Chapter 3
Tokens to Syntax Tree — Syntax Analysis

There are two ways of doing parsing: top-down and bottom-up. For top-down parsers, one has the choice of writing them by hand or having them generated automatically, but bottom-up parsers can only be generated. In all three cases, the syntax structure to be recognized is specified using a context-free grammar; grammars were discussed in Section 1.8. Sections 3.2 and 3.5.10 detail considerations concerning error detection and error recovery in syntax analysis.

Roadmap

3 Tokens to Syntax Tree — Syntax Analysis 115
   3.1 Two classes of parsing methods 117
   3.2 Error detection and error recovery 120
   3.3 Creating a top-down parser manually 122
   3.4 Creating a top-down parser automatically 126
   3.5 Creating a bottom-up parser automatically 156
   3.6 Recovering grammars from legacy code 193

Grammars are an essential tool in language specification; they have several important aspects. First, a grammar serves to impose a structure on the linear sequence of tokens which is the program. This structure is all-important since the semantics of the program is specified in terms of the nodes in this structure. The process of finding the structure in the flat stream of tokens is called parsing, and a module that performs this task is a parser.

Second, using techniques from the field of formal languages, a parser can be constructed automatically from a grammar. This is a great help in compiler construction.

Third, grammars are a powerful documentation tool. They help programmers to write syntactically correct programs and provide answers to detailed questions about the syntax. They do the same for compiler writers.

There are two well-known and well-researched ways to do parsing, deterministic left-to-right top-down (the LL method) and deterministic left-to-right bottom-up
(the LR and LALR methods), and a third, emerging, technique, generalized LR. **Left-to-right** means that the program text, or more precisely the sequence of tokens, is processed from left to right, one token at the time. Intuitively speaking, **deterministic** means that no searching is involved: each token brings the parser one step closer to the goal of constructing the syntax tree, and it is never necessary to undo one of these steps. The theory of formal languages provides a more rigorous definition. The terms top-down and bottom-up will be explained below.

The deterministic parsing methods have the advantage that they require an amount of time that is a linear function of the length of the input: they are **linear-time** methods. There is also another reason to require determinacy: a grammar for which a deterministic parser can be generated is guaranteed to be non-ambiguous, which is of course a very important property of a programming language grammar. Being non-ambiguous and allowing deterministic parsing are not exactly the same (the second implies the first but not vice versa), but requiring determinacy is technically the best non-ambiguity test we have.

Unfortunately, deterministic parsers do not solve all parsing problems: they work for restricted classes of grammars only. A grammar copied “as is” from a language manual has a very small chance of leading to a deterministic method, unless of course the language designer has taken pains to make the grammar match such a method. There are several ways to deal with this problem:

- transform the grammar so that it becomes amenable to a deterministic method;
- allow the user to “add” sufficient determinism;
- use a non-deterministic method.

Methods to transform the grammar are explained in Sections 3.4.3. The transformed grammar will assign syntax trees to at least some programs that differ from the original trees. This unavoidably causes some problems in further processing, since the semantics is described in terms of the original syntax trees. So grammar transformation methods must also create transformed semantic rules. Methods to add extra-grammatical determinism are described in Section 3.4.3.3 and 3.5.7. They use so-called “conflict resolvers,” which specify decisions the parser cannot take. This can be convenient, but takes away some of the safety inherent in grammars.

Dropping the determinism—allowing searching to take place—results in algorithms that can handle practically all grammars. These algorithms are not linear-time and their time and space requirements vary. One such algorithm is “generalized LR”, which is reasonably well-behaved when applied to programming language grammars. Generalized LR is most often used in (re)compiling legacy code for which no deterministic grammar exists. Generalized LR is treated in Section 3.5.8.

We will assume that the grammar of the programming language is non-ambiguous. This implies that to each input program there belongs either one syntax tree, and then the program is syntactically correct, or no syntax tree, and then the program contains one or more syntax errors.