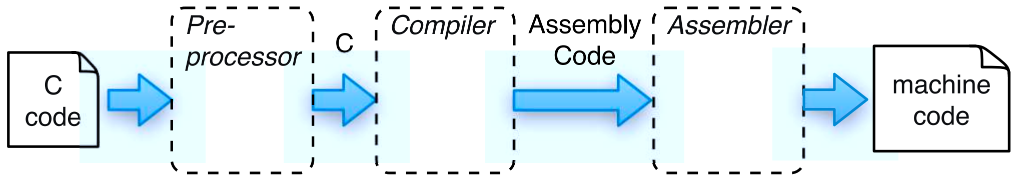


Figure 1.6: C compilation process pipeline

Let's follow the pipeline (shown in Figure 1.6) for the C function in file t.c:

```
void f() { ; }
```

First we preprocess t.c:

```
$ cpp t.c tmp.c         # preprocess t.c, put output into tmp.c
$
```

That gives us the following C code:

```
# 1 "t.c"               // line information generated by preprocessor
# 1 "<built-in>"        // it's not C code per se
# 1 "<command line>"
# 1 "t.c"
void f() { ; }
```

If we had included stdio.h, we'd see a huge pile of stuff in front of f(). To compile tmp.c down to assembly code instead of all the way to machine code, we use option -S. The following session compiles and prints out the generated assembly code:

```
$ gcc -S tmp.c              # compile tmp.c to tmp.s
$ cat tmp.s                 # print assembly code to standard output
        .text
.globl _f
_f:                         ; define function f
        pushl   %ebp        ; do method bookkeeping
        movl    %esp, %ebp  ; you can ignore this stuff
        subl    $8, %esp
        leave               ; clean up stack
        ret                 ; return to invoking function
        .subsections_via_symbols
$
```

To assemble tmp.s, we run as to get the object file tmp.o:

```
$ as -o tmp.o tmp.s         # assemble tmp.s to tmp.o
$ ls tmp.*
tmp.c   tmp.o   tmp.s
$
```

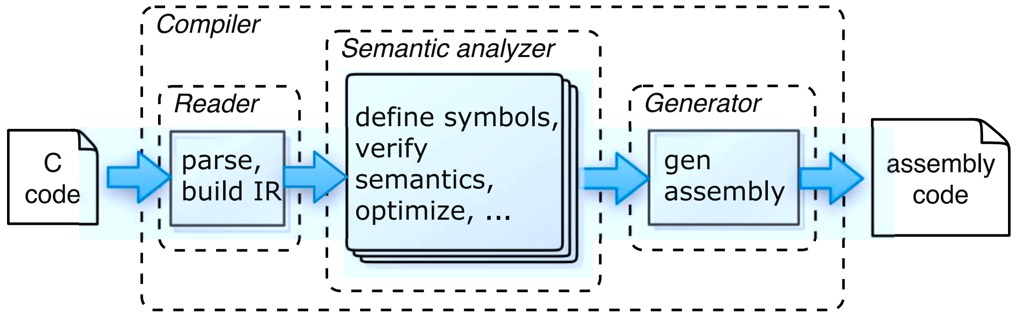


Figure 1.7: Isolated C compiler application pipeline

Now that we know about the overall compilation process, let's zoom in on the pipeline inside the C compiler itself.

The main components are highlighted in Figure 1.7. Like other language applications, the C compiler has a reader that parses the input and builds an IR. On the other end, the generator traverses the IR, emitting assembly instructions for each subtree. These components (the *front end* and *back end*) are not the hard part of a compiler.

All the scary voodoo within a compiler happens inside the semantic analyzer and optimizer. From the IR, it has to build all sorts of extra data structures in order to produce an efficient version of the input C program in assembly code. Lots of set and graph theory algorithms are at work. Implementing these complicated algorithms is challenging. If you'd like to dig into compilers, I recommend the famous "Dragon" book: *Compilers: Principles, Techniques, and Tools* [ALSU06] (Second Edition).

Rather than build a complete compiler, we can also leverage an existing compiler. In the next section, we'll see how to implement a language by translating it to an existing language.

Leveraging a C Compiler to Implement C++

Imagine you are Bjarne Stroustrup, the designer and original implementer of C++. You have a cool idea for extending C to have classes, but you're faced with a mammoth programming project to implement it from scratch.

To get C++ up and running in fairly short order, Stroustrup simply reduced C++ compilation to a known problem: C compilation. In other

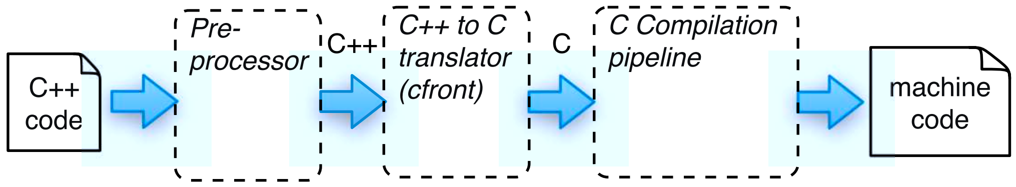


Figure 1.8: C++ (cfront) compilation process pipeline

words, he built a C++ to C translator calledcfront. He didn't have to build a compiler at all. By generating C, his nascent language was instantly available on any machine with a C compiler. We can see the overall C++ application pipeline in Figure 1.8. If we zoomed in on cfront, we'd see yet another reader, semantic analyzer, and generator pipeline.

As you can see, language applications are all pretty similar. Well, at least they all use the same basic architecture and share many of the same components. To implement the components, they use a lot of the same patterns. Before moving on to the patterns in the subsequent chapters, let's get a general sense of how to hook them together into our own applications.

1.4 Choosing Patterns and Assembling Applications

I chose the patterns in this book because of their importance and how often you'll find yourself using them. From my own experience and from listening to the chatter on the ANTLR interest list, we programmers typically do one of two things. Either we implement DSLs or we process and translate general-purpose programming languages. In other words, we tend to implement graphics and mathematics languages, but very few of us build compilers and interpreters for full programming languages. Most of the time, we're building tools to refactor, format, compute software metrics, find bugs, instrument, or translate them to another high-level language.

If we're not building implementations for general-purpose programming languages, you might wonder why I've included some of the patterns I have. For example, all compiler textbooks talk about symbol table management and computing the types of expressions. This book also spends roughly 20 percent of the page count on those subjects. The reason is that some of the patterns we'd need to build a compiler are also