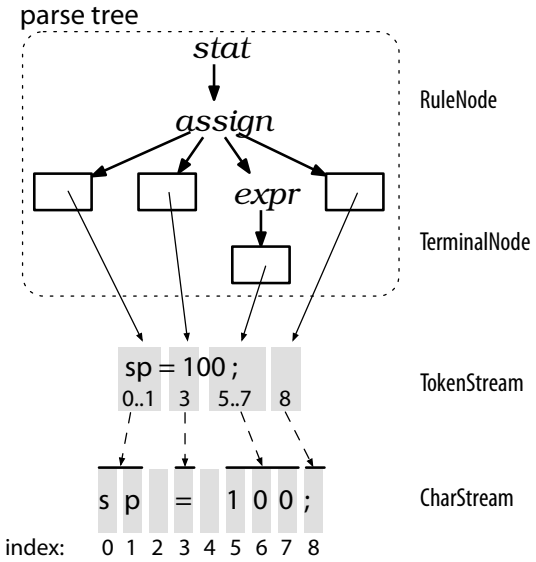
Parsers by themselves test input sentences only for language membership and build a parse tree. That's crucial stuff, but it's time to see how language applications use parse trees to interpret or translate the input.

2.4 Building Language Applications Using Parse Trees

To make a language application, we have to execute some appropriate code for each input phrase or subphrase. The easiest way to do that is to operate on the parse tree created automatically by the parser. The nice thing about operating on the tree is that we're back in familiar Java territory. There's no further ANTLR syntax to learn in order to build an application.

Let's start by looking more closely at the data structures and class names ANTLR uses for recognition and for parse trees. A passing familiarity with the data structures will make future discussions more concrete.

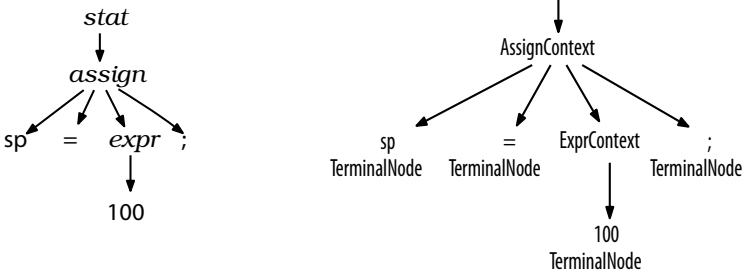
Earlier we learned that lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree. The corresponding ANTLR classes are CharStream, Lexer, Token, Parser, and ParseTree. The "pipe" connecting the lexer and parser is called a TokenStream. The diagram below illustrates how objects of these types connect to each other in memory.



These ANTLR data structures share as much data as possible to reduce memory requirements. The diagram shows that leaf (token) nodes in the parse tree are containers that point at tokens in the token stream. The tokens record start and stop character indexes into the CharStream, rather than making copies

of substrings. There are no tokens associated with whitespace characters (indexes 2 and 4) since we can assume our lexer tosses out whitespace.

The figure also shows ParseTree subclasses RuleNode and TerminalNode that correspond to subtree roots and leaf nodes. RuleNode has familiar methods such as getChild() and getParent(), but RuleNode isn't specific to a particular grammar. To better support access to the elements within specific nodes, ANTLR generates a RuleNode subclass for each rule. The following figure shows the specific classes of the subtree roots for our assignment statement example, which are StatContext, AssignContext, and ExprContext:



Parse tree

Parse tree node class names

These are called *context* objects because they record everything we know about the recognition of a phrase by a rule. Each context object knows the start and stop tokens for the recognized phrase and provides access to all of the elements of that phrase. For example, AssignContext provides methods ID() and expr() to access the identifier node and expression subtree.

Given this description of the concrete types, we could write code by hand to perform a depth-first walk of the tree. We could perform whatever actions we wanted as we discovered and finished nodes. Typical operations are things such as computing results, updating data structures, or generating output. Rather than writing the same tree-walking boilerplate code over again for each application, though, we can use the tree-walking mechanisms that ANTLR generates automatically.

2.5 Parse-Tree Listeners and Visitors

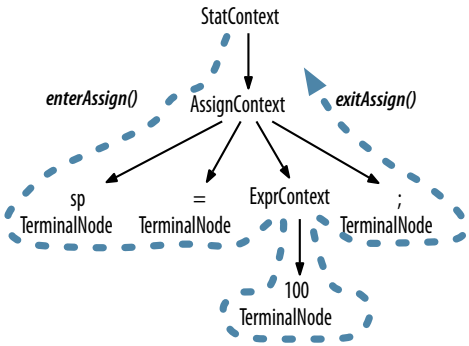
ANTLR provides support for two tree-walking mechanisms in its runtime library. By default, ANTLR generates a parse-tree *listener* interface that responds to events triggered by the built-in tree walker. The listeners themselves are exactly like SAX document handler objects for XML parsers. SAX listeners receive notification of events like startDocument() and endDocument(). The

methods in a listener are just callbacks, such as we'd use to respond to a checkbox click in a GUI application. Once we look at listeners, we'll see how ANTLR can also generate tree walkers that follow the visitor design pattern.[1]

Parse-Tree Listeners

To walk a tree and trigger calls into a listener, ANTLR's runtime provides class ParseTreeWalker. To make a language application, we build a ParseTreeListener implementation containing application-specific code that typically calls into a larger surrounding application.

ANTLR generates a ParseTreeListener subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule assign, for example, it triggers enterAssign() and passes it the AssignContext parse-tree node. After the walker visits all children of the assign node, it triggers exitAssign(). The tree diagram shown below shows ParseTreeWalker performing a depth-first walk, represented by the thick dashed line.



It also identifies where in the walk ParseTreeWalker calls the enter and exit methods for rule assign. (The other listener calls aren't shown.)

And the diagram in Figure 1, *ParseTreeWalker call sequence*, on page 19 shows the complete sequence of calls made to the listener by ParseTreeWalker for our statement tree.

The beauty of the listener mechanism is that it's all automatic. We don't have to write a parse-tree walker, and our listener methods don't have to explicitly visit their children.

1. http://en.wikipedia.org/wiki/Visitor_pattern

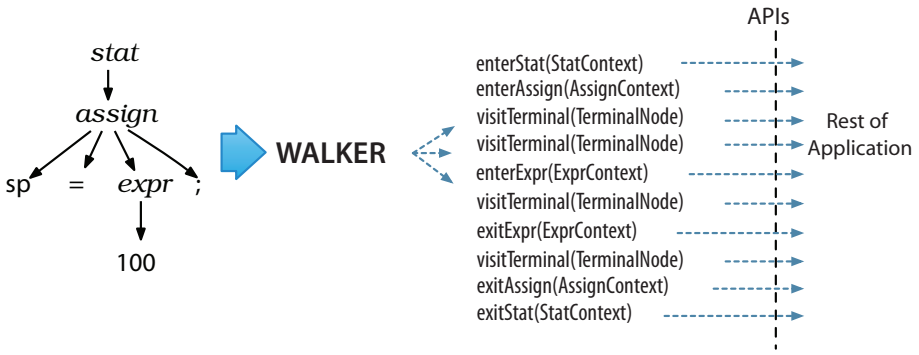
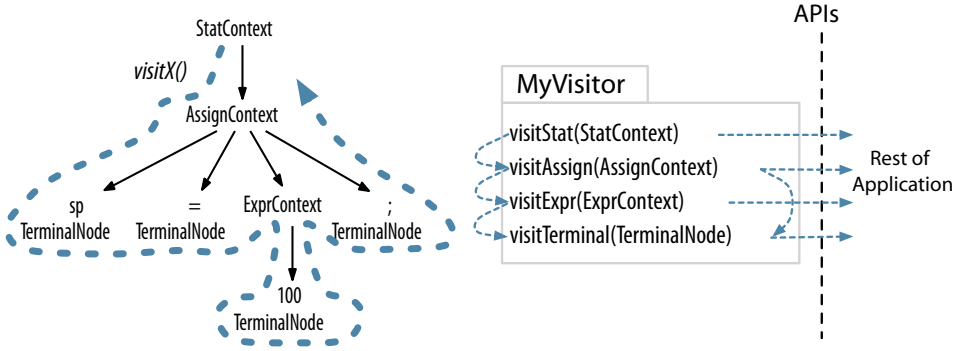


Figure 1—ParseTreeWalker call sequence

Parse-Tree Visitors

There are situations, however, where we want to control the walk itself, explicitly calling methods to visit children. Option -visitor asks ANTLR to generate a visitor interface from a grammar with a visit method per rule. Here's the familiar visitor pattern operating on our parse tree:



The thick dashed line shows a depth-first walk of the parse tree. The thin dashed lines indicate the method call sequence among the visitor methods. To initiate a walk of the tree, our application-specific code would create a visitor implementation and call visit().

```
ParseTree tree = ... ; // tree is result of parsing
MyVisitor v = new MyVisitor();
v.visit(tree);
```

ANTLR's visitor support code would then call visitStat() upon seeing the root node. From there, the visitStat() implementation would call visit() with the children as arguments to continue the walk. Or, visitMethod() could explicitly call visitAssign(), and so on.

ANTLR gives us a leg up over writing everything ourselves by generating the visitor interface and providing a class with default implementations for the visitor methods. This way, we avoid having to override every method in the interface, letting us focus on just the methods of interest. We'll learn all about visitors and listeners in Chapter 7, *Decoupling Grammars from Application-Specific Code*, on page 109.

Parsing Terms

This chapter introduced a number of important language recognition terms.

Language A language is a set of valid sentences; sentences are composed of phrases, which are composed of subphrases, and so on.

Grammar A grammar formally defines the syntax rules of a language. Each rule in a grammar expresses the structure of a subphrase.

Syntax tree or parse tree This represents the structure of the sentence where each subtree root gives an abstract name to the elements beneath it. The subtree roots correspond to grammar rule names. The leaves of the tree are symbols or tokens of the sentence.

Token A token is a vocabulary symbol in a language; these can represent a category of symbols such as "identifier" or can represent a single operator or keyword.

Lexer or tokenizer This breaks up an input character stream into tokens. A lexer performs lexical analysis.

Parser A parser checks sentences for membership in a specific language by checking the sentence's structure against the rules of a grammar. The best analogy for parsing is traversing a maze, comparing words of a sentence to words written along the floor to go from entrance to exit. ANTLR generates top-down parsers called *ALL(*)* that can use all remaining input symbols to make decisions. Top-down parsers are goal-oriented and start matching at the rule associated with the coarsest construct, such as `program` or `inputFile`.

Recursive-descent parser This is a specific kind of top-down parser implemented with a function for each rule in the grammar.

Lookahead Parsers use lookahead to make decisions by comparing the symbols that begin each alternative.

So, now we have the big picture. We looked at the overall data flow from character stream to parse tree and identified the key class names in the ANTLR runtime. And we just saw a summary of the listener and visitor mechanisms used to connect parsers with application-specific code. Let's make this all more concrete by working through a real example in the next chapter.