**4** Lexical and Syntax Analysis

1. [**4.1 Introduction**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001274.xhtml#P7001015124000000000000000001274)
2. [**4.2 Lexical Analysis**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml#P7001015124000000000000000001285)
3. [**4.3 The Parsing Problem**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000012CE.xhtml#P70010151240000000000000000012CE)
4. [**4.4 Recursive-Descent Parsing**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001303.xhtml#P7001015124000000000000000001303)
5. [**4.5 Bottom-Up Parsing**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml#P70010151240000000000000000013AC)

This chapter begins with an introduction to lexical analysis, along with a ­simple example. Next, the general parsing problem is discussed, including the two ­primary approaches to parsing, and the complexity of parsing. Then, we ­introduce the recursive-descent implementation technique for top-down parsers, including examples of parts of a recursive-descent parser and a trace of a parse using one. The last section discusses bottom-up parsing and the LR parsing algorithm. This section includes an example of a small LR parsing table and the parse of a string using the LR parsing process.

4.1 Introduction

A serious investigation of compiler design requires at least a semester of intensive study, including the design and implementation of a compiler for a small but realistic programming language. The first part of such a course is devoted to lexical and syntax analyses. The syntax analyzer is the heart of a compiler, because several other important components, including the semantic analyzer and the intermediate code generator, are driven by the actions of the syntax analyzer.

Some readers may wonder why a chapter on any part of a compiler would be included in a book on programming languages. There are at least two reasons to include a discussion of lexical and syntax analyses in this text: First, syntax analyzers are based directly on the grammars discussed in [**Chapter 3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000009DC.xhtml#P70010151240000000000000000009DC), so it is natural to discuss them as an application of grammars. Second, lexical and syntax analyzers are needed in numerous situations outside compiler design. Many applications, among them program listing formatters, programs that compute the complexity of programs, and programs that must analyze and react to the contents of a configuration file, all need to do lexical and syntax analyses. Therefore, lexical and syntax analyses are important topics for software developers, even if they never need to write a compiler. Furthermore, some computer science programs no longer require students to take a compiler design course, which leaves students with no instruction in lexical or syntax analysis. In those cases, this chapter can be covered in the programming language course. In degree programs that require a compiler design course, this chapter can be skipped.

Three different approaches to implementing programming languages are introduced in [**Chapter 1**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000000299.xhtml#P7001015124000000000000000000299): compilation, pure interpretation, and hybrid implementation. The compilation approach uses a program called a compiler, which translates programs written in a high-level programming language into machine code. Compilation is typically used to implement programming languages that are used for large applications, often written in languages such as C++ and COBOL. Pure interpretation systems perform no translation; rather, programs are interpreted in their original form by a software interpreter. Pure interpretation is usually used for smaller systems in which execution efficiency is not critical, such as scripts embedded in HTML documents, written in ­languages such as JavaScript. Hybrid implementation systems translate programs written in high-level languages into intermediate forms, which are interpreted. These systems are now more widely used than ever, thanks in large part to the popularity of scripting languages. Traditionally, hybrid systems have resulted in much slower program execution than compiler systems. However, in recent years the use of Just-in-Time (JIT) compilers has become widespread, particularly for Java programs and programs written for the Microsoft .NET system. A JIT compiler, which translates intermediate code to machine code, is used on methods at the time they are first called. In effect, a JIT compiler transforms a hybrid system to a delayed compiler system.

All three of the implementation approaches just discussed use both lexical and syntax analyzers.

Syntax analyzers, or parsers, are nearly always based on a formal description of the syntax of programs. The most commonly used syntax-description formalism is context-free grammars, or BNF, which is introduced in [**Chapter 3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000009DC.xhtml#P70010151240000000000000000009DC). Using BNF, as opposed to using some informal syntax description, has at least three compelling advantages. First, BNF descriptions of the syntax of programs are clear and concise, both for humans and for software systems that use them. Second, the BNF description can be used as the direct basis for the syntax analyzer. Third, implementations based on BNF are relatively easy to maintain because of their modularity.

Nearly all compilers separate the task of analyzing syntax into two distinct parts, lexical analysis and syntax analysis, although this terminology is confusing. The lexical analyzer deals with small-scale language constructs, such as names and numeric literals. The syntax analyzer deals with the large-scale constructs, such as expressions, statements, and program units. [**Section 4.2**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml#P7001015124000000000000000001285) introduces lexical analyzers. [**Sections 4.3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000012CE.xhtml#P70010151240000000000000000012CE), [**4.4**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001303.xhtml#P7001015124000000000000000001303), and [**4.5**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml#P70010151240000000000000000013AC) discuss syntax analyzers.

There are three reasons why lexical analysis is separated from syntax analysis:

1. Simplicity—Techniques for lexical analysis are less complex than those required for syntax analysis, so the lexical-analysis process can be simpler if it is separate. Also, removing the low-level details of lexical analysis from the syntax analyzer makes the syntax analyzer both smaller and less complex.
2. Efficiency—Although it pays to optimize the lexical analyzer, because lexical analysis requires a significant portion of total compilation time, it is not fruitful to optimize the syntax analyzer. Separation facilitates this selective optimization.
3. Portability—Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform dependent. However, the syntax analyzer can be platform independent. It is always good to isolate machine-dependent parts of any software system.

4.2 Lexical Analysis

A lexical analyzer is essentially a pattern matcher. A pattern matcher attempts to find a substring of a given string of characters that matches a given character pattern. Pattern matching is a traditional part of computing. One of the earliest uses of pattern matching was with text editors, such as the ed line editor, which was introduced in an early version of UNIX. Since then, pattern matching has found its way into some programming languages—for example, Perl and JavaScript. It is also available through the standard class libraries of Java, C++, and C#.

A lexical analyzer serves as the front end of a syntax analyzer. Technically, lexical analysis is a part of syntax analysis. A lexical analyzer performs syntax analysis at the lowest level of program structure. An input program appears to a compiler as a single string of characters. The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to their structure. In [**Chapter 3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000009DC.xhtml#P70010151240000000000000000009DC), these logical groupings are named **lexemes**, and the internal codes for categories of these groupings are named **tokens**. Lexemes are recognized by matching the input character string against character string patterns. Although tokens are usually represented as integer values, for the sake of readability of lexical and syntax analyzers, they are often referenced through named constants.

Consider the following example of an assignment statement:

result = oldsum - value / 100;

Following are the tokens and lexemes of this statement:

Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens. In the early days of compilers, lexical analyzers often processed an entire source program file and produced a file of tokens and lexemes. Now, however, most lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer. So, each call to the lexical analyzer returns a single lexeme and its token. The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time.

The lexical-analysis process includes skipping comments and white space outside lexemes, as they are not relevant to the meaning of the program. Also, the lexical analyzer inserts lexemes for user-defined names into the symbol table, which is used by later phases of the compiler. Finally, lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and report such errors to the user.

There are three approaches for building a lexical analyzer:

1. Write a formal description of the token patterns of the language using a descriptive language related to regular expressions.[**1**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml?brand=vitalsource&create=true&favre=brett#P70010151240000000000000000015CD) These descriptions are used as input to a software tool that automatically generates a lexical analyzer. There are many such tools available for this. The oldest of these, named lex, is commonly included as part of UNIX systems.

**1.** These regular expressions are the basis for the pattern-matching facilities now part of many programming languages, either directly or through a class library.

[**[Return to reference]**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml?brand=vitalsource&create=true&favre=brett#r__P70010151240000000000000000015CD)

1. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
2. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

A state transition diagram, or just **state diagram**, is a directed graph. The nodes of a state diagram are labeled with state names. The arcs are labeled with the input characters that cause the transitions among the states. An arc may also include actions the lexical analyzer must perform when the transition is taken.

State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called **finite automata**. Finite automata can be designed to recognize members of a class of languages called **regular languages**. Regular grammars are generative devices for regular languages. The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton.

We now illustrate lexical-analyzer construction with a state diagram and the code that implements it. The state diagram could simply include states and transitions for each and every token pattern. However, that approach results in a very large and complex diagram, because every node in the state diagram would need a transition for every character in the character set of the language being analyzed. We therefore consider ways to simplify it.

Suppose we need a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands. Assume that the variable names consist of strings of uppercase letters, lowercase letters, and digits but must begin with a letter. Names have no length limitation. The first thing to observe is that there are 52 different characters (any uppercase or lowercase letter) that can begin a name, which would require 52 transitions from the transition diagram’s initial state. However, a lexical analyzer is interested only in determining that it is a name and is not concerned with which specific name it happens to be. Therefore, we define a character class named LETTER for all 52 letters and use a single transition on the first letter of any name.

Another opportunity for simplifying the transition diagram is with the integer literal tokens. There are 10 different characters that could begin an integer literal lexeme. This would require 10 transitions from the start state of the state diagram. Because specific digits are not a concern of the lexical analyzer, we can build a much more compact state diagram if we define a character class named DIGIT for digits and use a single transition on any character in this character class to a state that collects integer literals.

Because our names can include digits, the transition from the node following the first character of a name can use a single transition on LETTER or DIGIT to continue collecting the characters of a name.

Next, we define some utility subprograms for the common tasks inside the lexical analyzer. First, we need a subprogram, which we can name getChar, that has several duties. When called, getChar gets the next character of input from the input program and puts it in the global variable nextChar. getChar also must determine the character class of the input character and put it in the global variable charClass. The lexeme being built by the lexical analyzer, which could be implemented as a character string or an array, will be named lexeme.

We implement the process of putting the character in nextChar into the string array lexeme in a subprogram named addChar. This subprogram must be explicitly called because programs include some characters that need not be put in lexeme, for example the white-space characters between lexemes. In a more realistic lexical analyzer, comments also would not be placed in lexeme.

When the lexical analyzer is called, it is convenient if the next character of input is the first character of the next lexeme. Because of this, a function named getNonBlank is used to skip white space every time the analyzer is called.

Finally, a subprogram named lookup is needed to compute the token code for the single-character tokens. In our example, these are parentheses and the arithmetic operators. Token codes are numbers arbitrarily assigned to tokens by the compiler writer.

The state diagram in [**Figure 4.1**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml?brand=vitalsource&create=true&favre=brett#P70010151240000000000000000012B6) describes the patterns for our tokens. It includes the actions required on each transition of the state diagram.

**Figure 4.1 A state diagram to recognize names, ­parentheses, and ­arithmetic operators**

[**Figure 4.1 Full Alternative Text**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/longalt/la_fig04-01.xhtml#la_fig04-01)

The following is a C implementation of a lexical analyzer specified in the state diagram of [**Figure 4.1**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml?brand=vitalsource&create=true&favre=brett#P70010151240000000000000000012B6), including a main driver function for testing purposes:

/\* front.c - a lexical analyzer system for simple

arithmetic expressions \*/

#include <stdio.h>

#include <ctype.h>

/\* Global declarations \*/

/\* Variables \*/

**int** charClass;

**char** lexeme [100];

**char** nextChar;

**int** lexLen;

**int** token;

**int** nextToken;

FILE \*in\_fp, \*fopen();

/\* Function declarations \*/

**void** addChar();

**void** getChar();

**void** getNonBlank();

**int** lex();

/\* Character classes \*/

#define LETTER 0

#define DIGIT 1

#define UNKNOWN 99

/\* Token codes \*/

#define INT\_LIT 10

#define IDENT 11

#define ASSIGN\_OP 20

#define ADD\_OP 21

#define SUB\_OP 22

#define MULT\_OP 23

#define DIV\_OP 24

#define LEFT\_PAREN 25

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* main driver \*/

main() {

/\* Open the input data file and process its contents \*/

**if** ((in\_fp = fopen("front.in", "r")) == NULL)

printf("ERROR - cannot open front.in \n");

**else** {

getChar();

**do** {

lex();

} **while** (nextToken! = EOF);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* lookup - a function to lookup operators and parentheses

and return the token \*/

**int** lookup(**char** ch) {

**switch** (ch) {

**case** '(':

addChar();

nextToken = LEFT\_PAREN;

**break**;

**case** ')':

addChar();

nextToken = RIGHT\_PAREN;

**break**;

**case** '+':

addChar();

nextToken = ADD\_OP;

**break**;

**case** '-':

addChar();

nextToken = SUB\_OP;

**break**;

**case** '\*':

addChar();

nextToken = MULT\_OP;

**break**;

**case** '/':

addChar();

nextToken = DIV\_OP;

**break**;

**default**:

addChar();

nextToken = EOF;

**break**;

}

**return** nextToken;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* addChar - a function to add nextChar to lexeme \*/

**void** addChar() {

**if** (lexLen <= 98) {

lexeme[lexLen++] = nextChar;

lexeme[lexLen] = 0;

}

**else**

printf("Error - lexeme is too long \n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* getChar - a function to get the next character of

input and determine its character class \*/

**void** getChar() {

**if** ((nextChar = getc(in\_fp)) = EOF) {

**if** (isalpha(nextChar))

charClass = LETTER;

**else if** (isdigit(nextChar))

charClass = DIGIT;

**else** charClass = UNKNOWN;

}

**else**

charClass = EOF;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* getNonBlank - a function to call getChar until it

returns a non-whitespace character \*/

**void** getNonBlank() {

**while** (isspace(nextChar))

getChar();

}

/

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* lex - a simple lexical analyzer for arithmetic

expressions \*/

**int** lex() {

lexLen = 0;

getNonBlank();

**switch** (charClass) {

/\* Parse identifiers \*/

**case** LETTER:

addChar();

getChar();

**while** (charClass == LETTER || charClass == DIGIT) {

addChar();

getChar();

}

nextToken = IDENT;

**break**;

/\* Parse integer literals \*/

**case** DIGIT:

addChar();

getChar();

**while** (charClass == DIGIT) {

addChar();

getChar();

}

nextToken = INT\_LIT;

**break**;

/\* Parentheses and operators \*/

**case** UNKNOWN:

lookup(nextChar);

getChar();

**break**;

/\* EOF \*/

**case** EOF:

nextToken = EOF;

lexeme[0] = 'E';

lexeme[1] = 'O';

lexeme[2] = 'F';

lexeme[3] = 0;

**break**;

} /\* End of switch \*/

printf("Next token is: %d, Next lexeme is %s\n",

nextToken, lexeme);

**return** nextToken;

} /\* End of function lex \*/

This code illustrates the relative simplicity of lexical analyzers. Of course, we have left out input buffering, as well as some other important details. ­Furthermore, we have dealt with a very small and simple input language.

Consider the following expression:

(sum + 47) / total

Following is the output of the lexical analyzer of front.c when used on this expression:

Next token is: 25 Next lexeme is (

Next token is: 11 Next lexeme is sum

Next token is: 21 Next lexeme is +

Next token is: 10 Next lexeme is 47

Next token is: 26 Next lexeme is )

Next token is: 24 Next lexeme is /

Next token is: 11 Next lexeme is total

Next token is: -1 Next lexeme is EOF

Names and reserved words in programs have similar patterns. Although it is possible to build a state diagram to recognize every specific reserved word of a programming language, that would result in a prohibitively large state diagram. It is much simpler and faster to have the lexical analyzer recognize names and reserved words with the same pattern and use a lookup in a table of reserved words to determine which names are reserved words. Using this approach considers reserved words to be exceptions in the names token category.

A lexical analyzer often is responsible for the initial construction of the symbol table, which acts as a database of names for the compiler. The entries in the symbol table store information about user-defined names, as well as the attributes of the names. For example, if the name is that of a variable, the variable’s type is one of its attributes that will be stored in the symbol table. Names are usually placed in the symbol table by the lexical analyzer. The attributes of a name are usually put in the symbol table by some part of the compiler that is subsequent to the actions of the lexical analyzer.

4.3 The Parsing Problem

The part of the process of analyzing syntax that is referred to as *syntax analysis* is often called **parsing**. We will use these two interchangeably.

This section discusses the general parsing problem and introduces the two main categories of parsing algorithms, top-down and bottom-up, as well as the complexity of the parsing process.

4.3.1 Introduction to Parsing

Parsers for programming languages construct parse trees for given programs. In some cases, the parse tree is only implicitly constructed, meaning that perhaps only a traversal of the tree is generated. But in all cases, the information required to build the parse tree is created during the parse. Both parse trees and derivations include all of the syntactic information needed by a language processor.

There are two distinct goals of syntax analysis: First, the syntax analyzer must check the input program to determine whether it is syntactically correct. When an error is found, the analyzer must produce a diagnostic message and recover. In this case, recovery means it must get back to a normal state and continue its analysis of the input program. This step is required so that the compiler finds as many errors as possible during a single analysis of the input program. If it is not done well, error recovery may create more errors, or at least more error messages. The second goal of syntax analysis is to produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input. The parse tree (or its trace) is used as the basis for translation.

Parsers are categorized according to the direction in which they build parse trees. The two broad classes of parsers are **top-down**, in which the tree is built from the root downward to the leaves, and **bottom-up**, in which the parse tree is built from the leaves upward to the root.

In this chapter, we use a small set of notational conventions for grammar symbols and strings to make the discussion less cluttered. For formal languages, they are as follows:

1. Terminal symbols—lowercase letters at the beginning of the alphabet (a, b, . . .)
2. Nonterminal symbols—uppercase letters at the beginning of the alphabet (A, B, . . .)
3. Terminals or nonterminals—uppercase letters at the end of the alphabet (W, X, Y, Z)
4. Strings of terminals—lowercase letters at the end of the alphabet (w, x, y, z)
5. Mixed strings (terminals and/or nonterminals)—lowercase Greek letters

For programming languages, terminal symbols are the small-scale syntactic constructs of the language, what we have referred to as lexemes. The nonterminal symbols of programming languages are usually connotative names or abbreviations, surrounded by angle brackets—for example, <while\_­statement>, <expr>, and <function\_def>. The sentences of a language (­programs, in the case of a programming language) are strings of terminals. Mixed strings describe right-hand sides (RHSs) of grammar rules and are used in parsing algorithms.

4.3.2 Top-Down Parsers

A top-down parser traces or builds a parse tree in preorder. A preorder traversal of a parse tree begins with the root. Each node is visited before its branches are followed. Branches from a particular node are followed in left-to-right order. This corresponds to a leftmost derivation.

In terms of the derivation, a top-down parser can be described as follows: Given a sentential form that is part of a leftmost derivation, the parser’s task is to find the next sentential form in that leftmost derivation. The general form of a left sentential form is  whereby our notational conventions x is a string of terminal symbols, A is a nonterminal, and  is a mixed string. Because x contains only terminals, A is the leftmost nonterminal in the sentential form, so it is the one that must be expanded to get the next sentential form in a leftmost derivation. Determining the next sentential form is a matter of choosing the correct grammar rule that has A as its LHS. For example, if the current sentential form is  and the A-rules are  and  a top-down parser must choose among these three rules to get the next sentential form, which could be  or  This is the parsing decision problem for top-down parsers.

Different top-down parsing algorithms use different information to make parsing decisions. The most common top-down parsers choose the correct RHS for the leftmost nonterminal in the current sentential form by comparing the next token of input with the first symbols that can be generated by the RHSs of those rules. Whichever RHS has that token at the left end of the string it generates is the correct one. So, in the sentential form  the parser would use whatever token would be the first generated by A to determine which A-rule should be used to get the next sentential form. In the example above, the three RHSs of the A-rules all begin with different terminal symbols. The parser can easily choose the correct RHS based on the next token of input, which must be a, b, or c in this example. In general, choosing the correct RHS is not so straightforward, because some of the RHSs of the leftmost nonterminal in the current sentential form may begin with a nonterminal.

The most common top-down parsing algorithms are closely related. A **recursive-descent parser** is a coded version of a syntax analyzer based directly on the BNF description of the syntax of language. The most common alternative to recursive descent is to use a parsing table, rather than code, to implement the BNF rules. Both of these, which are called **LL ­algorithms**, are equally powerful, meaning they work on the same subset of all context-free grammars. The first L in LL specifies a left-to-right scan of the input; the second L specifies that a leftmost derivation is generated. [**Section 4.4**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001303.xhtml#P7001015124000000000000000001303) introduces the recursive-descent approach to implementing an LL parser.

4.3.3 Bottom-Up Parsers

A bottom-up parser constructs a parse tree by beginning at the leaves and progressing toward the root. This parse order corresponds to the reverse of a rightmost derivation. That is, the sentential forms of the derivation are produced in order of last to first. In terms of the derivation, a bottom-up parser can be described as follows: Given a right sentential form  the parser must determine what substring of  is the RHS of the rule in the grammar that must be reduced to its LHS to produce the previous sentential form in the rightmost derivation. For example, the first step for a bottom-up parser is to determine which substring of the initial given sentence is the RHS to be reduced to its corresponding LHS to get the second last sentential form in the derivation. The process of finding the correct RHS to reduce is complicated by the fact that a given right sentential form may include more than one RHS from the grammar of the language being parsed. The correct RHS is called the ­**handle**. A right sentential form is a sentential form that appears in a rightmost derivation.

Consider the following grammar and derivation:

* S => aAc => aaAc => aabc

A bottom-up parser of this sentence, aabc, starts with the sentence and must find the handle in it. In this example, this is an easy task, for the string contains only one RHS, b. When the parser replaces b with its LHS, A, it gets the second to last sentential form in the derivation, aaAc. In the general case, as stated previously, finding the handle is much more difficult, because a sentential form may include several different RHSs.

A bottom-up parser finds the handle of a given right sentential form by examining the symbols on one or both sides of a possible handle. Symbols to the right of the possible handle are usually tokens in the input that have not yet been analyzed.

The most common bottom-up parsing algorithms are in the LR family, where the L specifies a left-to-right scan of the input and the R specifies that a rightmost derivation is generated.

4.3.4 The Complexity of Parsing

Parsing algorithms that work for any unambiguous grammar are complicated and inefficient. In fact, the complexity of such algorithms is  which means the amount of time they take is on the order of the cube of the length of the string to be parsed. This relatively large amount of time is required because these algorithms frequently must back up and reparse part of the sentence being analyzed. Reparsing is required when the parser has made a mistake in the parsing process. Backing up the parser also requires that part of the parse tree being constructed (or its trace) must be dismantled and rebuilt.  algorithms are normally not useful for practical processes, such as syntax analysis for a compiler, because they are far too slow. In situations such as this, computer scientists often search for algorithms that are faster, though less general. Generality is traded for efficiency. In terms of parsing, faster algorithms have been found that work for only a subset of the set of all possible grammars. These algorithms are acceptable as long as the subset includes grammars that describe programming languages. (Actually, as discussed in [**Chapter 3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000009DC.xhtml#P70010151240000000000000000009DC), the whole class of context-free grammars is not adequate to describe all of the syntax of most programming languages.)

All algorithms used for the syntax analyzers of commercial compilers have complexity O(n), which means the time they take is linearly related to the length of the string to be parsed. This is vastly more efficient than  algorithms.

4.4 Recursive-Descent Parsing

This section introduces the recursive-descent top-down parser implementation process.

4.4.1 The Recursive-Descent Parsing Process

A recursive-descent parser is so named because it consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order. This recursion is a reflection of the nature of programming languages, which include several different kinds of nested structures. For example, statements are often nested in other statements. Also, parentheses in expressions must be properly nested. The syntax of these structures is naturally described with recursive grammar rules.

EBNF is ideally suited for recursive-descent parsers. Recall from ­[**Chapter 3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000009DC.xhtml#P70010151240000000000000000009DC) that the primary EBNF extensions are braces, which specify that what they enclose can appear zero or more times, and brackets, which specify that what they enclose can appear once or not at all. Note that in both cases, the enclosed symbols are optional. Consider the following examples:

* <if\_statement>  **if** <logic\_expr> <statement> [**else** <statement>]
* <ident\_list>  ident {, ident}

In the first rule, the **else** clause of an **if** statement is optional. In the second, an <ident\_list> is an identifier, followed by zero or more repetitions of a comma and an identifier.

A recursive-descent parser has a subprogram for each nonterminal in its associated grammar. The responsibility of the subprogram associated with a particular nonterminal is as follows: When given an input string, it traces out the parse tree that can be rooted at that nonterminal and whose leaves match the input string. In effect, a recursive-descent parsing subprogram is a parser for the language (set of strings) that is generated by its associated nonterminal.

Consider the following EBNF description of simple arithmetic expressions:

* <expr>  <term> {(+ | -) <term>}
* <term>  <factor> {(\* | /) <factor>}
* <factor>  id | int\_constant | ( <expr> )

Recall from [**Chapter 3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000009DC.xhtml#P70010151240000000000000000009DC) that an EBNF grammar for arithmetic expressions, such as this one, does not force any associativity rule. Therefore, when using such a grammar as the basis for a compiler, one must take care to ensure that the code generation process, which is normally driven by syntax analysis, produces code that adheres to the associativity rules of the language. This can be done easily when recursive-descent parsing is used.

In the following recursive-descent function, expr, the lexical analyzer is the function that is implemented in [**Section 4.2**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml#P7001015124000000000000000001285). It gets the next lexeme and puts its token code in the global variable nextToken. The token codes are defined as named constants, as in [**Section 4.2**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml#P7001015124000000000000000001285).

A recursive-descent subprogram for a rule with a single RHS is relatively simple. For each terminal symbol in the RHS, that terminal symbol is compared with nextToken. If they do not match, it is a syntax error. If they match, the lexical analyzer is called to get the next input token. For each nonterminal, the parsing subprogram for that nonterminal is called.

The recursive-descent subprogram for the first rule in the previous example grammar, written in C, is

/\* expr

Parses strings in the language generated by the rule:

<expr> -> <term> {(+ | -) <term>}

\*/

**void** expr() {

printf("Enter <expr>\n");

/\* Parse the first term \*/

term();

/\* As long as the next token is + or -, get

the next token and parse the next term \*/

**while** (nextToken == ADD\_OP || nextToken == SUB\_OP) {

lex();

term();

}

printf("Exit <expr>\n");

} /\* End of function expr \*/

Notice that the expr function includes tracing output statements, which are included to produce the example output shown later in this section.

Recursive-descent parsing subprograms are written with the convention that each one leaves the next token of input in nextToken. So, whenever a parsing function begins, it assumes that nextToken has the code for the leftmost token of the input that has not yet been used in the parsing process.

The part of the language that the expr function parses consists of one or more terms, separated by either plus or minus operators. This is the language generated by the nonterminal <expr>. Therefore, first it calls the function that parses terms (term). Then it continues to call that function as long as it finds ADD\_OP or SUB\_OP tokens (which it passes over by calling lex). This recursive-descent function is simpler than most, because its associated rule has only one RHS. Furthermore, it does not include any code for syntax error detection or recovery, because there are no detectable errors associated with the grammar rule.

A recursive-descent parsing subprogram for a nonterminal whose rule has more than one RHS begins with code to determine which RHS is to be parsed. Each RHS is examined (at compiler construction time) to determine the set of terminal symbols that can appear at the beginning of sentences it can generate. By matching these sets against the next token of input, the parser can choose the correct RHS.

The parsing subprogram for <term> is similar to that for <expr>:

/\* term

Parses strings in the language generated by the rule:

<term> -> <factor> {(\* | /) <factor>)

\*/

**void** term() {

printf("Enter <term>\n");

/\* Parse the first factor \*/

factor();

/\* As long as the next token is \* or /, get the

next token and parse the next factor \*/

**while** (nextToken == MULT\_OP || nextToken == DIV\_OP) {

lex();

factor();

}

printf("Exit <term>\n");

} /\* End of function term \*/

The function for the <factor> nonterminal of our arithmetic expression grammar must choose between its two RHSs. It also includes error detection. In the function for <factor>, the reaction to detecting a syntax error is simply to call the error function. In a real parser, a diagnostic message must be produced when an error is detected. Furthermore, parsers must recover from the error so that the parsing process can continue.

/\* factor

Parses strings in the language generated by the rule:

<factor> -> id | int\_constant | ( <expr )

\*/

**void** factor() {

printf("Enter <factor>\n");

/\* Determine which RHS \*/

**if** (nextToken == IDENT || nextToken == INT\_LIT)

/\* Get the next token \*/

lex();

/\* If the RHS is ( <expr> ), call lex to pass over the

left parenthesis, call expr, and check for the right

parenthesis \*/

**else** {

**if** (nextToken == LEFT\_PAREN) {

lex();

expr();

**if** (nextToken == RIGHT\_PAREN)

lex();

**else**

error();

} /\* End of if (nextToken == ... \*/

/\* It was not an id, an integer literal, or a left

parenthesis \*/

**else** error();

} /\* End of else \*/

printf("Exit <factor>\n");;

} /\* End of function factor \*/

Following is the trace of the parse of the example expression (sum + 47) / total, using the parsing functions expr, term, and factor, and the function lex from [**Section 4.2**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001285.xhtml#P7001015124000000000000000001285). Note that the parse begins by calling lex and the start symbol routine, in this case, expr.

Next token is: 25 Next lexeme is (

Enter <expr>

Enter <term>

Enter <factor>

Next token is: 11 Next lexeme is sum

Enter <expr>

Enter <term>

Enter <factor>

Next token is: 21 Next lexeme is +

Exit <factor>

Exit <term>

Next token is: 10 Next lexeme is 47

Enter <term>

Enter <factor>

Next token is: 26 Next lexeme is )

Exit <factor>

Exit <term>

Exit <expr>

Next token is: 24 Next lexeme is /

Exit <factor>

Next token is: 11 Next lexeme is total

Enter <factor>

Next token is: -1 Next lexeme is EOF

Exit <factor>

Exit <term>

Exit <expr>

The parse tree traced by the parser for the preceding expression is shown in [**Figure 4.2**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001303.xhtml?brand=vitalsource&create=true&favre=brett#P700101512400000000000000000134E).

**Figure 4.2 Parse tree for (sum  47)/ total**

[**Figure 4.2 Full Alternative Text**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/longalt/la_fig04-02.xhtml#la_fig04-02)

One more example grammar rule and parsing function should help solidify the reader’s understanding of recursive-descent parsing. Following is a grammatical description of the Java **if** statement:

<ifstmt>  **if** (<boolexpr>) <statement> [**else** <statement>]

The recursive-descent subprogram for this rule follows:

/\* Function ifstmt

Parses strings in the language generated by the rule:

<ifstmt> -> if (<boolexpr>) <statement>

[else <statement>]

\*/

**void** ifstmt() {

/\* Be sure the first token is 'if' \*/

**if** (nextToken = IF\_CODE)

error();

**else** {

/\* Call lex to get to the next token \*/

lex();

/\* Check for the left parenthesis \*/

**if** (nextToken = LEFT\_PAREN)

error();

**else** {

/\* Parse the Boolean expression \*/

boolexpr();

/\* Check for the right parenthesis \*/

**if** (nextToken = RIGHT\_PAREN)

error();

**else** {

/\* Parse the then clause \*/

statement();

/\* If an else is next, parse the else clause \*/

**if** (nextToken == ELSE\_CODE) {

/\* Call lex to get over the else \*/

lex();

statement();

} /\* end of if (nextToken == ELSE\_CODE ... \*/

} /\* end of else of if (nextToken != RIGHT ... \*/

} /\* end of else of if (nextToken != LEFT ... \*/

} /\* end of else of if (nextToken != IF\_CODE ... \*/

} /\* end of ifstmt \*/

Notice that this function uses parser functions for statements and Boolean expressions that are not given in this section.

The objective of these examples is to convince you that a recursive-descent parser can be easily written if an appropriate grammar is available for the language. The characteristics of a grammar that allows a recursive-descent parser to be built are discussed in the following subsection.

4.4.2 The LL Grammar Class

Before choosing to use recursive descent as a parsing strategy for a compiler or other program analysis tool, one must consider the limitations of the approach, in terms of grammar restrictions. This section discusses these restrictions and their possible solutions.

One simple grammar characteristic that causes a catastrophic problem for LL parsers is left recursion. For example, consider the following rule:

A recursive-descent parser subprogram for A immediately calls itself to parse the first symbol in its RHS. That activation of the A parser subprogram then immediately calls itself again, and again, and so forth. It is easy to see that this leads nowhere (except to stack overflow).

The left recursion in the rule  is called **direct left recursion**, because it occurs in one rule. Direct left recursion can be eliminated from a grammar by the following process:

For each nonterminal, A,

1. Group the A-rules as  where none of the  begins with A
2. Replace the original A-rules with

Note that  specifies the empty string. A rule that has  as its RHS is called an *erasure rule*, because its use in a derivation effectively erases its LHS from the sentential form.

Consider the following example grammar and the application of the above process:

For the E-rules, we have  = + T and  = T, so we replace the E-rules with

For the T-rules, we have  = \*F and  = F, so we replace the T-rules with

Because there is no left recursion in the F-rules, they remain the same, so the complete replacement grammar is

This grammar generates the same language as the original grammar but is not left recursive.

As was the case with the expression grammar written using EBNF in [**Section 4.4.1**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001303.xhtml?brand=vitalsource&create=true&favre=brett#P7001015124000000000000000001306), this grammar does not specify left associativity of operators. However, it is relatively easy to design the code generation based on this grammar so that the addition and multiplication operators will have left associativity.

Indirect left recursion poses the same problem as direct left recursion. For example, suppose we have

A recursive-descent parser for these rules would have the A subprogram immediately call the subprogram for B, which immediately calls the A subprogram. So, the problem is the same as for direct left recursion. The problem of left recursion is not confined to the recursive-descent approach to building top-down parsers. It is a problem for all top-down parsing algorithms. Fortunately, left recursion is not a problem for bottom-up parsing algorithms.

There is an algorithm to modify a given grammar to remove indirect left recursion ([**Aho et al., 2006**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004D68)), but it is not covered here. When writing a ­grammar for a programming language, one can usually avoid including left recursion, both direct and indirect.

Left recursion is not the only grammar trait that disallows top-down parsing. Another is whether the parser can always choose the correct RHS on the basis of the next token of input, using only the first token generated by the leftmost nonterminal in the current sentential form. There is a relatively simple test of a non-left recursive grammar that indicates whether this can be done, called the **pairwise disjointness test**. This test requires the ability to compute a set based on the RHSs of a given nonterminal symbol in a grammar. These sets, which are called FIRST, are defined as

* FIRST  =  =>\*  (IF  =>\*  is in FIRST )

in which =>\* means 0 or more derivation steps.

An algorithm to compute FIRST for any mixed string  can be found in [**Aho et al. (2006)**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004D68). For our purposes, FIRST can usually be computed by inspection of the grammar.

The pairwise disjointness test is as follows:

For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules,  and  it must be true that

ϕ

(The intersection of the two sets,  and  must be empty.)

In other words, if a nonterminal A has more than one RHS, the first terminal symbol that can be generated in a derivation for each of them must be unique to that RHS. Consider the following rules:

The FIRST sets for the RHSs of the A-rules are {a}, {b}, and {c}, {d}, which are clearly disjoint. Therefore, these rules pass the pairwise disjointness test. What this means, in terms of a recursive-descent parser, is that the code of the subprogram for parsing the nonterminal A can choose which RHS it is dealing with by seeing only the first terminal symbol of input (token) that is generated by the nonterminal. Now consider the rules

The FIRST sets for the RHSs in the A-rules are {a} and {a}, {b} which are clearly not disjoint. So, these rules fail the pairwise disjointness test. In terms of the parser, the subprogram for A could not determine which RHS was being parsed by looking at the next symbol of input, because if it were an a, it could be either RHS. This issue is of course more complex if one or more of the RHSs begin with nonterminals.

In many cases, a grammar that fails the pairwise disjointness test can be modified so that it will pass the test. For example, consider the rule

<variable>  identifier  identifier [<expression>]

This states that a <variable> is either an identifier or an identifier followed by an expression in brackets (a subscript). These rules clearly do not pass the pairwise disjointness test, because both RHSs begin with the same terminal, identifier. This problem can be alleviated through a process called **left factoring**.

We now take an informal look at left factoring. Consider our rules for <variable>. Both RHSs begin with identifier. The parts that follow identifier in the two RHSs are  (the empty string) and [<expression>]. The two rules can be replaced by the following two rules:

<variable>  identifier <new>

<new>  [<expression>]

It is not difficult to see that together, these two rules generate the same language as the two rules with which we began. However, these two pass the pairwise disjointness test.

If the grammar is being used as the basis for a recursive-descent parser, an alternative to left factoring is available. With an EBNF extension, the problem disappears in a way that is very similar to the left factoring solution. Consider the original rules above for <variable>. The subscript can be made optional by placing it in square brackets, as in

<variable>  identifier [[<expression>]]

In this rule, the outer brackets are metasymbols that indicate that what is inside is optional. The inner brackets are terminal symbols of the programming language being described. The point is that we replaced two rules with a single rule that generates the same language but passes the pairwise disjointness test.

A formal algorithm for left factoring can be found in [**Aho et al. (2006)**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004D68). Left factoring cannot solve all pairwise disjointness problems of grammars. In some cases, rules must be rewritten in other ways to eliminate the problem.

4.5 Bottom-Up Parsing

This section introduces the general process of bottom-up parsing and includes a description of the LR parsing algorithm.

4.5.1 The Parsing Problem for Bottom-Up Parsers

Consider the following grammar for arithmetic expressions:

* +
* \*
* (E)

Notice that this grammar generates the same arithmetic expressions as the example in [**Section 4.4**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000001303.xhtml#P7001015124000000000000000001303). The difference is that this grammar is left recursive, which is acceptable to bottom-up parsers. Also note that grammars for bottom-up parsers normally do not include metasymbols such as those used to specify extensions to BNF. The following rightmost derivation illustrates this grammar:

* E =>
* => E + T \* F
* => E + T \* id
* => E + F \* id
* => E + id \* id
* => T + id \* id
* => F + id \* id
* => id + id \* id

The underlined part of each sentential form in this derivation is the RHS that is rewritten as its corresponding LHS to get the previous sentential form. The process of bottom-up parsing produces the reverse of a rightmost derivation. So, in the example derivation, a bottom-up parser starts with the last sentential form (the input sentence) and produces the sequence of sentential forms from there until all that remains is the start symbol, which in this grammar is E. In each step, the task of the bottom-up parser is to find the specific RHS, the handle, in the sentential form that must be rewritten to get the next (previous) sentential form. As mentioned earlier, a right sentential form may include more than one RHS. For example, the right sentential form

* E + T \* id

includes three RHSs,  T and id. Only one of these is the handle. For example, if the RHS  were chosen to be rewritten in this sentential form, the resulting sentential form would be E \* id, but E \* id is not a legal right sentential form for the given grammar.

The handle of a right sentential form is unique. The task of a bottom-up parser is to find the handle of any given right sentential form that can be generated by its associated grammar. Formally, handle is defined as follows:

* Definition:  is the **handle** of the right sentential form  if and only if

In this definition,  specifies a rightmost derivation step, and  specifies zero or more rightmost derivation steps. Although the definition of a handle is mathematically concise, it provides little help in finding the handle of a given right sentential form. In the following, we provide the definitions of several substrings of sentential forms that are related to handles. The purpose of these is to provide some intuition about handles.

* Definition:  is a **phrase** of the right sentential form  if and only if

In this definition,  means one or more derivation steps.

* Definition:  is a **simple phrase** of the right sentential form  if and only if

If these two definitions are compared carefully, it is clear that they differ only in the last derivation specification. The definition of phrase uses one or more steps, while the definition of simple phrase uses exactly one step.

The definitions of phrase and simple phrase may appear to have the same lack of practical value as that of a handle, but that is not true. Consider what a phrase is relative to a parse tree. It is the string of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree. A simple phrase is just a phrase that takes a single derivation step from its root nonterminal node. In terms of a parse tree, a phrase can be derived from a single nonterminal in one or more tree levels, but a simple phrase can be derived in just a single tree level. Consider the parse tree shown in [**Figure 4.3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml?brand=vitalsource&create=true&favre=brett#P70010151240000000000000000013FB).

**Figure 4.3 A parse tree for id**

The leaves of the parse tree in [**Figure 4.3**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml?brand=vitalsource&create=true&favre=brett#P70010151240000000000000000013FB) comprise the sentential form  Because there are three internal nodes, there are three phrases. Each internal node is the root of a subtree, whose leaves are a phrase. The root node of the whole parse tree, E, generates all of the resulting sentential form,  which is a phrase. The internal node, T, generates the leaves T \* id, which is another phrase. Finally, the internal node, F, generates id, which is also a phrase. So, the phrases of the sentential form  are  and id. Notice that phrases are not necessarily RHSs in the underlying grammar.

The simple phrases are a subset of the phrases. In the previous example, the only simple phrase is id. A simple phrase is always a RHS in the grammar.

The reason for discussing phrases and simple phrases is this: The handle of any rightmost sentential form is its leftmost simple phrase. So now we have a highly intuitive way to find the handle of any right sentential form, assuming we have the grammar and can draw a parse tree. This approach to finding handles is of course not practical for a parser. (If you already have a parse tree, why do you need a parser?) Its only purpose is to provide the reader with some intuitive feel for what a handle is, relative to a parse tree, which is easier than trying to think about handles in terms of sentential forms.

We can now consider bottom-up parsing in terms of parse trees, although the purpose of a parser is to produce a parse tree. Given the parse tree for an entire sentence, you easily can find the handle, which is the first thing to rewrite in the sentence to get the previous sentential form. Then the handle can be pruned from the parse tree and the process repeated. Continuing to the root of the parse tree, the entire rightmost derivation can be constructed.

4.5.2 Shift-Reduce Algorithms

Bottom-up parsers are often called **shift-reduce algorithms**, because shift and reduce are the two most common actions they specify. An integral part of every bottom-up parser is a stack. As with other parsers, the input to a bottom-up parser is the stream of tokens of a program and the output is a sequence of grammar rules. The shift action moves the next input token onto the parser’s stack. A reduce action replaces an RHS (the handle) on top of the parser’s stack by its corresponding LHS. Every parser for a programming language is a **pushdown automaton** (**PDA**), because a PDA is a recognizer for a context-free language. You need not be intimate with PDAs to understand how a bottom-up parser works, although it helps. A PDA is a very simple mathematical machine that scans strings of symbols from left to right. A PDA is so named because it uses a pushdown stack as its memory. PDAs can be used as recognizers for context-free languages. Given a string of symbols over the alphabet of a context-free language, a PDA that is designed for the purpose can determine whether the string is or is not a sentence in the language. In the process, the PDA can produce the information needed to construct a parse tree for the sentence.

With a PDA, the input string is examined, one symbol at a time, left to right. The input is treated very much as if it were stored in another stack, because the PDA never sees more than the leftmost symbol of the input.

Note that a recursive-descent parser is also a PDA. In that case, the stack is that of the run-time system, which records subprogram calls (among other things), which correspond to the nonterminals of the grammar.

4.5.3 LR Parsers

Many different bottom-up parsing algorithms have been devised. Most of them are variations of a process called LR. LR parsers use a relatively small program and a parsing table that is built for a specific programming language. The original LR algorithm was designed by Donald Knuth ([**Knuth, 1965**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004E63)). This algorithm, which is sometimes called **canonical LR**, was not used in the years immediately following its publication because producing the required parsing table required large amounts of computer time and memory. Subsequently, several variations on the canonical LR table construction process were developed ([**DeRemer, 1971**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004DD8); [**DeRemer and Pennello, 1982**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004DDA)). These are characterized by two properties: (1) They require far less computer resources to produce the required parsing table than the canonical LR algorithm, and (2) they work on smaller classes of grammars than the canonical LR algorithm.

There are three advantages to LR parsers:

1. They can be built for all programming languages.
2. They can detect syntax errors as soon as it is possible in a left-to-right scan.
3. The LR class of grammars is a proper superset of the class parsable by LL parsers (for example, many left recursive grammars are LR, but none are LL).

The only disadvantage of LR parsing is that it is difficult to produce by hand the parsing table for a given grammar for a complete programming language. This is not a serious disadvantage, however, for there are several programs available that take a grammar as input and produce the parsing table, as ­discussed later in this section.

Prior to the appearance of the LR parsing algorithm, there were a number of parsing algorithms that found handles of right sentential forms by looking both to the left and to the right of the substring of the sentential form that was suspected of being the handle. Knuth’s insight was that one could effectively look to the left of the suspected handle all the way to the bottom of the parse stack to determine whether it was the handle. But all of the information in the parse stack that was relevant to the parsing process could be represented by a single state, which could be stored on the top of the stack. In other words, Knuth discovered that regardless of the length of the input string, the length of the sentential form, or the depth of the parse stack, there were only a relatively small number of different situations, as far as the parsing process is concerned. Each situation could be represented by a state and stored in the parse stack, one state symbol for each grammar symbol on the stack. At the top of the stack would always be a state symbol, which represented the relevant information from the entire history of the parse, up to the current time. We will use subscripted uppercase S’s to represent the parser states.

[**Figure 4.4**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml?brand=vitalsource&create=true&favre=brett#P7001015124000000000000000001418) shows the structure of an LR parser. The contents of the parse stack for an LR parser have the following form:

**Figure 4.4 The structure of an LR parser**

[**Figure 4.4 Full Alternative Text**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/longalt/la_fig04-04.xhtml#la_fig04-04)

where the S’s are state symbols and the X’s are grammar symbols. An LR parser configuration is a pair of strings (stack, input), with the detailed form

Notice that the input string has a dollar sign at its right end. This sign is put there during initialization of the parser. It is used for normal termination of the parser. Using this parser configuration, we can formally define the LR parser process, which is based on the parsing table.

An LR parsing table has two parts, named ACTION and GOTO. The ACTION part of the table specifies most of what the parser does. It has state symbols as its row labels and the terminal symbols of the grammar as its column labels. Given a current parser state, which is represented by the state symbol on top of the parse stack, and the next symbol (token) of input, the parse table specifies what the parser should do. The two primary parser actions are shift and reduce. Either the parser shifts the next input symbol onto the parse stack, along with a state symbol, or it already has the handle on top of the stack, which it reduces to the LHS of the rule whose RHS is the same as the handle. Two other actions are possible: accept, which means the parser has successfully completed the parse of the input, and error, which means the parser has detected a syntax error.

The rows of the GOTO part of the LR parsing table have state symbols as labels. This part of the table has nonterminals as column labels. The values in the GOTO part of the table indicate which state symbol should be pushed onto the parse stack after a reduction has been completed, which means the handle has been removed from the parse stack and the new nonterminal has been pushed onto the parse stack. The specific symbol is found at the row whose label is the state symbol on top of the parse stack after the handle and its associated state symbols have been removed. The column of the GOTO table that is used is the one with the label, that is the LHS of the rule used in the reduction.

Consider the traditional grammar for arithmetic expressions that follows:

1. + T
2. \* F
3. (E)

The rules of this grammar are numbered to provide a simple way to reference them in a parsing table.

[**Figure 4.5**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml?brand=vitalsource&create=true&favre=brett#P700101512400000000000000000143C) shows the LR parsing table for this grammar. Abbreviations are used for the actions: R for reduce and S for shift. R4 means reduce using rule 4; S6 means shift the next symbol of input onto the stack and push state 6 onto the stack. Empty positions in the ACTION table indicate syntax errors. In a complete parser, these could have calls to error-handling routines.

**Figure 4.5 The LR parsing table for an arithmetic expression grammar**

[**Figure 4.5 Full Alternative Text**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/longalt/la_fig04-05.xhtml#la_fig04-05)

LR parsing tables can easily be constructed using a software tool, such as yacc ([**Johnson, 1975**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004E5B)), which takes the grammar as input. Although LR parsing tables can be produced by hand, for a grammar of a real programming language, the task would be lengthy, tedious, and error prone. For real compilers, LR parsing tables are always generated with software tools.

The initial configuration of an LR parser is

The parser actions are informally defined as follows:

1. The Shift process is simple: The next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the ACTION table.
2. For a Reduce action, the handle must be removed from the stack. Because for every grammar symbol on the stack there is a state symbol, the number of symbols removed from the stack is twice the number of symbols in the handle. After removing the handle and its associated state symbols, the LHS of the rule is pushed onto the stack. Finally, the GOTO table is used, with the row label being the symbol that was exposed when the handle and its state symbols were removed from the stack, and the column label being the nonterminal that is the LHS of the rule used in the reduction.
3. When the action is Accept, the parse is complete and no errors were found.
4. When the action is Error, the parser calls an error-handling routine.

Although there are many parsing algorithms based on the LR concept, they differ only in the construction of the parsing table. All LR parsers use this same parsing algorithm.

Perhaps the best way to become familiar with the LR parsing process is through an example. Initially, the parse stack has the single symbol 0, which represents state 0 of the parser. The input contains the input string with an end marker, in this case a dollar sign, attached to its right end. At each step, the parser actions are dictated by the top (rightmost in [**Figure 4.4**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml?favre=brett#P7001015124000000000000000001418)) symbol of the parse stack and the next (leftmost in [**Figure 4.4**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml?favre=brett#P7001015124000000000000000001418)) token of input. The correct action is chosen from the corresponding cell of the ACTION part of the parse table. The GOTO part of the parse table is used after a reduction action. Recall that GOTO is used to determine which state symbol is placed on the parse stack after a reduction.

Following is a trace of a parse of the string id + id \* id, using the LR parsing algorithm and the parsing table shown in [**Figure 4.5**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP70010151240000000000000000013AC.xhtml?favre=brett#P700101512400000000000000000143C).

The algorithms to generate LR parsing tables from given grammars, which are described in [**Aho et al. (2006)**](https://jigsaw.yuzu.com/books/9780135102251/epub/OPS/xhtml/fileP7001015124000000000000000004D5C.xhtml#P7001015124000000000000000004D68), are not overly complex but are beyond the scope of a book on programming languages. As stated previously, there are a number of different software systems available to generate LR parsing tables.

SUMMARY

Syntax analysis is a common part of language implementation, regardless of the implementation approach used. Syntax analysis is normally based on a formal syntax description of the language being implemented. A context-free grammar, which is also called BNF, is the most common approach for describing syntax. The task of syntax analysis is usually divided into two parts: lexical analysis and syntax analysis. There are several reasons for separating lexical analysis—namely, simplicity, efficiency, and portability.

A lexical analyzer is a pattern matcher that isolates the small-scale parts of a program, which are called lexemes. Lexemes occur in categories, such as integer literals and names. These categories are called tokens. Each token is assigned a numeric code, which along with the lexeme is what the lexical analyzer produces. There are three distinct approaches to constructing a lexical analyzer: using a software tool to generate a table for a table-driven analyzer, building such a table by hand, and writing code to implement a state diagram description of the tokens of the language being implemented. The state diagram for tokens can be reasonably small if character classes are used for transitions, rather than having transitions for every possible character from every state node. Also, the state diagram can be simplified by using a table lookup to recognize reserved words.

Syntax analyzers have two goals: to detect syntax errors in a given program and to produce a parse tree, or possibly only the information required to build such a tree, for a given program. Syntax analyzers are either top-down, meaning they construct leftmost derivations and a parse tree in top-down order, or bottom-up, in which case they construct the reverse of a rightmost derivation and a parse tree in bottom-up order. Parsers that work for all unambiguous grammars have complexity  However, parsers used for implementing syntax analyzers for programming languages work on subclasses of unambiguous grammars and have complexity O(n).

A recursive-descent parser is an LL parser that is implemented by writing code directly from the grammar of the source language. EBNF is ideal as the basis for recursive-descent parsers. A recursive-descent parser has a subprogram for each nonterminal in the grammar. The code for a given grammar rule is simple if the rule has a single RHS. The RHS is examined left to right. For each nonterminal, the code calls the associated subprogram for that nonterminal, which parses whatever the nonterminal generates. For each terminal, the code compares the terminal with the next token of input. If they match, the code simply calls the lexical analyzer to get the next token. If they do not, the subprogram reports a syntax error. If a rule has more than one RHS, the subprogram must first determine which RHS it should parse. It must be possible to make this determination on the basis of the next token of input.

Two distinct grammar characteristics prevent the construction of a ­recursive- descent parser based on the grammar. One of these is left recursion. The process of eliminating direct left recursion from a grammar is relatively simple. Although we do not cover it, an algorithm exists to remove both direct and indirect left recursion from a grammar. The other problem is detected with the pairwise disjointness test, which tests whether a parsing subprogram can determine which RHS is being parsed on the basis of the next token of input. Some grammars that fail the pairwise disjointness test often can be modified to pass it, using left factoring.

The parsing problem for bottom-up parsers is to find the substring of the current sentential form that must be reduced to its associated LHS to get the next (previous) sentential form in the rightmost derivation. This substring is called the handle of the sentential form. A parse tree can provide an intuitive basis for recognizing a handle. A bottom-up parser is a shift-reduce algorithm, because in most cases it either shifts the next lexeme of input onto the parse stack or reduces the handle that is on top of the stack.

The LR family of shift-reduce parsers is the most commonly used ­bottom-up parsing approach for programming languages, because parsers in this family have several advantages over alternatives. An LR parser uses a parse stack, which contains grammar symbols and state symbols to maintain the state of the parser. The top symbol on the parse stack is always a state symbol that represents all of the information in the parse stack that is relevant to the parsing process. LR parsers use two parsing tables: ACTION and GOTO. The ACTION part specifies what the parser should do, given the state symbol on top of the parse stack and the next token of input. The GOTO table is used to determine which state symbol should be placed on the parse stack after a reduction has been done.