2 Grammars

The term "grammar" has a wide range of definitions and nuances and it is hardly surprising that we need a tight and formal definition for it when used in the context of programming languages. A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

if (expression) statement **else** statement

that is, an if-else statement is the concatenation of the keyword **if**, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword **else**, and another statement. Using the variable ***expr*** to denote an expression and the variable ***stmt*** to denote a statement, this structuring rule can be expressed as

stmt -> **if** (*expr*) *stmt* **else** *stmt*

in which the arrow may be read as "can have the form". Such a rule is called a production. In a production, lexical elements like the keyword **if** and the parenthesis are called ***terminals***. Variables like ***expr*** and ***stmt*** represent sequences of terminals and are called ***nonterminals***.

2.1 Definition of Grammars

The *grammar* (*G*) of a language is defined as a 4-tuple *G* = (*N*,*T*, *S*, *P*) where:

- *N* is the finite set of non-terminal symbols, sometimes called "syntactic variables". Each nonterminal represents a set of strings of terminals, in a manner we shall describe.
- *T* is the finite set of terminal symbols (*N* and *T* are disjoint), sometimes referred to as "tokens". The terminals are the elementary symbols of the language defined by the grammar.

- S is the *starting symbol*, $S \in N$. The starting symbol is the unique non-terminal symbol that is used as the starting point for the generation of all the strings of the language.
- P is the finite set of *production rules*. A production rule defines a string transformation and it has the general form $a \rightarrow \beta$. This rule specifies that any occurrence of the string a in the string to be transformed is replaced by the string β. There are constraints on the constitution of the strings a and β. If U is defined by

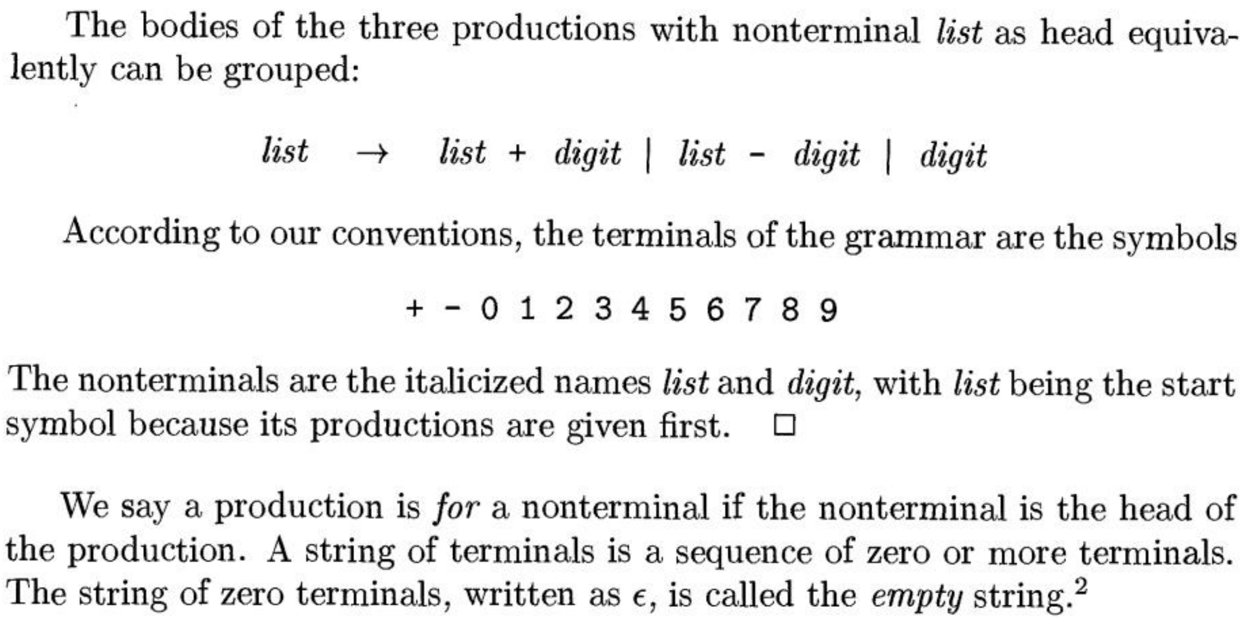
  $U = N \cup T$ (i.e. U is the set of all non-terminal and terminal symbols), then

  - a has to be a member of the set of all non-empty strings that can be formed by the concatenation of members of U, and it has to contain at least one member of N.
  - β has to be a member of the set of all strings that can be formed by the concatenation of members of U, including the empty string (i.e. $\beta \in U^*$).

We specify grammars by listing their productions, with the productions for the start symbol listed first. We assume that digits, signs such as < and <=, and boldface strings such as **while** are terminals. An italicized name is a nonterminal, and any nonitalicized name or symbol may be assumed to be a terminal.[1] For notational convenience, productions with the same nonterminal as the head can have their bodies grouped, with the alternative bodies separated by the symbol |, which we read as "or."

Example 2.1: Several examples in this chapter use expressions consisting of digits and plus and minus signs; e.g., strings such as 9-5+2, 3-1, or 7. Since a plus or minus sign must appear between two digits, we refer to such expressions as "lists of digits separated by plus or minus signs." The following grammar describes the syntax of these expressions. The productions are:

$$\begin{array}{lcll} list & \rightarrow & list \; + \; digit & (2.1) \\ list & \rightarrow & list \; - \; digit & (2.2) \\ list & \rightarrow & digit & (2.3) \\ digit & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 & (2.4) \end{array}$$



The bodies of the three productions with nonterminal *list* as head equivalently can be grouped:

$$list \quad \rightarrow \quad list + digit \mid list - digit \mid digit$$

According to our conventions, the terminals of the grammar are the symbols

$$\texttt{+ - 0 1 2 3 4 5 6 7 8 9}$$

The nonterminals are the italicized names *list* and *digit*, with *list* being the start symbol because its productions are given first. □

We say a production is *for* a nonterminal if the nonterminal is the head of the production. A string of terminals is a sequence of zero or more terminals. The string of zero terminals, written as ϵ, is called the *empty* string.[2]

Note:

Grammars play a crucial role in specifying the syntax of a programming language. They are used to define the rules and structure of valid programs in the language. Grammars are typically formalized using formal language theory*, with context-free grammars (CFGs) being one of the most commonly used formalisms.*

Grammars are used across many of the compilation stages as described below:

1. Defining Syntax: *Grammars are used to define the syntax of a programming language. This includes specifying the rules for constructing valid programs, such as the order of statements, expressions, declarations, and other language constructs.*
2. Parsing*: Once a grammar is defined, it can be used to parse source code. Parsing involves analyzing the input program according to the rules specified in the grammar to determine its syntactic structure. This process is often done using parsing algorithms such as recursive descent parsing, LL parsing, or LR parsing.*
3. Abstract Syntax Trees (AST)*: During parsing, the syntactic structure of the program is typically represented as an abstract syntax tree (AST). Each node in the AST corresponds to a specific language construct, and the tree structure reflects the hierarchical*

relationships between these constructs. Grammars dictate how the AST is constructed during parsing.

4. **Semantic Analysis**: *Grammars may also include semantic rules in addition to syntactic rules. Semantic analysis is the phase of compilation where the meaning of the program is analyzed. This includes type checking, scope resolution, and other checks to ensure that the program adheres to the language's semantic rules.*
5. **Code Generation**: *Once the input program has been parsed and analyzed, the compiler can generate executable code or intermediate representations (such as bytecode or assembly code). The structure and content of the generated code are heavily influenced by the rules defined in the grammar.*
6. **Error Handling**: *Grammars also help in error handling during parsing and semantic analysis. When a syntax error is encountered, the parser can often provide meaningful error messages based on the expected structure defined by the grammar.*

Chomsky Hierarchy

Looking at the definition of a grammar in the last section, it is clear that the important and potentially problematic component is P, the set of production rules. A production rule has the form $\alpha \rightarrow \beta$, loosely translated as "anything can be transformed to anything" (although we have already stated some restrictions on the content of α and β). The key question then is to remove this generality by restricting the forms of the production rules to see whether less general rules can be useful for defining and analysing computer programming languages.

In the 1950s, Noam Chomsky produced a hierarchical classification of formal grammars which provides a framework for the definition of programming languages as well as for the analysis of programs written in these languages. This hierarchy is made up of four levels, as follows:

- A *Chomsky type 0* or a *free grammar* or an *unrestricted grammar* contains productions of the form $\alpha \rightarrow \beta$. The restrictions on α and β are those already mentioned in the section above. This was our starting point in the definition of a grammar and as suggested, these grammars are not sufficiently restricted to be of any practical use for programming languages.

- A *Chomsky type 1* or a *context-sensitive grammar* has productions of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ where $\alpha,\beta,\gamma \in U^*$, γ is non-null and A is a single non-terminal symbol. The left context is α, the right context is β and in this particular context, A is transformed to γ.

  This type of grammar turns out to have significant relevance to programming languages. The concept of context is central to programming language definition—"…this statement is only valid in the context of an appropriate declaration of *i* …", for example. However, in practice, these grammars do not turn out to be particularly helpful because defining the context-sensitive aspects of a programming language in terms of a type 1 grammar can turn into a nightmare of complexity. So we need to simplify further and specify context-sensitive aspects by resorting to other means, such as English language descriptions.
- A *Chomsky type 2* or a *context-free grammar* has productions of the form $A \rightarrow \gamma$ where A is a single non-terminal symbol. These productions correspond directly to BNF rules. In general, if BNF can be used in the definition of a language, then that language is no more complex than Chomsky type 2.

  These grammars are central to the definition and analysis of the majority of programming languages. Despite the simplicity of the productions they are capable of defining powerful and complex program syntax. Programming languages are generally defined using type 2 grammars and these grammars are used directly in the production of code for program analysis.
- A *Chomsky type 3* or a *regular grammar* or a *finite-state grammar* puts further restrictions on the form of the productions. Here, all productions are of the form $A \rightarrow a$ or $A \rightarrow aB$ where A and B are non-terminal symbols and a is a terminal symbol. These grammars turn out to be far too restrictive for defining the syntax of conventional programming languages, but do have a key place in the specification of the syntax of the basic lexical tokens dealt with by the lexical analysis phase of a compiler. They are the grammars for

languages that can be specified using *regular expressions* and programs in these languages can be recognized using a *finite-state machine*.

2.2 Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.

Example 2.2: The language defined by the grammar of Example 2.1 consists of lists of digits separated by plus and minus signs. The ten productions for the nonterminal *digit* allow it to stand for any of the terminals `0`, `1`, ... , `9`. From production (2.3), a single digit by itself is a list. Productions (2.1) and (2.2) express the rule that any list followed by a plus or minus sign and then another digit makes up a new list.

Productions (2.1) to (2.4) are all we need to define the desired language. For example, we can deduce that `9-5+2` is a *list* as follows.

a) `9` is a *list* by production (2.3), since `9` is a *digit*.

b) `9-5` is a *list* by production (2.2), since `9` is a *list* and `5` is a *digit*.

c) `9-5+2` is a *list* by production (2.1), since `9-5` is a *list* and `2` is a *digit*.

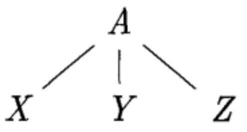
Example 2.3: A somewhat different sort of list is the list of parameters in a function call. In Java, the parameters are enclosed within parentheses, as in the call `max(x,y)` of function `max` with parameters `x` and `y`. One nuance of such lists is that an empty list of parameters may be found between the terminals `(` and `)`. We may start to develop a grammar for such sequences with the productions:

Note that the second possible body for *optparams* ("optional parameter list") is ϵ, which stands for the empty string of symbols. That is, *optparams* can be replaced by the empty string, so a *call* can consist of a function name followed by the two-terminal string `()`. Notice that the productions for *params* are analogous to those for *list* in Example 2.1, with comma in place of the arithmetic operator `+` or `-`, and *param* in place of *digit*. We have not shown the productions for *param*, since parameters are really arbitrary expressions. Shortly, we shall discuss the appropriate productions for the various language constructs, such as expressions, statements, and so on. □

Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string. Parsing is one of the most fundamental problems in all of compiling; the main approaches to parsing are discussed in Chapter 4. In this chapter, for simplicity, we begin with source programs like `9-5+2` in which each character is a terminal; in general, a source program has multicharacter lexemes that are grouped by the lexical analyzer into tokens, whose first components are the terminals processed by the parser.

2.3 Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production $A \rightarrow XYZ$, then a parse tree may have an interior node labeled A with three children labeled X, Y, and Z, from left to right:



Formally, given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by ϵ.
3. Each interior node is labeled by a nonterminal.
4. If A is the nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1X_2 \cdots X_n$. Here, $X_1, X_2, \ldots, X_n$ each stand for a symbol that is either a terminal or a nonterminal. As a special case, if $A \rightarrow \epsilon$ is a production, then a node labeled A may have a single child labeled ϵ.

Example 2.4: The derivation of 9-5+2 in Example 2.2 is illustrated by the tree in Fig. 2.5. Each node in the tree is labeled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the head of the production, the children to the body.

In Fig. 2.5, the root is labeled *list*, the start symbol of the grammar in Example 2.1. The children of the root are labeled, from left to right, *list*, +, and *digit*. Note that

$$list \rightarrow list + digit$$

is a production in the grammar of Example 2.1. The left child of the root is similar to the root, with a child labeled - instead of +. The three nodes labeled *digit* each have one child that is labeled by a digit. □

From left to right, the leaves of a parse tree form the *yield* of the tree, which is the string *generated* or *derived* from the nonterminal at the root of the parse

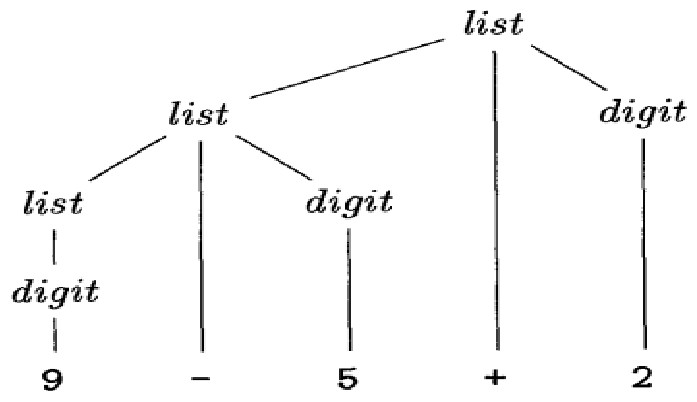


Fig 2.5 Parse tree for 9-5+2 according to the grammar in Example 2.1

2.4 Ambiguity

We have to be careful in talking about *the* structure of a string according to a grammar. A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous*. To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree. Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

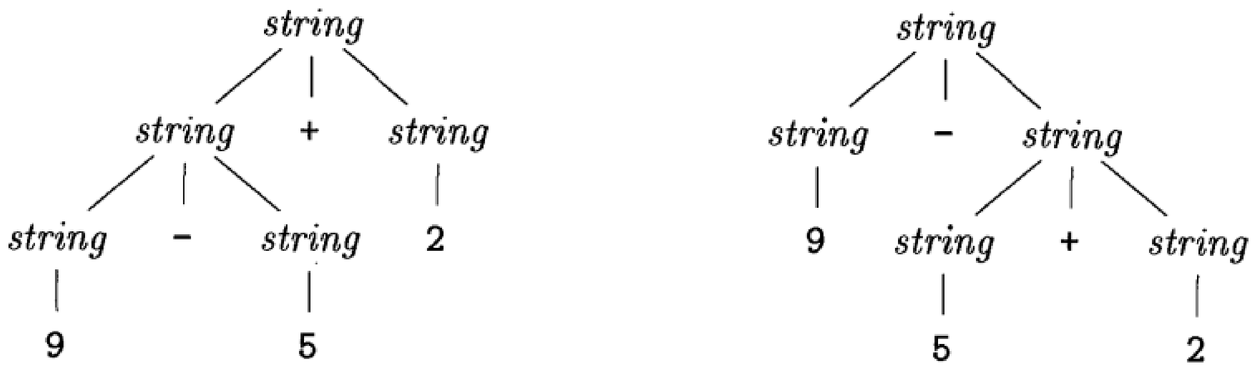


Fig 2.6 Two parse trees for 9-5+2

2.5 Associativity of Operators

By convention, 9+5+2 is equivalent to (9+5)+2 and 9-5-2 is equivalent to (9-5)-2. When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand. We say that the operator + *associates* to the left, because an operand with plus signs on both sides of it belongs to the operator to its left. In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative.

Some common operators such as exponentiation are right-associative. As another example, the assignment operator = in C and its descendants is right-associative; that is, the expression a=b=c is treated in the same way as the expression a=(b=c).

Strings like a=b=c with a right-associative operator are generated by the following grammar:

$$\begin{array}{lcl} \textit{right} & \rightarrow & \textit{letter} = \textit{right} \mid \textit{letter} \\ \textit{letter} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \end{array}$$

The contrast between a parse tree for a left-associative operator like - and a parse tree for a right-associative operator like = is shown by Fig. 2.7. Note that the parse tree for 9-5-2 grows down towards the left, whereas the parse tree for a=b=c grows down towards the right.

2.6 Precedence of Operators

Consider the expression 9+5*2. There are two possible interpretations of this expression: (9+5)*2 or 9+(5*2). The associativity rules for + and * apply to occurrences of the same operator, so they do not resolve this ambiguity. Rules defining the relative precedence of operators are needed when more than one kind of operator is present.

We say that * has *higher precedence* than + if * takes its operands before + does. In ordinary arithmetic, multiplication and division have higher precedence than addition and subtraction. Therefore, 5 is taken by * in both 9+5*2 and 9*5+2; i.e., the expressions are equivalent to 9+(5*2) and (9*5)+2, respectively.

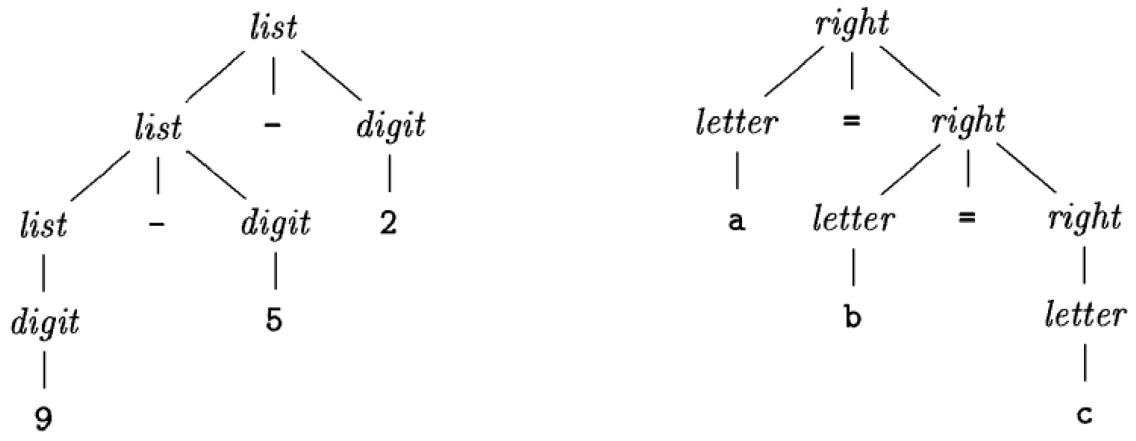


Fig 2.7 Parse trees for left and right associativity grammars