CHAPTER 3

Building a Parser

This chapter explains the steps in designing and coding a working parser. The core design of a parser is the same for all the parsers in this book: recognize a language and build a result.

3.1 Design Overview

Usually the first step in designing a parser is to think of some sample strings that you want your parser to recognize. To parse this set of strings, you will create a new language. A language is always a set of strings, and your language will become a set that includes your sample strings.

You can begin to design your parser by writing the rules, or *grammar*, of your language. (Section 3.4, "Grammars: A Shorthand for Parsers," explains how to write your grammar.) Your parser will recognize strings that follow the rules of your grammar. Once you have a grammar, you can write the Java code for your parser as a direct translation of the grammar rules.

The other main aspect of a parser's design is the design of your assemblers. Assemblers let you create a new object when your parser recognizes an input string. After you have designed your assemblers and your rules, you bring them together. You plug assemblers in to subparsers to assemble parts of a target object as your parser recognizes text.

It is a good idea to work incrementally and iteratively. When you work incrementally, you get part of your language to work before the entire language works. Working iteratively means that you can expect to cycle through the steps of designing, coding, and testing many times on each increment you create.

Build your language gradually, expanding your parser and adding new features as you go. You will see your language grow, and you will become skillful in expanding the features of your language.

3.2 Deciding to Tokenize

An early design decision is whether you want to treat your language as a pattern of characters or as a pattern of tokens. Most commonly, you will *not* want to use a tokenizer for languages that let a user specify patterns of characters to match against. Chapter 8, "Parsing Regular Expressions," gives an example of parsing without using a tokenizer.

Tokens are composed of characters, so every language that is a pattern of tokens is also a pattern of characters. Theoretically, then, tokenizers are never necessary. However, it is usually practical to tokenize text and to specify a grammar for a language in terms of token terminals. Consider a robot control language that allows this command:

```
move robot 7.1 meters from base
```

If you do not plan to tokenize, your parser must recognize every character, including the whitespace between words. You also must ensure that you properly gather characters into words, and you must build the number value yourself. All of this is work that a tokenizer will happily perform for you. Chapter 9, "Advanced Tokenizing," discusses how to customize a tokenizer. When you are learning to design new languages, you may want to limit your languages to those that can benefit from the default behavior of class `Tokenizer` in package `sjm.parse.tokens`.

3.3 Designing Assemblers

One way to get a grip on the design of your parser is to think about how you will build the result you want from the text you recognize. So one way to get started with the design of a new parser is to start designing your assemblers.

3.3.1 The Collaboration of Parsers, Assemblers, and Assemblies

An assembly provides both a stack and a *target* object for the parser's assemblers to work on. The target object is like a sculpture, taking form as the parser recognizes the input text. Figure 3.1 shows the `Parser`, `Assembler`, and `Assembly` classes, which collaborate to sculpt text into a result.

3.3.2 Using an Assembly's Stack

Assembly objects contain two work areas: a stack and a target. By default, subclasses of `Terminal`, such as `Word` and `Num`, place on an assembly's stack the object they recognize from the assembly object's text. (You can prevent this by sending the

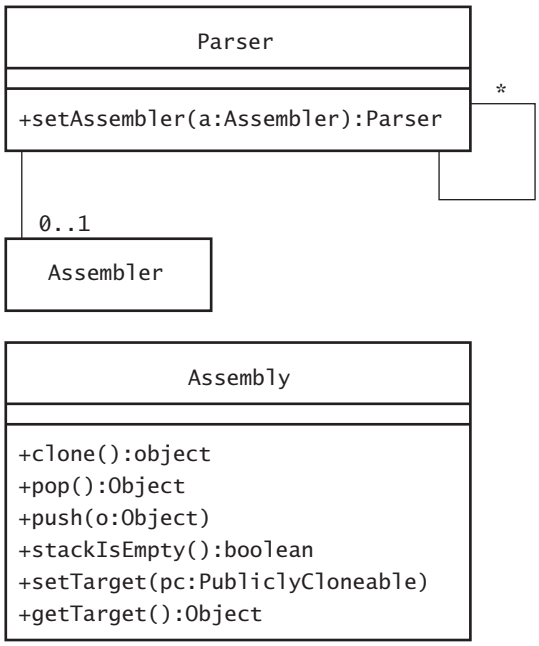

Figure 3.1 The `Parser`, `Assembler`, and `Assembly` classes. Each parser in a parser composite tries to match against the assembly, and each may use an assembler to work on the assembly after a successful match.

`Terminal` object a `discard()` message.) The following code shows a repetition of a `Num` parser that recognizes and stacks a series of numbers.

```
package sjm.examples.design;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to use an assembly's stack.
 */
public class ShowStack {

public static void main(String args[]) {

    Parser p = new Repetition(new Num());
    Assembly a = p.completeMatch(
        new TokenAssembly("2 4 6 8"));
    System.out.println(a);
}
}
```

This code creates a `TokenAssembly` around the string `"2 4 6 8"` and passes it to the parser `p`. The result of sending `completeMatch()` to `p` is a *new* `TokenAssembly`. The `completeMatch()` method returns the result as an abstract `Assembly` object, which we could cast to a `TokenAssembly` if we needed to.

Running this class prints

```
[2.0, 4.0, 6.0, 8.0]2.0/4.0/6.0/8.0^
```

When the assembly `a` prints itself, it first shows its stack, which is `[2.0, 4.0, 6.0, 8.0]`. This demonstrates the nature of `Num`, which treats every number as a `double` and places on the assembly's stack the tokens it finds. The output assembly also shows its tokenized input text, separating the tokens with slashes. Finally, the output assembly shows the location of its index at the end of the tokens.

3.3.3 Assemblers Plug In to Parser Composites

When a parser recognizes text, it knows nothing about the big picture in which it executes. For example, consider the `Num` parser in `sjm.parse.tokens`. A `Num` object might be recognizing one of a series of numbers in a string such as `"1.2 2.3 3.4"`, or it might be recognizing the price of a pound of coffee. The `Num` parser simply puts its number on the assembly's stack, leaving any further work to other assemblers. Typically, `Num` parsers and other terminals are not stand-alone parsers but rather are part of a composite. After a terminal places an object on an assembly's stack, another parser higher in the composite can find this object and do other work on the assembly.

3.3.4 A Language to Plug In To: Minimath

To see how assemblers plug in to a parser composite, consider a minimal arithmetic parser that recognizes only the "-" operator. That is, you want to recognize a language that contains numbers, all differences of two numbers, differences of three numbers, and so on. For example, the language contains the following:

```
{"0.0",
 "1.1 - 2.2",
 "1.1 - 2.2 - 3.3",
 "1.1 - 2.2 - 3.3 - 4.4", ...}
```

Let's call this language Minimath. You can describe the contents of Minimath with the following rules:

```
expression = Num minusNum*;
minusNum   = '-' Num;
```

These rules are shorthand for describing a language. They are also shorthand for describing the composition of a parser. Section 3.4 explains the rules for this shorthand in detail. You can abbreviate these rules to

```
e = Num m*;
m = '-' Num;
```

The first rule means that `e` is a number followed by zero or more occurrences of `m`. For example, `"25 - 16 - 9"` is the number `"25"` followed by `"- 16"` and `"- 9"`. The first rule uses the capitalized word `Num` to mean a terminal. In fact, you will use the class `Num` in `sjm.parse.tokens` when you build the `e` parser. The `e` rule uses the non-capitalized word `m` to refer to another rule, and it uses an asterisk ("`*`") to indicate repetition of `m`. The `m` rule indicates a pattern that has a minus sign ("`-`") followed by a number.

These rules describe the patterns of strings that make up the Minimath language, and they give you a formula for composing a parser to match Minimath. For example, the parser `e` recognizes a string such as `"25 - 16 - 9"` as an element of the Minimath language. If all you want is to recognize elements of Minimath and not compute their value, you can build the `e` parser as follows:

```
package sjm.examples.minimath;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to build a parser to recognize elements
 * of the language "Minimath".
 */
public class MinimathRecognize {

public static void main(String args[]) {
    Sequence e = new Sequence();

    e.add(new Num());

    Sequence m = new Sequence();
    m.add(new Symbol('-'));
    m.add(new Num());

    e.add(new Repetition(m));

    System.out.println(
        e.completeMatch(
            new TokenAssembly("25 - 16 - 9")));
}
}
```

This code prints the following:

```
[25.0, -, 16.0, -, 9.0]25.0/-/16.0/-/9.0^
```

This shows that the parser `e` recognizes the text `"25 - 16 - 9"`. Of course, the point of having a Minimath parser is not only to recognize text but also to build the value that the text represents.

3.3.5 Calculating a Minimath Result

You want the parser `e` to calculate a difference and leave it as a `Double` object on an assembly's stack. To accomplish this, you need two assemblers: one to handle numbers as the `Num` subparser finds them, and a second one to handle subtraction.

When a `Num` parser recognizes a number, it places an `sjm.parse.tokens.Token` object on the assembly's stack. To calculate an arithmetic sum, your `Num` parser needs an assembler to replace this `Token` object with a `Double` value that corresponds to the token's value. You can describe the design of the assembler you need as, "Pop the token at the top of the assembly's stack and push a corresponding number." In this example, you can reuse an assembler from `sjm.examples.arithmetic`. The code for `NumAssembler` is as follows:

```
package sjm.examples.arithmetic;

import sjm.parse.*;
import sjm.parse.tokens.*;

public class NumAssembler extends Assembler {
/**
 * Replace the top token in the stack with the token's
 * Double value.
 */
public void workOn(Assembly a) {
    Token t = (Token) a.pop();
    a.push(new Double(t.nval()));
}
}
```

This method assumes that the top of the input assembly's stack is a `Token` object. In practice, this assumption is safe as long as the assembler plugs in to a parser that stacks a token. Assemblers are small, non-reusable classes that plug behavior in to a particular parser.

The other assembler that a Minimath parser needs is one to handle subtraction. You can design the assembler without addressing which subparser it belongs to. Let us assume that, at some point, a Minimath composite parser places two numbers on the

stack of the assembly it is matching. At that point, you can describe the design of the needed assembler as, "Pop the top two numbers and push their difference." Again, you can reuse an assembler from `sjm.examples.arithmetic`. The code for `MinusAssembler` is as follows:

```
package sjm.examples.arithmetic;

import sjm.parse.*;

public class MinusAssembler extends Assembler {
/**
 * Pop two numbers from the stack and push the result of
 * subtracting the top number from the one below it.
 */
public void workOn(Assembly a) {
    Double d1 = (Double) a.pop();
    Double d2 = (Double) a.pop();
    Double d3 =
        new Double(d2.doubleValue() - d1.doubleValue());
    a.push(d3);
}
}
```

The only remaining design question is where this assembler belongs. Note from the rules that an expression `e` is a number followed by one or more occurrences of `m`. An effective strategy is to associate a `NumAssembler` with each `Num` parser, and a `MinusAssembler` with the `m` parser. The code looks like this:

```
package sjm.examples.minimath;

import sjm.parse.*;
import sjm.parse.tokens.*;
import sjm.examples.arithmetic.*;

/**
 * ...
 * This class shows, in a minimal example, where assemblers
 * plug in to a parser composite.
 */
public class MinimathCompute {

public static void main(String args[]) {
    Sequence e = new Sequence();

    Num n = new Num();
    n.setAssembler(new NumAssembler());

    e.add(n);

    Sequence m = new Sequence();
    m.add(new Symbol('-').discard());
```

```
        m.add(n);
        m.setAssembler(new MinusAssembler());

        e.add(new Repetition(m));

        TokenAssembly t = new TokenAssembly("25 - 16 - 9");
        Assembly out = e.completeMatch(t);
        System.out.println(out.pop());
    }
    }
```

The `main()` method creates a composite parser `e` for Minimath and plugs in assemblers to assemble a result for an arithmetic string. The code discards the minus signs because they serve their purpose in guiding the parser and need not appear on the stack. The method asks the parser for a complete match against a tokenized string and prints the top of the stack of the resulting assembly. Running this class prints the conventional answer:

```
0.0
```

3.3.6 The Minimath Parser as an Object

Figure 3.2 shows an object diagram for the parser `e`. The parser `e` is a sequence of two subparsers: a `Num` and a `Repetition`. Each time the `Num` parser in `e` recognizes a number token, it uses a `NumAssembler` object to replace the number token on the stack with a corresponding `Double`. Two of the subparsers—`n` and `m`—use assemblers to work on the assembly they match against. There is only one `n` subparser, although this object appears twice in the diagram.

3.3.7 Building a Target

In addition to being unusually small, Minimath is unusual in that you can build a parser for it that uses only the assembly's stack while creating a useful result. Usually, you want a parser to build some kind of domain object. The `Assembly` class maintains a `target` attribute and provides methods for manipulating it, as Figure 3.1 shows. These methods let you build any kind of object from input text, with one restriction: The target of an `Assembly` object must be publicly cloneable (a topic addressed in the following section).

Consider designing a program that calculates the average length of words in a string. Let us take the approach of first creating a `RunningAverage` class that accepts word lengths and keeps track of the total number of words and their total length. With such a class available, you can create an `Assembler` class that works by updating a target `RunningAverage` object. Specifically, you can create an `AverageAssembler` class that

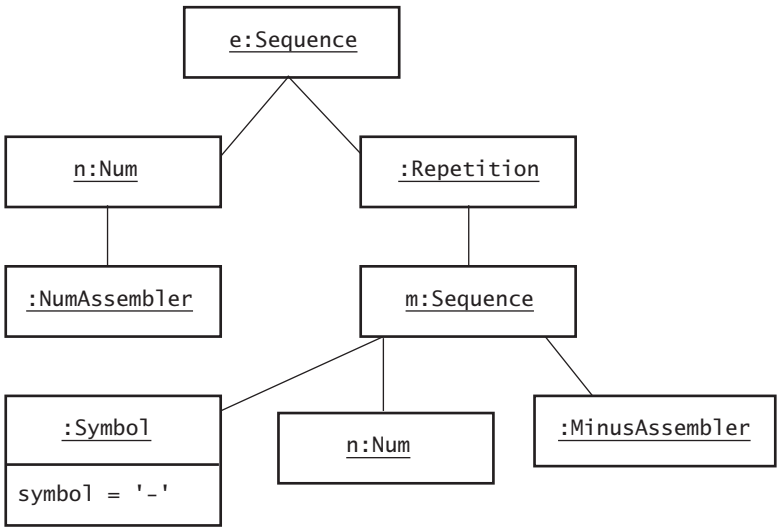

Figure 3.2 Minimath. This object diagram shows the structure of a parser composite that matches Minimath expressions such as `"3 - 2 - 1"`.

pops a word from the stack of an `Assembly` object and updates a target `RunningAverage` object with the length of the popped string. Here is `RunningAverage.java`:

```
package sjm.examples.design;

/**
 * Objects of this class maintain a running average. Each
 * number that is added with the <code>add</code> method
 * increases the count by 1, and the total by the amount
 * added.
 */
public class RunningAverage
    implements sjm.utensil.PubliclyCloneable {

    protected double count = 0;
    protected double total = 0;

/**
 * Add a value to the running average, increasing the count
 * by 1 and the total by the given value.
 */
public void add(double d) {
    count++;
    total += d;
}
```

```
/**
 * Return the average so far.
 */
public double average() {
    return total / count;
}

/**
 * Return a copy of this object.
 */
public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}
}
```

This class makes it easy to keep a running average. In your design, you can use a `RunningAverage` object as the target of an assembly. You can write an `AverageAssembler` class that expects this target. Here is the code for `AverageAssembler.java`:

```
package sjm.examples.design;

import sjm.parse.*;
import sjm.parse.tokens.*;
import sjm.engine.*;

public class AverageAssembler extends Assembler {

/**
 * Increases a running average, by the length of the string
 * on the stack.
 */
public void workOn(Assembly a) {
    Token t = (Token) a.pop();
    String s = t.sval();
    RunningAverage avg = (RunningAverage) a.getTarget();
    avg.add(s.length());
}
}
```

The `AverageAssembler` class updates a `RunningAverage` target object by the length of whatever string is on the input assembly's stack. Now you have the pieces you need to create a program that calculates the average length of words in a string. You will use a `RunningAverage` object as the target object of an assembly. You will use a `Word` object to recognize words, and you will plug an `AverageAssembler` object in to it. Then you can use a `Repetition` of the `Word` object to match a string of words. Figure 3.3 shows the objects you need.

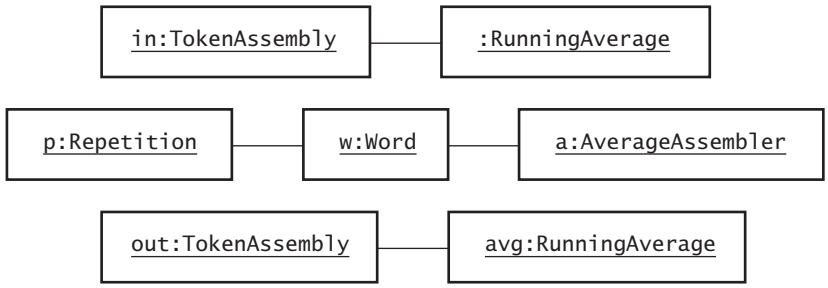

Figure 3.3 Object diagram for calculating a running average. An input assembly and an output assembly both have `RunningAverage` objects as their targets. The parser `p` is a repetition of a `Word` object that uses an `AverageAssembler` object to update a running average.

The parser `p` in Figure 3.3 matches an input assembly and creates an output assembly. The output has an updated clone of the `RunningAverage` object that reflects the average length of words in an input string. Here is a program that shows the objects in action:

```
package sjm.examples.design;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to use an assembler. The example shows how to
 * calculate the average length of words in a string.
 */
public class ShowAssembler {

public static void main(String args[]) {

    // As Polonius says, in "Hamlet"...
    String quote = "Brevity is the soul of wit";

    Assembly in = new TokenAssembly(quote);
    in.setTarget(new RunningAverage());
    Word w = new Word();
    w.setAssembler(new AverageAssembler());
    Parser p = new Repetition(w);

    Assembly out = p.completeMatch(in);

    RunningAverage avg = (RunningAverage) out.getTarget();
    System.out.println(
        "Average word length: " + avg.average());
}
}
```

The `main()` method in this class constructs the objects in Figure 3.3. This method wraps an input string in a `TokenAssembly` and sets the assembly's target to be a `RunningAverage` object. The method creates a parser that is a repetition of a `Word` object that uses an `AverageAssembler` to update the running average. The method creates an output assembly by matching the parser against the input assembly. Finally, the method shows the results of the parse. Running this class prints:

```
Average word length: 3.5
```

This example shows a typical collaboration of assemblies, assemblers, targets, and parsers:

- An input target plugs in to an input assembly.
- A parser establishes an assembler to work on the target. (In a composite parser, each subparser can have its own assembler.)
- The parser matches the input assembly, creating an output assembly.
- The output assembly contains an updated target.

This approach relies on the ability to clone an assembly as the parser pursues a match. Because the target is a part of the assembly, targets must also be cloneable.

3.3.8 Making a Target Cloneable

Assembly objects know almost nothing about the targets that they hold, but one message that an assembly sends to its target is `clone()`. This springs from the way that the parsers in `sjm.parse` model the nondeterminism inherent in recognizing most languages. These parsers use a backtracking mechanism, and this means that they must clone an assembly and its target each time they consume a token.

When you design your parser to create a target object from input text, your target class must have a public `clone()` method. To enforce this, the package `sjm.utensil` includes the interface `PubliclyCloneable`, whose code is as follows:

```
package sjm.utensil;

/**
 * Defines a type of object that anybody can clone.
 */
public interface PubliclyCloneable extends Cloneable {

public Object clone();
}
```

Providing a public clone method without implementing `PubliclyCloneable` is insufficient. The `Assembly.setTarget()` method must know that the object it receives is an instance of a class that implements a public `clone()`. It insists on this by receiving its input as datatype `PubliclyCloneable`.

To clone an object means to make a copy of the object. To make a class cloneable by any object, write a `clone()` method and declare that the class implements `PubliclyCloneable`. A typical cloneable class method looks like this:

```
package sjm.examples.cloning;

import sjm.utensil.*;

/**
 * This class shows a typical clone() method.
 */
public class Course implements PubliclyCloneable {
    protected Professor professor;
    protected Textbook textbook;

// gets and sets...

/**
 * Return a copy of this object.
 */
public Object clone() {
    try {
        Course copy = (Course) super.clone();
        copy.setProfessor((Professor) professor.clone());
        copy.setTextbook((Textbook) textbook.clone());
        return copy;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}
}
```

The sample method calls `super.clone()`, referring to the method `clone()` of class `Object`. This method creates a new object of the same class as the object that receives the `clone()` message; then it initializes each of the new object's fields by assigning them the same values as the corresponding fields in the copied object. This is a *shallow* copy as opposed to a *deep* copy, which would also make copies of each of an object's fields. You might think of `Object.clone()` as `Object.newObjectSameFields()`. In your `clone()` method, you must create a clone of each attribute in your class that is not a primitive type or a string. (Strings are immutable, so there is no need to clone them.)

The `Object.clone()` method throws `CloneNotSupportedException`, which you must handle. Surround your call to `super.clone()` in a `try/catch` block that throws `InternalError`. The `InternalError` exception is arguably the wrong exception to throw, because this error indicates a problem in the virtual machine. However, `Vector` and other important classes in Java throw an `InternalError` in this situation, and this book follows that precedent.

With an understanding of assemblers, assemblies, and parsers, you can begin to create meaningful new languages. Before you begin to code, however, it will prove helpful to have a way to work with parsers at a design level.

3.4 Grammars: A Shorthand for Parsers

A *grammar* is a collection of related parser definitions in which the definitions follow a standard shorthand. A goal of the design phase in software construction is to illustrate in compact form the important features that will appear in Java code. Consider again the code that builds a parser to recognize a description of a good cup of coffee:

```
package sjm.examples.introduction;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to create a composite parser.
 */
public class ShowComposite {

public static void main(String[] args) {

    Alternation adjective = new Alternation();
    adjective.add(new Literal("steaming"));
    adjective.add(new Literal("hot"));

    Sequence good = new Sequence();
    good.add(new Repetition(adjective));
    good.add(new Literal("coffee"));

    String s = "hot hot steaming hot coffee";
    Assembly a = new TokenAssembly(s);
    System.out.println(good.bestMatch(a));
}
}
```

You can more simply describe the `good` parser with the following shorthand, or grammar:

```
good      = adjective* "coffee";
adjective = "steaming" | "hot";
```

This shorthand relies on a few conventions, including showing literal values in quotes, showing alternation with a bar, and showing repetition with an asterisk. The point of using a grammar is that it is far more manageable than its corresponding Java code.

3.4.1 Standard Grammar Shorthand

This book observes the following conventions for writing a grammar, which is a compact definition of a parser.

1. Show the definition of a parser by giving its name, an equal sign, a subparser expression, and a semicolon. For example, the rule

   ```
   adjective = "steaming" | "hot";
   ```

   defines the makeup of an `adjective`.

2. Reference other subparsers as words that begin with a lowercase letter. For example, the rule

   ```
   good = adjective* "coffee";
   ```

   defines a `good` parser by referring to the `adjective` subparser.

3. Show a specific string to match by writing it in quotes. For example, the rule

   ```
   adjective = "steaming" | "hot";
   ```

   uses quotes to show that `"steaming"` and `"hot"` are specific strings to match.

4. Show a specific single character to match by writing it in single quotes, such as `'-'` in the Minimath rule:

   ```
   m = '-' Num;
   ```

   Note that a tokenizer will return many, but perhaps not all, arithmetic operators, Boolean operators, and punctuation marks as separate symbol tokens, rather than as parts of words. Section 9.6, "Tokenizer Lookup Tables," describes which characters the `Tokenizer` class treats (by default) as individual symbols.

5. Show repetition by using an asterisk ("`*`"). Consider the `good` grammar:

   ```
   good      = adjective* "coffee";
   adjective = "steaming" | "hot";
   ```

   The rule for `good` uses an asterisk to describe a string that begins with 0 or more `adjectives` and ends with `"coffee"`.

6. Imply sequence by writing subparsers next to each other. For example, the `good` grammar implies that `good` is a sequence of a-repetition-of-adjectives followed by the word `"coffee"`.
7. Show alternation by using a vertical bar. For example, the `adjective` rule declares that both `"steaming"` and `"hot"` are suitable adjectives.
8. Indicate precedence by using parentheses. For example, if you wanted to show the `good` grammar on one line, you could write

```
good = ("hot" | "steaming")* "coffee";
```

9. Show terminals as words that begin with a capital letter, such as `Num`, as in

```
phrase = '(' expression ')' | Num;
```

   Section 2.5, "Terminal Parsers," shows the terminals available in `sjm.parse.tokens`. Chapter 11, "Extending the Parser Toolkit," describes how to add new types of terminals.

10. Parameterize subparsers if it makes your grammar a better design. (This is an optional element of grammar design. It works only if you will code your subparsers as methods of a class. See Section 3.6, "Translating a Grammar to Code.") For example, consider a grammar for a small set of markup tags:

```
tag      = nameTag | roastTag | priceTag;
nameTag  = '<' "name" '>';
roastTag = '<' "roast" '>';
priceTag = '<' "price" '>';
```

    To avoid repeating the pattern of writing angle braces around a literal value, you can use a parameterized rule:

```
tag       = braces("name") | braces("roast") |
            braces("price");
braces(p) = '<' p '>';
```

    The parameter to `braces` is a parser, specifically a `CaselessLiteral` for `"name"`, `"roast"`, or `"price"`.

3.4.2 Top-Down Grammar Design

When you design a parser for a language, you can use either a top-down or a bottom-up approach to design. A bottom-up approach includes designing small parts of your parser that you know you will need. If you understand the goal of your parser, you may be able to design your assemblers, which can help guide the decomposition of the language you intend to recognize. When you reach the point of designing the grammar itself, you will most likely find that a top-down approach to grammar writing is natural and intuitive.

A top-down design approach begins by breaking the design problem into components. When you are using a top-down approach to write a parser, you state the design problem as, "Can I decompose this design into parts?" You can also state the design challenge as, "Can I represent the parser I want as a composition?" As it happens, parsers are always either terminals or composites of other parsers. The fact that parsers are composites makes the top-down design approach natural. Here is an effective algorithm for designing a new parser:

1. Define the parser you want as a composite of subparsers.
2. Repeat step 1 until every subparser is defined or is a terminal.

This algorithm creates a grammar that will often be sufficient for direct translation to Java code. Otherwise, you must transform the grammar before implementing it, a topic described in Section 3.6, "Translating a Grammar to Code." Before looking at grammar transformation, it is useful to walk through an example of applying the design algorithm to a small example.

3.5 Example: Designing a Grammar for a Track Robot

Figure 3.4 shows a miniature factory for which we want to design a command language. The heart of the factory is a track robot, a machine that can move forward and backward along a track. The robot can pick up material from conveyor belts and place

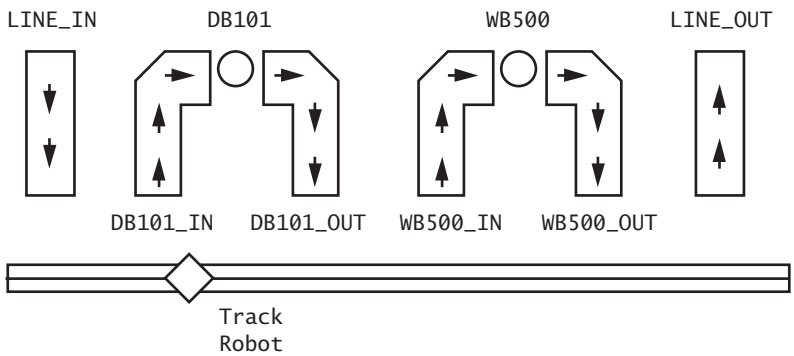

Figure 3.4 A track robot. In this miniature automated factory, a simple robot runs along a track, picking and placing material on machines.

the material on other conveyors. The robot transports material in metal containers called "carriers." The basic flow of the factory has the robot pick and place carriers so that they go through two processing machines and arrive at an output conveyor. Carriers have bar code labels that the robot is able to scan, letting the robot ascertain the identity of a carrier and arrive at a machine's output.

Humans tend to see the factory in Figure 3.4 as consisting of a track robot, two input and output conveyors, and two processing machines. The robot does not see the machines—it sees only conveyors—so our command language for the robot will be conveyor-centric. Here are some example commands for the robot:

```
pick carrier from LINE_IN
place carrier at DB101_IN
pick carrier from DB101_OUT
place carrier at WB500_IN
pick carrier from WB500_OUT
place carrier at LINE_OUT
scan DB101_OUT
```

3.5.1 A Track Robot Grammar

You want a command parser that will recognize the language that will drive the robot. Initially, your grammar is

```
command
```

You need to define the `command` parser in terms of other parsers, and you know that you want to recognize three commands. So you can give the first rule of the grammar as

```
command = pickCommand | placeCommand | scanCommand;
```

You need to refine this grammar, expanding every parser on the right side of a definition until every subparser is defined or is a terminal. Consider the subparser `pickCommand`. You know that the robot is willing to pick up carriers; you only have to tell it the location. For a definition that matches your sample strings, you can augment the grammar:

```
command     = pickCommand | placeCommand | scanCommand;
pickCommand = "pick" "carrier" "from" location;
```

You could make the words `"carrier"` and `"from"` optional, but let's keep the language simple for now. With simplicity in mind, let's also assume that each word in quotes in the grammar will be a `CaselessLiteral` so that users can type `"pick"`, `"Pick"`, or `"PICK"`.

Judging by the sample command strings, a location is always a single word:

```
command     = pickCommand | placeCommand | scanCommand;
pickCommand = "pick" "carrier" "from" location;
location    = Word;
```

You can complete the grammar design by defining the remaining subparsers—`placeCommand` and `scanCommand`:

```
command      = pickCommand | placeCommand | scanCommand;
pickCommand  = "pick" "carrier" "from" location;
placeCommand = "place" "carrier" "at" location;
scanCommand  = "scan" location;
location     = Word;
```

3.5.2 Checking for Left Recursion and Cycles

The grammar for the track robot language is complete, but only because it contains no left recursion and no cyclic dependencies. Left recursion exists if a parser's definition begins with itself. Cyclic dependencies exist if a parser's definition ultimately depends on itself. The grammar doesn't have these features so you can skip some steps described in Chapter 6, "Transforming a Grammar."

3.6 Translating a Grammar to Code

You can write the code of a parser directly from its grammar. You apply each principle of grammar translation in turn until the grammar becomes a set of Java statements that define a parser. The following principles apply:

- Treat quoted strings as `CaselessLiteral` objects.
- Create `Sequence` objects for sequences.
- Create `Alternation` objects for alternations.
- Translate `Terminal` references to objects.
- Create a subparser for each rule.
- Declare each subparser, or arrange subparsers as methods.
- Add a `start()` method.

3.6.1 Translate Quoted Strings

Treat each quoted word, such as `"pick"`, as a `CaselessLiteral`. For example, translate

```
pickCommand  = "pick" "carrier" "from" location;
```

to

```
pickCommand = new CaselessLiteral("pick")
              new CaselessLiteral("carrier")
              new CaselessLiteral("from")
              location;
```

This translation immediately begins to look like Java code, although it is not yet compilable. When all translations are complete, the result will be compilable code.

3.6.2 Translate Sequences

When you write a grammar, you imply sequences simply by showing two subparsers next to each other. For example,

```
placeCommand = "place" "carrier" "at" location;
```

implies

```
placeCommand  = new Sequence();
placeCommand.add(new CaselessLiteral("place"));
placeCommand.add(new CaselessLiteral("carrier"));
placeCommand.add(new CaselessLiteral("at"));
placeCommand.add(location);
```

Note that this is still not valid Java code, although you are approaching that goal. Specifically, you have not yet declared the type of `placeCommand`, nor have you established a strategy for referring to other subparsers. These translations follow shortly.

3.6.3 Translate Alternations

A vertical bar in a subparser definition means that the subparsers on either side of the bar may produce a successful match. When a series of vertical bars appears, you can create a single `Alternation` object. For example, translate

```
command = pickCommand | placeCommand | scanCommand;
```

to

```
command = new Alternation();
command.add(pickCommand);
command.add(placeCommand);
command.add(scanCommand);
```

This code looks almost like compilable Java, but you still have to translate references to subparsers.

3.6.4 Translate Terminals

Many of the terminals in the track robot language are literals, such as `"place"` and `"carrier"`. The only other terminal is `Word` in the `location` definition. To translate the grammar to Java code, replace each such terminal with a new terminal object of the specified type:

```
location = new Word();
```

3.6.5 Create a Subparser for Each Rule

The translation steps given so far leave each subparser as a word that begins with a lowercase letter, such as `pickCommand`. There are two strategies for translating subparser definitions into Java code. You can declare each subparser as an appropriate kind of parser, or you can arrange the subparsers as a class's methods.

3.6.6 Option 1: Declare Each Subparser

For a small language, you can make each subparser a separate variable. The track robot command language is a little too large for this approach. To illustrate, here is `RobotMonolithic.java`:

```
package sjm.examples.robot;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to create a parser and use it in a single
 * method.
 */
public class RobotMonolithic {

public static void main(String[] args) {
    Alternation command = new Alternation();
    Sequence pickCommand = new Sequence();
    Sequence placeCommand = new Sequence();
    Sequence scanCommand = new Sequence();
    Word location = new Word();
```

```
        command.add(pickCommand);
        command.add(placeCommand);
        command.add(scanCommand);

        pickCommand.add(new CaselessLiteral("pick"));
        pickCommand.add(new CaselessLiteral("carrier"));
        pickCommand.add(new CaselessLiteral("from"));
        pickCommand.add(location);

        placeCommand.add(new CaselessLiteral("place"));
        placeCommand.add(new CaselessLiteral("carrier"));
        placeCommand.add(new CaselessLiteral("at"));
        placeCommand.add(location);

        scanCommand.add(new CaselessLiteral("scan"));
        scanCommand.add(location);

        String s = "pick carrier from DB101_IN";

        System.out.println(
            command.bestMatch(new TokenAssembly(s)));
    }
    }
```

All the subparser declarations appear at the top of the code. The assignment statements build the `command` object into a parser for the track robot command language. Running this class prints the following:

```
[pick, carrier, from, DB101_IN]
pick/carrier/from/DB101_IN^
```

The output shows that the `command` parser can completely parse at least one sample element of the language.

3.6.7 Option 2: Arrange Subparsers as Methods

For readability, you can create a method for each subparser of a grammar. For example, you can lift out the preceding code that creates the `command` object and place it in a method called `command()`. You can reapply this strategy, creating a method for each subparser. Because all the subparsers except `command` are useful only in constructing the `command` subparser, it is a good idea to make them protected and not public. Subparsers such as `location` are not intended for public use but might be overridden in a subclass.

Refactoring the `RobotMonolithic` class to apply the strategy of making subparsers methods results in the following:

```
package sjm.examples.robot;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Provide an example of a class that affords a parser for
 * the "robot" command language. This class is a refactored
 * version of the <code>RobotMonolithic</code> class, with
 * one method for each subparser in the robot language.
 */
public class RobotRefactored {

public Parser command() {
    Alternation a = new Alternation();
    a.add(pickCommand());
    a.add(placeCommand());
    a.add(scanCommand());
    return a;
}

protected Parser pickCommand() {
    Sequence s = new Sequence();
    s.add(new CaselessLiteral("pick"));
    s.add(new CaselessLiteral("carrier"));
    s.add(new CaselessLiteral("from"));
    s.add(location());
    return s;
}

protected Parser placeCommand() {
    Sequence s = new Sequence();
    s.add(new CaselessLiteral("place"));
    s.add(new CaselessLiteral("carrier"));
    s.add(new CaselessLiteral("at"));
    s.add(location());
    return s;
}

protected Parser scanCommand() {
    Sequence s = new Sequence();
    s.add(new CaselessLiteral("scan"));
    s.add(location());
    return s;
}

protected Parser location() {
    return new Word();
}
}
```

Here's a class that uses the refactored parser class:

```
package sjm.examples.robot;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to use a parser class that arranges its
 * subparsers as methods.
 */
public class ShowRobotRefactored {

public static void main(String[] args) {
    Parser p = new RobotRefactored().command();
    String s = "place carrier at WB500_IN";
    System.out.println(p.bestMatch(new TokenAssembly(s)));
}
}
```

Running this class prints the following:

```
[place, carrier, at, WB500_IN]
place/carrier/at/WB500_IN^
```

The class `RobotRefactored` is a refactoring of `RobotMonolithic`, with the subparsers arranged as a coordinated set of methods. This approach can lead to an infinite loop if rules in your grammar refer to each other in a cycle. Fortunately, you can eliminate such loops by using lazy initialization; Section 6.5 explains how. Many grammars, including the track robot grammar, are small and acyclic, so we defer this topic for now.

In the refactoring, I changed the variable names. This is an esthetic choice. You can decide whether you think it is easier to read and understand this:

```
protected Parser scanCommand() {
    Sequence s = new Sequence();
    s.add(new CaselessLiteral("scan"));
    s.add(location());
    return s;
}
```

or this:

```
protected Parser scanCommand() {
    Sequence scanCommand = new Sequence();
    scanCommand.add(new CaselessLiteral("scan"));
    scanCommand.add(location());
    return scanCommand;
}
```

3.6.8 Add a Start Method

A user of your class needs to be able to tell which subparser is the "primary" parser, the one that matches a useful language. To tell your prospective user which parser to use, introduce a `start()` method that returns the primary parser. For example, you could add the following method to `RobotRefactored` to make it easy for a user of the class to find the primary parser:

```
/**
 * Returns a parser that will recognize a command for a
 * track robot, and build a corresponding command object.
 */
public static Parser start() {
    return new RobotParser().command();
}
```

Making this method static allows a user to simply call `start()` as a class method.

3.7 Completing a Parser

The class `RobotRefactored` is the result of a translation from the track robot command language grammar into code. It is complete in that it provides a parser that recognizes the desired language. This parser is not complete, however, in the sense of doing anything useful. To go beyond recognition of a language to taking some useful action based on the recognition, a parser must control the pushing of terminals onto an assembly's stack, and it must plug assemblers in to the appropriate subparsers.

3.7.1 Control Pushing

By default, all terminals push whatever they recognize onto an assembly's stack. For most terminals, this is a useful and often necessary function. For example, when a parser recognizes a `Word` or a `Num`, the parser usually needs to do something with whatever `Word` or `Num` it recognizes. On the other hand, when a parser recognizes a `Literal` or `CaselessLiteral`, such as `"carrier"`, the parser normally does not need to do any work with the literal it sees. For example, consider the subparser for `pickCommand`:

```
pickCommand = "pick" "carrier" "from" location;
```

In this subparser, the words `"pick"`, `"carrier"`, and `"from"` serve to identify a particular type of command. The `pickCommand` subparser successfully matches only text that begins with these three words. There is no reason to stack these words; you know what they are, and you know you must be recognizing a `pickCommand` if the

match succeeds. Typically, you will want to ask all the `Literal` parsers in your parser to discard the terminal they see. To ask a `Literal` not to push itself, send it a `discard()` message. Making this change in `RobotRefactored.pickCommand()` results in the following:

```
protected Parser pickCommand() {
    Sequence s = new Sequence();
    s.add(new CaselessLiteral("pick").discard());
    s.add(new CaselessLiteral("carrier").discard());
    s.add(new CaselessLiteral("from").discard());
    s.add(location());
    return s;
}
```

Keeping the literals from pushing means that after a `pickCommand()` matches an assembly's text, the assembly's stack will contain only the value of the `location` for the command. You will see shortly how to plug in assemblers to work on the assembly's stack and target.

3.7.2 Design the Target

Having developed the code to recognize a robot command language, the next step is to arrange for the parser to actually build a command object from input command text. In this example, the target of parsing text is an object from a hierarchy of commands. Figure 3.5 shows the targets for the robot language, namely the three subclasses of `RobotCommand`.

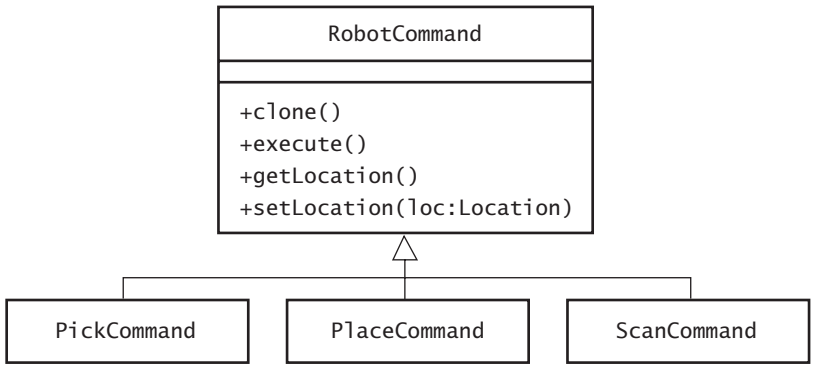

Figure 3.5 A command hierarchy. You can run a little factory with commands for picking, placing, and scanning carriers.

Here is the code for RobotCommand:

```
package sjm.examples.robot;

/**
 * A <code>RobotCommand</code> encapsulates the work that
 * lies behind a high-level command such as "pick carrier from
 * input1". In this package, the commands are just sample
 * targets of a parser; their <code>execute()</code> methods
 * are not implemented.
 */
public class RobotCommand
    implements sjm.utensil.PubliclyCloneable {

    protected String location;

/**
 * Return a copy of this object. If the location attribute
 * becomes something more complicated than a String, then
 * this method will become insufficient if location is not
 * immutable.
 */
public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}

/**
 * If we were really driving a factory, this is where we
 * would turn high-level commands into the protocols that
 * various machines would understand. For example, a pick
 * command might send messages to both a conveyor and
 * a track robot.
 */
public void execute() {
}

/**
 * Return the location that this command is for.
 */
public String getLocation() {
    return location;
}

/**
 * Set the location for this command.
 */
public void setLocation(String location) {
```

```
    this.location = location;
}
}
```

The purpose of a `RobotCommand` object is to operate real equipment when the command's `execute()` method runs. In a real factory system you would have to write the code that makes this execution happen. Each subclass of `RobotCommand` would override `execute()` appropriately. In this example you are aiming only to translate text into the right kind of command object, so the subclasses are empty. For example, the code for `PickCommand` is

```
package sjm.examples.robot;

/**
 * Just for demonstration.
 */
public class PickCommand extends RobotCommand {

/**
 * Return a textual description of this object.
 */
public String toString() {
    return "pick " + location;
}
}
```

3.7.3 Plug In Assemblers

The assemblers for your parser set an assembly's target and inform the target of its associated location. Each subclass of the command needs a corresponding assembler to set the appropriate target. For example, you will need a `PickAssembler` class:

```
package sjm.examples.robot;

import sjm.parse.*;
import sjm.parse.tokens.Token;

/**
 * Sets an assembly's target to be a <code>PickCommand
 * </code> object and notes its location.
 */
public class PickAssembler extends Assembler {

public void workOn(Assembly a) {
    PickCommand pc = new PickCommand();
    Token t = (Token) a.pop();
    pc.setLocation(t.sval());
    a.setTarget(pc);
}
}
```

The approach taken here assumes the parser will parse a single command and construct a corresponding `RobotCommand` object. Which subclass of `RobotCommand` to instantiate and to set as the target of the parse depends on the input text. The `PickAssembler` object plugs in to the `pickCommand` subparser, which will successfully match a `"pick"` command. The assembler's `workOn()` method executes after an (entire) `pickCommand` subparser matches. At this time, a pick location will be on the stack. When the `workOn()` method executes, it creates a `PickCommand` object and sets this object as the target of the parse. The assembler pops the location object and uses it to establish the location of the `PickCommand` object.

You can now convert the code for `RobotRefactored` into `RobotParser`, plugging in the assemblers and adding some comments. The result is as follows:

```
package sjm.examples.robot;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * This class's start() method provides a parser that
 * will recognize a command for a track robot and build a
 * corresponding command object.
 * <p>
 * The grammar for the language that this class recognizes
 * is:
 *
 * <blockquote><pre>
 *     command      = pickCommand | placeCommand |
 *                    scanCommand;
 *     pickCommand  = "pick" "carrier" "from" location;
 *     placeCommand = "place" "carrier" "at" location;
 *     scanCommand  = "scan" location;
 *     location     = Word;
 * </pre></blockquote>
 */
public class RobotParser {

/**
 * Returns a parser that will recognize a command for a
 * track robot and build a corresponding command
 */
public static Parser start() {
    return new RobotParser().command();
}

/**
 * Returns a parser that will recognize a command for a
 * track robot and build a corresponding command object.
 *
 * (This method returns the same value as
```

```
     * <code>start()</code>).
     */
    public Parser command() {
        Alternation a = new Alternation();
        a.add(pickCommand());
        a.add(placeCommand());
        a.add(scanCommand());
        return a;
    }

    /*
     * Returns a parser that will recognize the grammar:
     *
     *     pickCommand = "pick" "carrier" "from" location;
     */
     protected Parser pickCommand() {
        Sequence s = new Sequence();
        s.add(new CaselessLiteral("pick"));
        s.add(new CaselessLiteral("carrier"));
        s.add(new CaselessLiteral("from"));
        s.add(location());
        s.setAssembler(new PickAssembler());
        return s;
    }

    /*
     * Returns a parser that will recognize the grammar:
     *
     *     placeCommand = "place" "carrier" "at" location;
     */
     protected Parser placeCommand() {
        Sequence s = new Sequence();
        s.add(new CaselessLiteral("place"));
        s.add(new CaselessLiteral("carrier"));
        s.add(new CaselessLiteral("at"));
        s.add(location());
        s.setAssembler(new PlaceAssembler());
        return s;
    }

    /*
     * Returns a parser that will recognize the grammar:
     *
     *     scanCommand = "scan" location;
     */
    protected Parser scanCommand() {
        Sequence s = new Sequence();
        s.add(new CaselessLiteral("scan"));
        s.add(location());
        s.setAssembler(new ScanAssembler());
        return s;
    }
```

```
/*
 * Returns a parser that will recognize the grammar:
 *
 *     location = Word;
 */
protected Parser location() {
    return new Word();
}
}
```

You can use the `RobotParser.start()` parser as follows:

```
package sjm.examples.robot;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to use the <code>RobotParser</code> class.
 */
public class ShowRobotParser {

public static void main(String[] args) {
    Parser p = RobotParser.start();

    String[] tests = new String[]{
        "pick carrier from LINE_IN",
        "place carrier at DB101_IN",
        "pick carrier from DB101_OUT",
        "place carrier at WB500_IN",
        "pick carrier from WB500_OUT",
        "place carrier at LINE_OUT",
        "scan DB101_OUT"};

    for (int i = 0; i < tests.length; i++) {
        TokenAssembly ta = new TokenAssembly(tests[i]);
        Assembly out = p.bestMatch(ta);
        System.out.println(out.getTarget());
    }
}
}
```

Running this class prints the results of parsing a few sample commands:

```
pick LINE_IN
place DB101_IN
pick DB101_OUT
place WB500_IN
pick WB500_OUT
place LINE_OUT
scan DB101_OUT
```

These are the results of the `toString()` methods of the commands built by the `RobotParser.start()` parser. If the command target objects were wired into a factory with functional `execute()` methods, you could use these commands to control the factory.

3.8 Summary

Building a new parser starts with envisioning the language you want to recognize. Write a few sample sentences of the language that you want, and write a grammar that comprehends these examples. A grammar shows the pattern of strings in your language and serves as a design document. Next, translate your grammar to code and verify that your parser recognizes the sample strings of your language. Once you get a parser working that recognizes your examples, you can add more grammar rules. You can work iteratively to build the complete language you want to recognize. At some point, you must start creating the auxiliary classes that let your parser do more than just recognize an input string. These supporting classes are assemblers and potentially a target. After a subparser recognizes text, the subparser's assembler can work on the assembly that contains the text. This work may be limited to the assembly's stack, or it may include changes to the assembly's target. You have complete control over how you define a target class except that your class must implement `PubliclyCloneable`.

Work iteratively, creating your parser as a composition of subparsers and plugging in assemblers that build a target. In a short time you can learn to create powerful new languages from these few steps.