Extracted from:

The Definitive ANTLR 4 Reference

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Definitive ANTLR 4 Reference

Terence Parr

Edited by Susannah Davidson Pfalzer

Printed in the United States of America.
ISBN-13: 978-1-93435-699-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—September 19, 2012

Now that we've got ANTLR installed and have some idea how to build and run a small example, we're going to look at the big picture. In this chapter, we'll learn about the important processes, terminology, and data structures associated with language applications. As we go along, we'll identify the key ANTLR objects and learn a little bit about what ANTLR does for us behind the scenes. To lock things in, we'll work through a small but useful project that processes nested arrays of integers like `{1, {2, 3}}`.

2.1 Let's Get Meta!

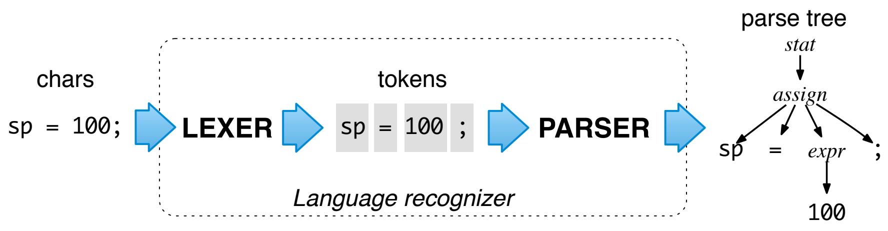
To implement a language we have to build an application that reads valid sentences and reacts appropriately to the phrases (sentence fragments) and input symbols it discovers. Broadly speaking, if an application computes or "executes" sentences, we call that application an *interpreter*. Examples include calculators, configuration file readers and Python interpreters. If we're converting sentences from one language to another, we call that application a *translator*. Examples include Java to C# converters and compilers.

In order to react appropriately, the interpreter or translator has to recognize all of the valid sentences, phrases, and subphrases of a particular language. Recognizing a phrase means we can identify the various components and can differentiate it from other phrases. For example, we recognize input `sp = 100;` as a programming language assignment statement. That means we know that `sp` is the assignment target and `100` is the value to store. Similarly, if we were recognizing English sentences, we'd identify the parts of speech, such as the subject, predicate, and object. The assignment is also clearly not an import statement. The actual language application would then perform an abstract operation like `performAssignment("sp", 100)` or `translateAssignment("sp", 100)`.

Programs that recognize languages are called *parsers* or *syntax analyzers*. *Syntax* refers to the rules governing language membership and in this book we're going to build ANTLR *grammars* to specify language syntax. A grammar is just a set of rules, each one expressing the structure of a phrase. The ANTLR tool translates grammars to parsers that look remarkably similar to what we'd build by hand. (ANTLR is a program that writes other programs.) Grammars themselves follow the syntax of a language optimized for specifying other languages: ANTLR's *meta-language*.

Parsing is much easier if we break it down into two similar but distinct tasks or stages. The separate stages mirror how our brains read English text. We don't read a sentence character by character. Instead, we perceive a sentence as a stream of words. The human brain subconsciously groups character sequences into words and looks them up in a dictionary before recognizing grammatical structure. This process is more obvious if we're reading Morse code because we have to convert the dots and dashes to characters before reading a message. It's also obvious when reading long words like Humuhumunukunukuapua'a, the Hawaiian state fish.

The process of grouping characters into words or symbols (*tokens*) is called *lexical analysis* or simply *tokenizing*. We call a program that tokenizes the input a *lexer*. The second stage is the actual parser and feeds off of these tokens to recognize the sentence structure, in this case an assignment statement. By default, ANTLR-generated parsers build a data structure called a *parse tree* or *syntax tree* that records how the parser recognized the structure of the input sentence and component phrases. The following diagram illustrates the basic data flow of a language recognizer.



The interior nodes of the parse tree are phrase names that group and identify their children. The root node is the most abstract phrase name, in this case *stat* (short for statement). The leaves of a parse tree are always the input tokens. Sentences, linear sequences of symbols, are really just serializations of parse trees we humans grok natively in hardware. To get an idea across to someone, we have to conjure up the same parse tree in their heads using a word stream.

Parse trees sit between a language recognizer and an interpreter or translator implementation. They are extremely effective data structures because they contain all of the input and complete knowledge of how the parser grouped the symbols into phrases. Better yet, they are easy to understand and the parser can generate them automatically.

Parse trees also allow us to reuse a single parser for any application that needs to recognize the same language because we don't have to inject application-specific code snippets into the parsing process itself. (ANTLR and other parser generators allow us to put raw code snippets into the grammar.)

Parse trees are also useful for translations that require multiple tree walks because of computation dependencies where one stage needs information from a previous stage. In other cases, an application is just a heck of a lot easier to code and test in multiple stages because it's so complex. Rather than reparse the input characters for each stage, we can just walk the parse tree multiple times, which is much more efficient.

Because we specify phrase structure with a set of rules, parse tree subtree roots correspond to grammar rule names. As a preview of things to come, here's the grammar rule that corresponds to the first level of the *assign* subtree from the diagram:

```
assign : ID '=' expr ';' ; // match an assignment statement like "sp = 100;"
```

Understanding how ANTLR translates such rules into human-readable parsing code is fundamental to using and debugging grammars, so let's dig deeper into how parsing works.

2.2 Implementing Parsers

The ANTLR tool generates *recursive-descent parsers* from grammar rules such as assign that we just saw. Recursive-descent parsers are really just a collection of recursive methods, one per rule. The *descent* term refers to the fact that parsing begins at the root of a parse tree and proceeds towards the leaves (tokens). The rule we invoke first, the *start symbol*, becomes the root of the parse tree. That would mean calling method stat() for the parse tree in the previous section. A more general term for this kind of parsing is *top-down parsing*; recursive-descent parsers are just one kind of top-down parser implementation.

To get an idea of what recursive-descent parsers look like, here's the (slightly cleaned up) method that ANTLR generates for rule assign:

```
// assign : ID '=' expr ';' ;
void assign() {     // method generated from rule assign
    match(ID);      // compare ID to current input symbol then consume
    match('=');
    expr();         // match an expression by calling expr()
    match(';');
}
```

The cool part about recursive-descent parsers is that the call graph traced out by invoking methods stat(), assign(), and expr() mirrors the interior parse tree nodes. (Take a quick peek back at the parse tree figure.) The calls to match() correspond to the parse tree leaves. To build a parse tree manually, we'd insert "add new subtree root" operations at the start of each rule method and "add new leaf node" to match().

Method assign() only has one thing to do; it just checks to make sure everything is present and in the right order. When the parser enters assign(), it doesn't have to choose between more than one *alternative*. An alternative is one of the choices on the right-hand side of a rule definition. For example, the stat rule that invokes assign likely has a list of other kinds of statements:

```
/** Match any kind of statement starting at the current input position */
stat: assign          // First alternative ('|' is alternative separator)
    | ifstat          // Second alternative
    | whilestat
   ...
    ;
```

A parsing rule for stat looks like a switch:

```
void stat() {
    switch ( «current input token» ) {
        CASE ID    : assign(); break;
        CASE IF    : ifstat(); break; // IF is token type for keyword 'if'
        CASE WHILE : whilestat(); break;
        ...
        default    : «raise no viable alternative exception»
    }
}
```

Method stat() has to make a *parsing decision* by examining the next input token. Parsing decisions predict which alternative will be successful. In this case, seeing a WHILE keyword predicts the third alternative of rule stat. Rule method stat() therefore calls whilestat(). You might've heard the term *lookahead token* before; that's just the next input token. A lookahead token is any token that the parser sniffs before matching and consuming it.

Sometimes, the parser needs lots of lookahead tokens to predict which alternative will succeed. It might even have to consider all tokens from the current position until the end of file! ANTLR silently handles all of this for you, but it's helpful have a basic understanding of decision-making so debugging generated parsers is easier.

To visualize parsing decisions, imagine a maze with a single entrance and exit that has words written on the floor. Every sequence of words along a path from entrance to exit represents a sentence. The structure of the maze is analogous to the rules in a grammar that define a language. To test a sentence for membership in a language, we compare the sentence's words with the words along the floor as we traverse the maze. If we can get to the exit by following the sentence's words, that sentence is valid.

To navigate the maze, we must choose a valid path at each fork, just as we must choose alternatives in a parser. We have to decide which path to take by comparing the next word or words in our sentence with the words visible down each path emanating from the fork. The words we can see from the fork are analogous to lookahead tokens. The decision is pretty easy when each path starts with a unique word. In rule stat, each alternative begins with unique token and so stat() can distinguish the alternatives by looking at the first lookahead token.

When the words starting each path from a fork overlap, a parser needs to look farther ahead, scanning for words that distinguish the alternatives. ANTLR automatically throttles the amount of lookahead up-and-down as necessary for each decision. If the lookahead is the same down multiple paths to the exit (end of file), there are multiple interpretations of the current input phrase. Resolving such ambiguities is our next topic. After that, we'll figure out how to use parse trees to build language applications.

2.3 You Can't Put Too Much Water into a Nuclear Reactor

An ambiguous phrase or sentence is one that has more than one interpretation. In other words, the words fit more than one grammatical structure. This section title is an ambiguous sentence from a Saturday Night Live sketch I saw years ago. The characters weren't sure if they should be careful *not* to put too much water into the reactor or they should put lots of water into the reactor.

For Whom No Thanks Is Too Much

One of my favorite ambiguous sentences is on the dedication page of my friend Kevin's Ph.D. thesis: "*To my Ph.D. supervisor, for whom no thanks is too much.*" It's unclear whether he was grateful or ungrateful. Kevin claimed it was the latter and so I asked why he had taken a postdoc job working for the same guy. His reply: "Revenge."

Ambiguity can be funny in natural language but causes problems for computer-based language applications. To interpret or translate a phrase, a program has to uniquely identify the meaning. That means we have to provide unambiguous grammars so that the generated parser can match each input phrase in exactly one way.

We haven't learned about grammars yet, but let's include a few ambiguous grammars here to make the notion of ambiguity more concrete. We can refer back to this section when we run into ambiguities later.

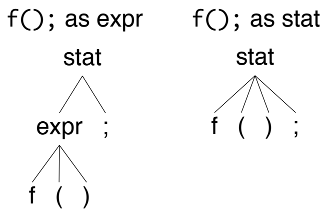
Some ambiguous grammars are obvious:

```
stat: ID '=' expr ';'  // match an assignment; can match "f();"
    | ID '=' expr ';'  // oops! an exact duplicate of previous alternative
    ;
expr: INT ;
```

Most of the time, though, the ambiguity will be more subtle as in the following grammar that can match a function call via both alternatives of rule stat.

```
stat: expr ';'         // expression statement
    | ID '(' ')' ';'  // function call statement
    ;
expr: ID '(' ')'
    | INT
    ;
```

Here are the two interpretations of input f(); starting in rule stat:



Since most language inventors design their syntax to be unambiguous, an ambiguous grammar is analogous to a programming bug. We need to reorganize the grammar to present a single choice to the parser for each input phrase. If the parser detects an ambiguous phrase, it resolves the ambiguity by choosing the first alternative involved in the decision. In this case, the parser would choose the interpretation of f(); on the left.

Ambiguities can occur in the lexer as well as the parser, but ANTLR resolves them so the rules behave naturally. ANTLR resolves lexical ambiguities by matching the input string to the rule specified first in the grammar. To see how this works, let's look at the extremely common ambiguity between keywords and identifier rules. Keyword begin is also an identifier, at least lexically, so the lexer can match b-e-g-i-n to either rule:

```
BEGIN : 'begin' ;  // match b-e-g-i-n sequence; ambiguity resolves to BEGIN
ID    : [a-z]+ ;   // match one or more of any lowercase letter
```

Sometimes the syntax for language is just plain ambiguous and no amount of grammar reorganization will change that fact. For example, the natural grammar for arithmetic expressions can interpret input such as 1+2*3 in two ways. Either by performing the operations left to right (as Smalltalk does) or in precedence order like most languages. We'll learn how to implicitly specify the operator precedence order for expressions in Section 4.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page ?.

The venerable C language exhibits another kind of ambiguity, which we can resolve using context information such as how an identifier is defined. Consider code snippet i*j;. It looks like an expression, but its meaning depends on whether i is a type name or variable. If it's a type name, then the snippet isn't an expression. It's a declaration of variable j as a pointer to type i. We'll see how to resolve these ambiguities in Chapter 10, *Altering the Parse with Semantic Predicates*, on page ?.

Parsers by themselves only test input sentences 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.