2

Specifying a programming language's grammar

This chapter covers

- The syntax of a programming language
- Specifying the grammar of a simple programming language
- Syntax diagrams and grammar files

Like any natural language that we use daily, a programming language must have rules to enable you to write programs in that language. These rules consist of the language's *syntax* — how to correctly write the language's statements and expressions — and the language's *semantics* — the meanings of those statements and expressions. You cannot build a useful programming language without a clear understanding of syntax and semantics. This chapter is about syntax, and the next chapter begins to discuss semantics.

2.1 A programming language's grammar

The syntax rules of a programming language comprise the *grammar* of the language. The grammar describes the *tokens* of the language and the rules regarding how to string the tokens together correctly to make the language's statements and expressions. A language's tokens are the fundamental "atoms" of the language, such as identifier names like `alpha`, numbers like `45.67`, reserved keywords like `if` and `else`, and special symbols like `+ * ( )` and `;`. A well-defined grammar enables you to write a syntactically correct program in the language, one that won't cause a compiler to produce syntax error messages when you attempt to compile it.

A syntactically correct program is not necessarily a meaningful one. As an analogy, consider the English language syntax rule *subject verb object*. With this rule, you can construct the syntactically correct (and meaningful) sentence *I ate lunch*. But the sentence *He walks table*

is syntactically correct but semantically meaningless. Of course, when writing a program, correct syntax and semantics won't guarantee that the program is logically correct and generates the right runtime output.

Every translator that you build in this book—converter, interpreter, and compiler—depends heavily on the grammar. *The grammar is the formal specification of a programming language.* Once you've defined the grammar for the language, you'll be able to use software tools such as ANTLR 4 that automatically generate parts of the translator.

2.2 Syntax diagrams

Let's start with a programming language which we'll name Simple. This is a very simple language where variable names and numbers are restricted to only one letter or digit each, respectively. Since Simple contains only addition and subtraction operators, you don't yet need to deal with operator precedence. You will in later chapters where our grammars will have more sophisticated expressions.

Here is an example of a Simple program:

Listing 2.1 A Simple program `example.simp`

```
i := 2;
writeln(i);

j := 3;
k := i + j;
writeln(k);

n := i + j - 7;
k := n;
writeln(k);
```

Based on this example, we can describe the syntax rules of the language's grammar in English:

1. A program consists of one statement or multiple statements separated by semicolons.
2. A statement is either an assignment statement, a write statement, or an empty statement.
3. An assignment statement is a variable followed by the special symbol := followed by an expression.
4. An expression is one operand or multiple operands separated by operators.
5. An operand is either a variable or a number.
6. A variable is one of the letters `a` through `z`.
7. A number is one of the digits `0` through `9`.
8. An operator is either the special symbol `+` or the special symbol `-`.
9. A write statement is the word `writeln` followed by the special symbol `(` followed by a variable followed by the special symbol `)`.

As you can see, even a simple programming language, it can be quite tedious to write the syntax rules in a natural language such as English. Natural-language descriptions of a grammar can be ambiguous. You need a more precise specification that both we humans and software tools can interpret consistently.

Another common way to specify a language's grammar is with *syntax diagrams*. These are also known as *railroad diagrams* due to their appearance and how you read them.

The syntax diagrams that express rules 1 and 2:

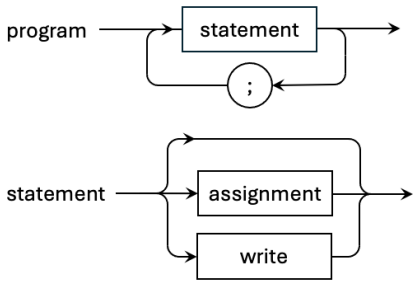


Figure 2.1: Syntax diagrams for program and statement. Follow the paths in the direction of the arrows.

The first diagram in figure 2.1 states that a program consists of a single statement (follow the horizontal path straight through) or several statements separated by semicolons (follow as many times as desired the curved path that loops back). The second diagram states that a statement is either an assignment statement or a write statement. It can also be an empty statement. An empty statement (one that does nothing at run time) is useful in a grammar to handle extra semicolons in a program, such as two consecutive semicolons or one after the last statement of a program.

Figure 2.2 shows the syntax diagrams for rules 3 and 4.

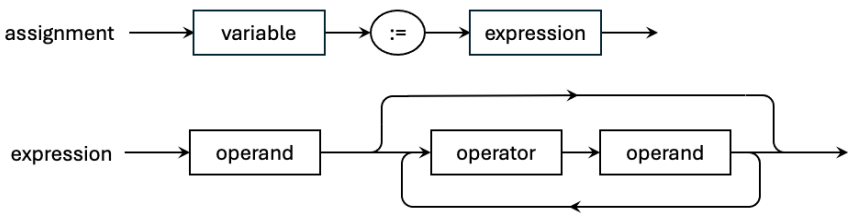


Figure 2.2: Syntax diagrams for assignment statement and expression.

Figure 2.3 shows the syntax diagrams for rules 5, 6, 7, and 8.

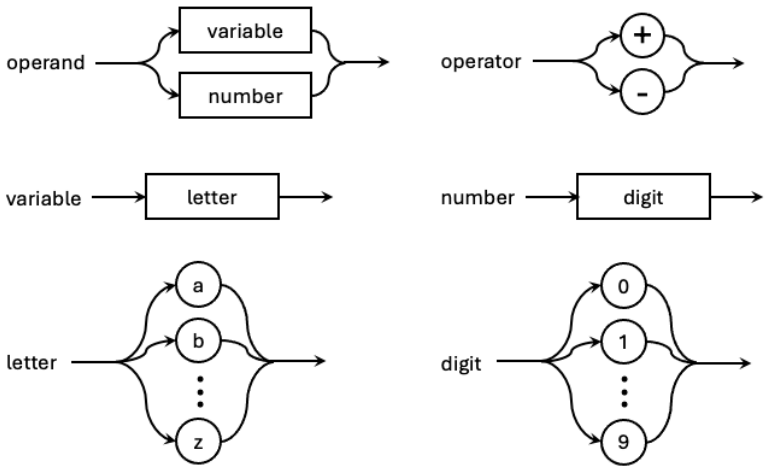


Figure 2.3: Syntax diagrams for operand, operator, variable, number, letter, and digit.

Finally, figure 2.4 shows the syntax diagram for the write statement.

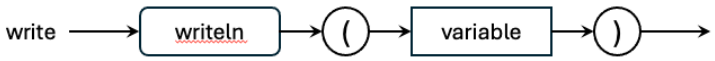


Figure 2.4: Syntax diagram for the write statement.

In these diagrams, rectangular boxes with sharp corners, such as for statement and expression, refer to other diagrams. Circular boxes and boxes with rounded corners, such as for the special symbols, letter, digits, and the word `writeln`, contain literal text that's part of the Simple language.

You can use syntax diagrams to represent the syntax rules of an entire language far more sophisticated than Simple. The major advantage of using these graphical diagrams is that they are easy for us humans to read and understand.

Syntax rules are also called *production rules*. Viewed from a different perspective, these rules can "produce" legitimate statements and expressions of the language.

2.3 A grammar file for ANTLR 4

In this book, you will use ANTLR 4, a popular compiler development tool, to generate major components of our programming language translators. Starting in chapter 4, you will see the great advantages of using such a powerful tool. One big advantage is that ANTLR 4 will generate translator components written in Java.

ANTLR 4 needs to know the syntax rules of a language. Unfortunately, it isn't able to read graphical syntax diagrams, and so you must represent the rules in a textual grammar file that

ANTLR 4 can read and understand. Listing 2.2 is `Simple.g4`, the grammar file containing the syntax rules of Simple in ANTLR 4's format. The names of ANTLR 4 grammar files use the suffix `.g4` by convention.

Listing 2.2: ANTLR 4 grammar file `Simple.g4`

```
grammar Simple ;

program : statement ( ';' statement )* ;          #A
                                                  #A
statement : assignment                            #A
          | write                                 #A
          | /* empty */                           #A
          ;                                       #A
                                                  #A
assignment : variable ':=' expression ;           #A
expression : operand ( operator operand )* ;      #A
operand    : variable | number ;                  #A
operator   : '+' | '-' ;                          #A
                                                  #A
write : WRITELN '(' variable ')' ;                #A
                                                  #A
variable : LETTER ;                               #A
number   : DIGIT ;                                #A

WRITELN : 'writeln' ;               #B
LETTER  : [a-z] ;                   #B
DIGIT   : [0-9] ;                   #B
                                    #B
WHITESPACE : [ \n\r\t]+ -> skip ;   #B
```

#A Syntax rules (production rules)
#B Token definitions

This short grammar file contains many examples of ANTLR 4's grammar notation. You can see the familiar names of the syntax rules `program`, `statement`, `assignment`, `write`, `expression`, `operand`, `operator`, `variable`, and `number`. The names of the language tokens are in all capitals: `WRITELN`, `LETTER`, and `DIGIT`. Literal text that is part of Simple's language are enclosed in single quotes, for example, `';'`, `':='`, `'+'`, `'('`, `')'`, and `'writeln'`.

Rule `variable` says that a variable is a `LETTER` token, which in turn is defined to be the character class of the letters a through z. Similarly, rule `number` says that a number is a `DIGIT` token, which is defined to be the character class of the digits 0 through 9. Token `WHITESPACE` is defined to be the character class of a blank, line feed (`\n`), carriage return (`\r`), and tab (`\t`) characters repeated one or more times, and those characters should be skipped (ignored) in a Simple program. Token `WRITELN` is the string `writeln`.

An ANTLR 4 grammar file uses metasymbols that have special meanings for ANTLR 4 but that are not necessarily part of the Simple language. The following table lists the metasymbols and their purposes in the grammar.

Table 2.1: Metasymbols in an ANTLR 4 grammar file.

Metasymbols	Purposes
`:`	Separates a syntax rule or token name from its definition.
`;`	Ends each syntax rule or token definition.
`|`	Separates alternative elements in a syntax rule.
`( )`	Groups elements of a syntax rule.
`' '`	Surrounds literal text that's part of the source language.
`*`	Follows an element or a group that should appear zero or more times.
`+`	Follows an element or a group that should appear at least once.
`?`	Follows an element or a group that is optional (appear zero or one time).
`.`	Represents any character.
`\`	Quote the following metacharacter to be a character in the source language. Examples: `\'` and `\;` represent a single quote and a semicolon in the source language, respectively.
`\t \n \r`	Represents the tab, line feed, and carriage return characters, respectively.
`[ ]`	Encloses a class of characters. Example: `[ \t\n]` is the class containing a blank, tab, and line feed characters.
`-`	Specifies a range of characters in a character class. Example: `[a-zA-z]` is the class containing all the lower- and upper-case letters.
`~`	Any characters except the following. Examples: `~('"')` is any character except a double quote. `~('\'')` is any character except a single quote. `~[\n\r]` is any character except a line feed or carriage return.
`-> skip`	Elements in a source language to ignore.

With this table, you should be able to compare `Simple.g4` in listing 2.2 with the syntax diagrams in figures 2.1 – 2.4.

2.4 Automatic generation of syntax diagrams

It is possible to automatically generate graphical syntax diagrams from an ANTLR 4 grammar file. A popular utility is `rrd-antlr4`.

Syntax diagram generator rrd-antlr4

Go to the GitHub repository at https://github.com/bkiers/rrd-antlr4 to download and install the Java-based "railroad diagram" utility `rrd-antlr4`. Given an ANTLR 4 grammar file, it generates the HTML code to display the grammar rules as graphical syntax diagrams.

Figure 2.5 shows the results of running the command

```
java -jar rrd-antlr4-0.1.2.jar Simple.g4
```

Simple.g4

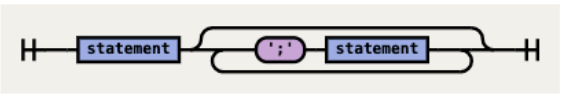
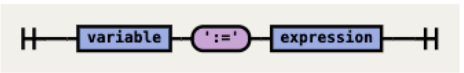
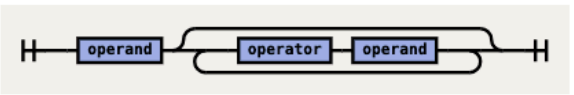
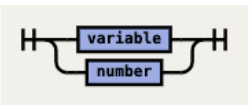
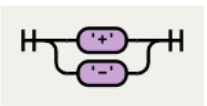
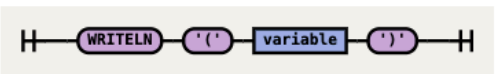
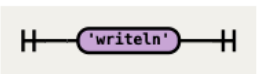
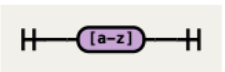
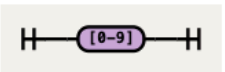
program	
statement	
assignment	
expression	
operand	
operator	
write	
variable	
number	
WRITELN	
LETTER	
DIGIT	
WHITESPACE	

Figure 2.5: Output from utility `rrd-antlr4` from grammar file `Simple.g4`.

Hand-drawn vs. computer-generated syntax diagrams

I personally think my syntax diagrams in figures 2.1 – 2.4, which I created using a simple drawing tool, are easier to understand. But they are tedious to draw! Since you'll be working with ANTLR 4 grammar files throughout this book, it's worth knowing that this utility can generate diagrams automatically from them.

2.5 Formal definition of a grammar and EBNF notation

Linguists and computer scientists have a formal definition of a grammar. In this definition, a grammar has four parts:

- *Terminal symbols*: These are the tokens of the language, including its special symbols and reserved words. Examples from our language include `:=`, `+`, `-`, `writeln`, and the individual letters and digits.
- *Nonterminal symbols*: These are the names of the syntax rules. Examples from our grammar include `program`, `statement`, and `expression`, `variable`, and `number`.
- *Production rules*: The syntax rules.
- *Start symbol*: The nonterminal symbol that is at the head of the syntax rules. In our grammar, the start symbol is `program`.

A classic notation for grammars is called BNF, for Backus-Naur form, named after its developers John Backus and Peter Naur. The ANTLR 4 grammar file format is based on EBNF (extended BNF) which adds the metasymbols `+` and `*` and others. Listing 2.3 shows the Simple grammar file in a variant of EBNF.

Listing 2.3: The Simple grammar in EBNF

```
program = statement ( ";" statement )*

statement = assignment | write

assignment = variable ":=" expression
expression = operand ( operator operand )*
operand    = variable | number
operator   = "+" | "-"

write = "writeln" "(" variable ")"

variable = letter
number   = digit

letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" |
         "h" | "i" | "j" | "k" | "l" | "m" | "n" |
         "o" | "p" | "q" | "r" | "s" | "t" | "u" |
```

```
          "v" | "w" | "x" | "y" | "z"
digit   = "0" | "1" | "2" | "3" | "4" |
          "5" | "6" | "7" | "8" | "9"

whitespace = ( "\n" | "\r" | "\t" )+
```

Comparing listings 2.2 and 2.3, you can see that EBNF uses equal signs instead of colons and double quotes instead of single quotes. As you'll see starting in chapter 4, ANTLR 4 will use a grammar file not only to specify a language's grammar but also to generate translator components, whereas an EBNF grammar is primarily a formal specification. Therefore, the latter doesn't distinguish tokens from the syntax rules, and it doesn't note that whitespace should be ignored.

2.6 Summary

- The syntax rules of a programming language specify how to write syntactically correct programs.
- The semantics of a language give the meanings of a program's statements and expressions.
- The syntax rules incorporate the language's tokens, which are the language's basic elements.
- You can represent a language's syntax rules with graphical syntax diagrams, which are easy for humans to read and understand.
- The compiler-development tool ANTLR 4 requires the syntax rules and token definitions in a text-based grammar file written in the format that it can read and understand. The format is based on extended Backus-Naur form (EBNF).
- Utility program `rrd-antlr4` can automatically generate syntax diagrams from an ANTLR 4 grammar file.

In the next chapter, you'll learn how to apply good software engineering principles to design the architecture and the control and data flows of translators.

2.7 Exercise

What new constructs do you want your programming language to have? Create an ANTLR 4 grammar file that contains these constructs and use `rrd-antlr4` to generate syntax diagrams.