Chapter 2
Syntax Analysis

Syntax and vocabulary are overwhelming constraints—the rules that run us. Language is using us to talk—we think we're using the language, but language is doing the thinking, we're its slavish agents.

Harry Mathews (1930–2017)

Where lexical analysis splits a text into tokens, the purpose of syntax analysis (also known as *parsing*) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This "something" is typically a data structure called the *syntax tree* of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree, and how the interior nodes of the tree are labelled.

In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting *syntax errors*.

As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation suitable for human understanding and algebraic manipulation is transformed into a machine-like low-level notation suitable for efficient execution. This process is called *parser generation*.

The notation we use for human manipulation is *context-free grammars*,[1] which is a recursive notation for describing sets of strings and imposing a structure on each such string. This notation can in some cases be translated almost directly into recursive programs, but it is often more convenient to generate *stack automata*. These are similar to the finite automata used for lexical analysis, but they can additionally use a stack, which allows counting and non-local matching of symbols. We shall see

[1] Because derivation is independent of context, as we will see.

T. Æ. Mogensen, *Introduction to Compiler Design*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-46460-7_2

two ways of generating such automata. The first of these, LL(1), is relatively simple, but works only for a somewhat restricted class of grammars. The SLR construction, which we present later, is more complex but handles a wider class of grammars. Sadly, neither of these work for *all* context-free grammars. Tools that handle all context-free grammars do exist, but they can incur a severe speed penalty, which is why most parser generators restrict the class of input grammars to what can be parsed efficiently using a specific method. This may require rewriting an otherwise correct grammar for a language into a form that can be handled by the method or by providing additional information to resolve ambiguities. We will show some techniques for doing so.

2.1 Context-Free Grammars

Like regular expressions, context-free grammars describe sets of strings, i.e., languages. Additionally, a context-free grammar also defines structure on the strings in the language it defines. A language is defined over some alphabet, for example the set of tokens produced by a lexer or the set of alphanumeric characters. The symbols in the alphabet are called *terminals*.

A context-free grammar recursively defines several sets of strings. Each set is denoted by a name, which is called a *nonterminal*. The set of nonterminals is disjoint from the set of terminals. One of the nonterminals are chosen to denote the main language described by the grammar. This nonterminal is called the *start symbol* of the grammar, and plays a role similar to the start state of a finite automaton. The sets are described by a number of *productions*. Each production describes some of the possible strings that are contained in the set denoted by a nonterminal. A production has the form

$$N \to X_1 \dots X_n$$

where N is a nonterminal and $X_1 \dots X_n$ are zero or more symbols, each of which is either a terminal or a nonterminal. The intended meaning of this notation is to say that the set denoted by N contains strings that are obtained by concatenating strings from the sets denoted by $X_1 \dots X_n$. In this setting, a terminal denotes a set consisting of a single string consisting of a single symbol, just like an alphabet character in a regular expression denotes a set consisting of a single string consisting of a single character. We will, when no confusion is likely, equate a nonterminal with the set of strings it denotes, like we did for alphabet characters in regular expressions.

Some examples:

$$A \to \texttt{a}$$

says that the set denoted by the nonterminal A contains the one-character string `a`.

$$A \to \texttt{a}A$$

says that the set denoted by A contains all strings formed by putting an a in front of a string taken from the set denoted by A. Together, these two productions indicate that A contains all non-empty sequences of as and is hence (in the absence of other productions) equivalent to the regular expression $\mathtt{a}^+$.

We can define a grammar equivalent to the regular expression $\mathtt{a}^*$ by the two productions

$$\begin{aligned} B &\rightarrow \\ B &\rightarrow \mathtt{a}B \end{aligned}$$

where the first production indicates that the empty string is part of the set B. Compare this grammar with the definition of s^* in Fig. 1.1.

Productions with empty right-hand sides are called *empty productions*. These are in some variants of grammar notation written with an ε on the right hand side instead of leaving it empty.

So far, we have not described any set that could not just as well have been described using regular expressions. Context-free grammars are, however, capable of expressing much more complex languages. In Sect. 1.9, we noted that the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ is not regular. It is, however, easily described by the grammar

$$\begin{aligned} S &\rightarrow \\ S &\rightarrow \mathtt{a}S\mathtt{b} \end{aligned}$$

The second production ensures that the as and bs are paired symmetrically around the middle of the string, so they occur in equal number.

The examples above have used only one nonterminal per grammar. When several nonterminals are used, we must make it clear which of these is the start symbol. By convention (if nothing else is stated), the nonterminal on the left-hand side of the first production is the start symbol. As an example, the grammar

$$\begin{aligned} T &\rightarrow R \\ T &\rightarrow \mathtt{a}T\mathtt{a} \\ R &\rightarrow \mathtt{b} \\ R &\rightarrow \mathtt{b}R \end{aligned}$$

has T as start symbol and denotes the set of strings that start with any number of as followed by a non-zero number of bs and then the same number of as with which it started.

In some variants of grammar notation, a shorthand notation is used where all the productions of the same nonterminal are combined to a single rule, using the alternative symbol (|) from regular expressions to separate the right-hand sides. In this notation, the above grammar would read

$$\begin{aligned} T &\rightarrow R \mid \mathtt{a}T\mathtt{a} \\ R &\rightarrow \mathtt{b} \mid \mathtt{b}R \end{aligned}$$

There are still four productions in the grammar, even though the arrow symbol $\rightarrow$ is only used twice. Some grammar notations (such as EBNF, the Extended Backus-Naur Form) also allow equivalents of ?, * and $^+$ from regular expressions. With such shorthands, we can write the above grammar as

$$T \rightarrow \texttt{b}^+ \mid \texttt{a}T\texttt{a}$$

We will in this book, for simplicity, stick to basic grammar notation without shorthands. In the grammar notation we use, any name with initial capital letter and written in *Italics* (and possibly with subscripts or superscripts) denotes a nonterminal. Any symbol in `typewriter font` denotes itself as a terminal. Any name written in **boldface** denotes a lexical *token* that represents a *set of* concrete strings such as number constants or variable names. In examples of strings that are derived from a grammar, we may replace a boldface token by an element of the corresponding set of strings. For example, if the token **num** represents integer constants, we might write `5` or `7` instead.

2.1.1 How to Write Context-Free Grammars

As hinted above, a regular expression can systematically be rewritten to an equivalent context-free grammar by using a nonterminal for every subexpression in the regular expression, and using one or two productions for each nonterminal. The construction is shown in Fig. 2.1. So, if we can represent a language as a regular expression, it is easy to make a grammar for it.

We will also use context-free grammars to describe non-regular languages. An example of a non-regular language is the kind of arithmetic expressions that are part of most programming languages (and also found on electronic calculators), and which consist of numbers, operators, and parentheses. If arithmetic expressions *do not* have parentheses, the language can be described by a regular expression such as

$$\textbf{num}((+\mid -\mid *\mid /)\textbf{num})*$$

where **num** represents any number constant.

However, if we *do* include parentheses (and these must match), the language can, as mentioned in Sect. 1.9, *not* be described by a regular expression, as a regular expression can not "count" the number of unmatched opening parentheses at a particular point in the string. Even without parentheses, the regular description above is not useful if you want operators to have different precedence, as it treats the expression as a flat string rather than as having structure. Arithmetic expressions with parentheses can be described by context-free grammars such as Grammar 2.2. This grammar, however, does not distinguish operators by precedence. We will look at structure and precedence rules in Sects. 2.2.1 and 2.3.

Form of s_i	Productions for N_i
ε	$N_i \rightarrow$
`a`	$N_i \rightarrow$ `a`
$s_j s_k$	$N_i \rightarrow N_j N_k$
$s_j \vert s_k$	$N_i \rightarrow N_j$ $N_i \rightarrow N_k$
s_j*	$N_i \rightarrow N_j N_i$ $N_i \rightarrow$
s_j+	$N_i \rightarrow N_j N_i$ $N_i \rightarrow N_j$
$s_j?$	$N_i \rightarrow N_j$ $N_i \rightarrow$

Each subexpression of the regular expression is numbered, and subexpression s_i is described by a nonterminal N_i. The productions for N_i depend on the shape of s_i as shown in the table above.

Fig. 2.1 Converting regular expressions to context-free grammars

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp \\
Exp &\rightarrow Exp - Exp \\
Exp &\rightarrow Exp * Exp \\
Exp &\rightarrow Exp \,/\, Exp \\
Exp &\rightarrow \textbf{num} \\
Exp &\rightarrow (\ Exp\ )
\end{aligned}
$$

Grammar 2.2 Simple expression grammar

Most constructions from programming languages are easily expressed by context-free grammars. In fact, the syntax most modern programming languages are designed to be mostly or completely describable using context-free grammars.

When writing a grammar for a programming language, one normally starts by dividing the constructs of the language into different *syntactic categories*. A syntactic category is a sub-language that embodies a particular language concept. Examples of common syntactic categories in programming languages are:

Expressions	are used to express calculation of values.
Statements	express actions that occur in a particular sequence.
Declarations	define properties of named entities such as variables or functions used in other parts of the program.
Types	are used in declarations to limit the kinds of values a variable can have or the parameters a function can take and the results it can return.

Each syntactic category is denoted by a nonterminal, e.g., *Exp* from Grammar 2.2. More than one nonterminal might be needed to describe a single syntactic category or to provide structure to elements of the syntactic category, as we shall see

Grammar 2.3 Simple statement grammar

Stat → **id** := *Exp*
Stat → *Stat* ; *Stat*
Stat → `if` *Exp* `then` *Stat* `else` *Stat*
Stat → `if` *Exp* `then` *Stat*

later, but a selected nonterminal is the main nonterminal for the syntactic category. Productions for one syntactic category can refer to nonterminals for other syntactic categories. For example, statements may contain expressions, so some of the productions for statements use the main nonterminal for expressions. A simple grammar for statements might look like Grammar 2.3, which refers to the *Exp* nonterminal from Grammar 2.2. The terminal **id** represents variable names.

Suggested exercises: 2.3 (ignore, for now, the word "unambiguous"), 2.21(a).

2.2 Derivation

So far, we have just appealed to intuitive notions of recursion when we describe the set of strings that a grammar produces. To formally define the set of strings that a grammar describes, we use *derivation*, as we did for regular expressions, except that the derivation rules are different. An advantage of using derivations is, as we will later see, that syntax analysis is closely related to derivation.

The basic idea of derivation is to consider productions as rewrite rules: Whenever we have a nonterminal, we can replace this by the right-hand side of *any* single production where the nonterminal appears on the left-hand side. We can do this anywhere in a sequence of symbols (terminals and nonterminals) and repeat doing so until we have only terminals left. The resulting sequence of terminals is a string in the language defined by the grammar. Formally, we define the derivation relation $\Rightarrow$ by the three rules

1. $\alpha N \beta \Rightarrow \alpha\gamma\beta$ if there is a production $N \to \gamma$
2. $\alpha \Rightarrow \alpha$
3. $\alpha \Rightarrow \gamma$ if there is a β such that $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$

where α, β and γ are (possibly empty) sequences of grammar symbols (terminals and nonterminals). The first rule states that using a production as a rewrite rule (anywhere in a sequence of grammar symbols) is a derivation step. The second states that the derivation relation is reflexive, i.e., that a sequence derives itself. The third rule describes transitivity, i.e., that a sequence of derivations is in itself a derivation.[2] Some texts about grammars use the symbol $\to^*$ instead of $\Rightarrow$ for derivation.

We can use derivation to formally define the language that a context-free grammar generates:

[2] The mathematically inclined will recognise that derivation is a preorder on sequences of grammar symbols.

Grammar 2.4 Example grammar

$$
\begin{aligned}
T &\to R \\
T &\to \mathtt{a}T\mathtt{c} \\
R &\to \\
R &\to R\mathtt{b}R
\end{aligned}
$$

Fig. 2.5 Derivation of the sequence aabbbcc using Grammar 2.4

$$
\begin{aligned}
& \underline{T} && \\
\Rightarrow\ & \mathtt{a}\underline{T}\mathtt{c} && \text{using the production } T \to \mathtt{a}T\mathtt{c} \\
\Rightarrow\ & \mathtt{aa}\underline{T}\mathtt{cc} && \text{using the production } T \to \mathtt{a}T\mathtt{c} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{cc} && \text{using the production } T \to R \\
\Rightarrow\ & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{cc} && \text{using the production } R \to R\mathtt{b}R \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{bcc} && \text{using the production } R \to \\
\Rightarrow\ & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{bcc} && \text{using the production } R \to R\mathtt{b}R \\
\Rightarrow\ & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{b}R\mathtt{bcc} && \text{using the production } R \to R\mathtt{b}R \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{bb}R\mathtt{bcc} && \text{using the production } R \to \\
\Rightarrow\ & \mathtt{aabb}\underline{R}\mathtt{bcc} && \text{using the production } R \to \\
\Rightarrow\ & \mathtt{aabbbcc} && \text{using the production } R \to
\end{aligned}
$$

Fig. 2.6 Leftmost derivation of the sequence aabbbcc using Grammar 2.4

$$
\begin{aligned}
& \underline{T} && \\
\Rightarrow\ & \mathtt{a}\underline{T}\mathtt{c} && \text{using the production } T \to \mathtt{a}T\mathtt{c} \\
\Rightarrow\ & \mathtt{aa}\underline{T}\mathtt{cc} && \text{using the production } T \to \mathtt{a}T\mathtt{c} \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{cc} && \text{using the production } T \to R \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{cc} && \text{using the production } R \to R\mathtt{b}R \\
\Rightarrow\ & \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc} && \text{using the production } R \to R\mathtt{b}R \\
\Rightarrow\ & \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{cc} && \text{using the production } R \to \\
\Rightarrow\ & \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc} && \text{using the production } R \to R\mathtt{b}R \\
\Rightarrow\ & \mathtt{aabb}\underline{R}\mathtt{b}R\mathtt{cc} && \text{using the production } R \to \\
\Rightarrow\ & \mathtt{aabbb}\underline{R}\mathtt{cc} && \text{using the production } R \to \\
\Rightarrow\ & \mathtt{aabbbcc} && \text{using the production } R \to
\end{aligned}
$$

Definition 2.1 Given a context-free grammar G with start symbol S, terminal symbols T and productions P, the language $L(G)$ that G generates is defined to be the set of sequences of terminal symbols that can be obtained by derivation from S using the productions P, i.e., the set $\{w \in T^* \mid S \Rightarrow w\}$.

As an example, we see that Grammar 2.4 generates the sequence aabbbcc by the derivation shown in Fig. 2.5. We have, for clarity, in each sequence of symbols underlined the nonterminal that will be rewritten in the following step.

In this derivation, we have applied derivation steps sometimes to the leftmost nonterminal, sometimes to the rightmost, and sometimes to a nonterminal that was neither. However, since derivation steps are local, the order does not matter. So, we might as well decide to always rewrite the leftmost nonterminal, as shown in Fig. 2.6.

A derivation that always rewrites the leftmost nonterminal is called a *leftmost derivation*. Similarly, a derivation that always rewrites the rightmost nonterminal is called a *rightmost derivation*.

2.2.1 Syntax Trees and Ambiguity

We can draw a derivation as a tree: The root of the tree is the start symbol of the grammar, and whenever we rewrite a nonterminal, we add as its children the symbols on the right-hand side of the production that was used. The leaves of the tree are terminals which, when read from left to right (i.e., following children from left to right), form the derived sequence. If a nonterminal is rewritten using an empty production, the child is shown as an empty leaf node, which is ignored when reading the sequence from the leaves of the tree. Some variants of syntax trees show empty leaf node as ε.

When we write such a *syntax tree*, the order of derivation is irrelevant: We get the same tree for left derivation, right derivation or any other derivation order. Only the choice of production for rewriting each nonterminal matters.

As an example, the derivations in Figs. 2.5 and 2.6 yield the same syntax tree, which is shown in Fig. 2.7.

The syntax tree adds structure to the string (sequence of terminal symbols) that it derives. It is this structure that we exploit in the later phases of the compiler.

For compilation, derivation is done backwards: We start with a string and want to produce a syntax tree. This process is called *syntax analysis* or *parsing*.

Even though the *order* of derivation does not matter when constructing a syntax tree, the *choice* of production for a nonterminal does. Obviously, different choices can lead to different strings being derived, but it may also happen that several different syntax trees can be built for the same string. As an example, Fig. 2.8 shows an alternative syntax tree for the same string that was derived in Fig. 2.7.

When a grammar permits several different syntax trees for some strings, we call the grammar *ambiguous*. If our only use of grammar is to describe sets of strings,

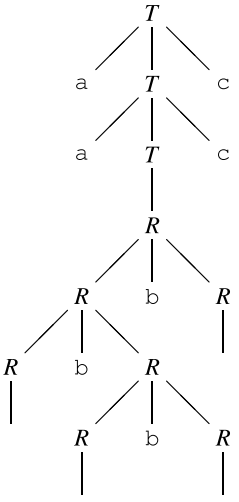


Fig. 2.7 Syntax tree for the string `aabbbcc` using Grammar 2.4

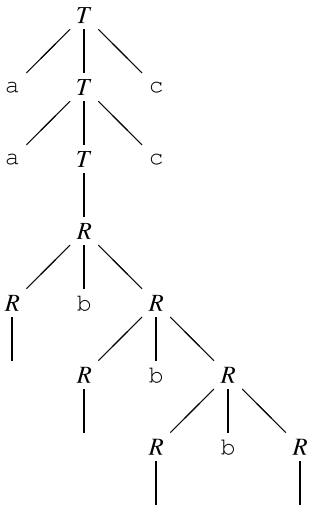


Fig. 2.8 Alternative syntax tree for the string aabbbcc using Grammar 2.4

Grammar 2.9 Unambiguous version of Grammar 2.4

$T \to R$
$T \to \texttt{a}T\texttt{c}$
$R \to$
$R \to \texttt{b}R$

ambiguity is not a problem. However, when we want to use the grammar to impose structure on strings, the structure had better be the same every time. Hence, it is a desirable feature for a grammar to be unambiguous. In most (but not all) cases, an ambiguous grammar can be rewritten to an unambiguous grammar that generates the same set of strings. An unambiguous version of Grammar 2.4 is shown in Grammar 2.9.

An alternative to rewriting an ambiguous grammar to an unambiguous grammar is to apply external rules (not expressed in the grammar) for choosing productions when several are possible. We will return to this in Sect. 2.15.

How do we know when a grammar is ambiguous? Proving ambiguity is conceptually simple: If we can find a string and show two alternative syntax trees for it, the grammar is ambiguous. It may, however, be hard to find such a string and, when the grammar is unambiguous, even harder to show that there are no strings with more than one syntax tree. In fact, the problem is formally undecidable, i.e., there is no method that for all grammars can answer the question "Is this grammar ambiguous?". But in many cases, it is not difficult to detect and prove ambiguity. For example, if a grammar has productions of the form

$$N \to N\alpha N$$
$$N \to \beta$$

where α and β are arbitrary (possibly empty) sequences of grammar symbols, the grammar is ambiguous. This is, for example, the case with Grammars 2.2 and 2.4.

We will, in Sects. 2.11 and 2.13, see methods for constructing parsers from grammars. These methods have the property that they only work on unambiguous grammars, so successful construction of a parser is a proof of unambiguity. However, the methods may for some unambiguous grammars fail to produce parsers, so failure to produce a parser is not a proof of ambiguity.

In the next section, we will see ways of rewriting a grammar to get rid of some sources of ambiguity. These transformations preserve the language that the grammar generates at the cost of changing the syntax trees derived from the grammar. By using such transformations (and others, which we will see later), we can create a large set of *equivalent* grammars, i.e., grammars that generate the same language.

Given two grammars, it would be nice to be able to tell if they are equivalent. Unfortunately, like ambiguity, equivalence of context-free grammars is undecidable. Sometimes, equivalence can be proven e.g., by induction over the set of strings that the grammars produce. The converse (i.e., non-equivalence) can be proven by finding an example of a string that one grammar can generate, but the other not. But in some cases, we just have to take claims of equivalence on faith or give up on deciding the issue.

Different, but equivalent, grammars will impose different syntax trees on the strings of their common language, so for compilers they are not equally useful—we want a grammar that imposes the intended structure on programs. Different structure is not exactly the same as different syntax trees: There may be several different grammars that impose the (for some intended purpose) correct structure, even if they do not yield the same syntax trees. We define when two different syntax trees represent the same structure by *reducing* syntax trees: If a node in the syntax tree has only one child, we replace the node by its child (which may be empty). A syntax tree that has no nodes with only one child is *fully reduced*. We deem two syntax trees to represent the same structure if their fully reduced trees are identical except for the names of nonterminals that represent the same syntactic category. Note that a reduced tree is not always a proper syntax tree: The edges do not represent single derivation steps, but rather sequences of derivation steps. Figure 2.10 shows a fully reduced version of the syntax tree in Fig. 2.7.

Suggested exercises: 2.1, 2.2, 2.21(b).

2.3 Operator Precedence

As mentioned in Sect. 2.1.1, we can describe traditional arithmetic expressions by Grammar 2.2. Note that **num** is a terminal that denotes all integer constants and that, here, the parentheses are terminal symbols (unlike in regular expressions, where they are used to impose structure on the regular expressions).

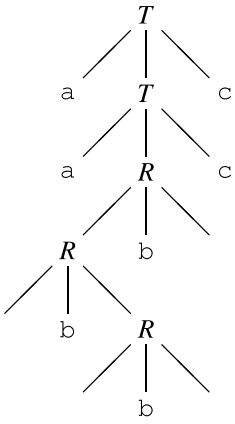


Fig. 2.10 Fully reduced tree for the syntax tree in Fig. 2.7

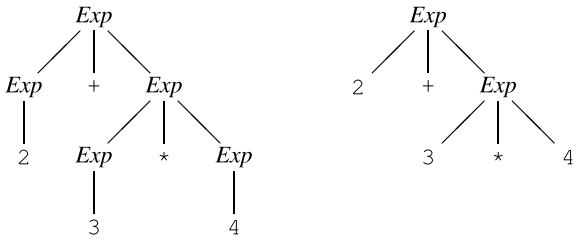


Fig. 2.11 Preferred syntax tree for `2+3*4` using Grammar 2.2, and the corresponding fully reduced tree

This grammar is ambiguous, as evidenced by, e.g., the productions

$$Exp \rightarrow Exp + Exp$$
$$Exp \rightarrow \textbf{num}$$

which have the form that in Sect. 2.2.1 was claimed to imply ambiguity. That expressions are ambiguous should not be surprising, as we are used to the fact that an expression like `2+3*4` can be read in two ways: Either as multiplying the sum of 2 and 3 by 4 or as adding 2 to the product of 3 and 4. Simple electronic calculators will choose the first of these interpretations (as they always calculate from left to right), whereas scientific calculators and most programming languages will choose the second, as they use a hierarchy of *operator precedences* which dictate that the product must be calculated before the sum. The hierarchy can be overridden by explicit parenthesisation, e.g., `(2+3)*4`.

Most programming languages use the same convention as scientific calculators, so we want to make this explicit in the grammar. Ideally, we would like the expression `2+3*4` to generate the syntax tree shown on the left in Fig. 2.11, which reflects the operator precedences by grouping of subexpressions: When evaluating an expression, the subexpressions represented by subtrees of the syntax tree are evaluated before the topmost operator is applied. A corresponding fully reduced tree is shown on the right in Fig. 2.11.

A possible way of resolving the ambiguity is during syntax analysis to use precedence rules (not stated in the grammar itself) to select among the possible syntax trees. Many parser generators allow this approach, as we shall see in Sect. 2.15. However, some parsing methods require the grammars to be unambiguous, so we have to express the operator hierarchy in the grammar itself.

We first define some concepts relating to infix operators:

- An operator $\oplus$ is *left-associative* if the expression $a \oplus b \oplus c$ must be evaluated from left to right, i.e., as $(a \oplus b) \oplus c$.
- An operator $\oplus$ is *right-associative* if the expression $a \oplus b \oplus c$ must be evaluated from right to left, i.e., as $a \oplus (b \oplus c)$.
- An operator $\oplus$ is *non-associative* if expressions of the form $a \oplus b \oplus c$ are illegal.

By the usual convention, `-` and `/` are left-associative, as e.g., `2-3-4` is calculated as `(2-3)-4`. `+` and `*` are associative in the mathematical sense, meaning that it does not matter if we calculate from left to right or from right to left. In programming languages, it can matter, as one order of addition may cause overflow, where the other does not, or one order may cause more loss of precision than another. Generally, we want to avoid ambiguity of expressions, even when they are mathematically equivalent, so we choose either left-associativity or right-associativity even for operators that mathematically are fully associative. By convention (and similarity to `-` and `/`) we choose to let addition and multiplication be left-associative. Also, having a left-associative `-` and right-associative `+` would not help resolving the ambiguity of `2-3+4`, as the operators so-to-speak "pull in different directions".

List construction operators in functional languages, e.g., `::` and `@` in SML, are typically right-associative, as are function arrows in types: `a -> b -> c` is read as `a -> (b -> c)`. The assignment operator in C is also right-associative: `a = b = c` is read as `a = (b = c)`.

In some languages (such as Pascal), comparison operators (such as $<$ and $>$) are non-associative, i.e., you are not allowed to write `2 < 3 < 4`.

2.3.1 Rewriting Ambiguous Expression Grammars

If we have an ambiguous grammar

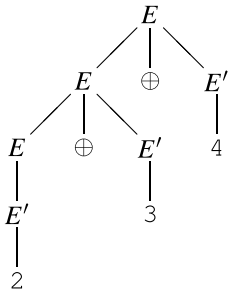
$$E \to E \oplus E$$
$$E \to \textbf{num}$$

and an intended structure on expressions, we can rewrite the ambiguous grammar to an unambiguous grammar that generates the correct structure. As a structure requires a specific associativity of $\oplus$, we use different rewrite rules for different associativities.

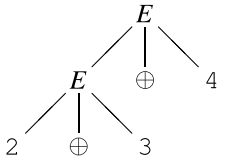
If $\oplus$ is left-associative, we make the grammar *left-recursive* by having a recursive reference to the left only of the operator symbol and replacing the right-recursive reference by a reference to a new nonterminal that represents the non-recursive cases:

$$
\begin{aligned}
E &\to E \oplus E' \\
E &\to E' \\
E' &\to \textbf{num}
\end{aligned}
$$

Now, the expression $2 \oplus 3 \oplus 4$ can only be parsed as



We get a slightly more complex syntax tree than ideally: We would have liked to avoid having two different nonterminals for expressions, and we would prefer to avoid the derivation $E \to E'$, but the tree certainly reflects the structure that the leftmost application of $\oplus$ binds more tightly than the rightmost application of $\oplus$. The corresponding fully reduced tree makes this more clear:



We handle right-associativity in a similar fashion: We make the offending production *right-recursive*:

$$
\begin{aligned}
E &\to E' \oplus E \\
E &\to E' \\
E' &\to \textbf{num}
\end{aligned}
$$

Non-associative operators are handled by *non-recursive* productions:

$$
\begin{aligned}
E &\to E' \oplus E' \\
E &\to E' \\
E' &\to \textbf{num}
\end{aligned}
$$

Note that the latter transformation actually changes the language that the grammar generates, as it makes expressions of the form **num** $\oplus$ **num** $\oplus$ **num** illegal.

So far, we have handled only cases where an operator interacts with itself. This is easily extended to the case where several operators with the same precedence and associativity interact with each other, as for example + and −:

$$
\begin{aligned}
E &\rightarrow E + E' \\
E &\rightarrow E - E' \\
E &\rightarrow E' \\
E' &\rightarrow \textbf{num}
\end{aligned}
$$

Operators with the same precedence must have the same associativity for this to work, as mixing left-recursive and right-recursive productions for the same nonterminal makes the grammar ambiguous. As an example, the grammar

$$
\begin{aligned}
E &\rightarrow E + E' \\
E &\rightarrow E' \oplus E \\
E &\rightarrow E' \\
E' &\rightarrow \textbf{num}
\end{aligned}
$$

seems like an obvious generalisation of the principles used above, giving + and $\oplus$ the same precedence and different associativity. But not only is the grammar ambiguous, it does not even accept the intended language. For example, the string **num**+**num**$\oplus$**num** is not derivable by this grammar.

In general, there is no obvious way to resolve ambiguity in an expression like `1+2⊕3`, where + is left-associative and $\oplus$ is right-associative (or *vice-versa*). Hence, most programming languages (and most parser generators) *require* operators at the same precedence level to have identical associativity.

We also need to handle operators with different precedences. This is done by using a nonterminal for each precedence level. The idea is that if an expression uses an operator of a certain precedence level, then its subexpressions cannot use operators of lower precedence (unless these are inside parentheses). Hence, the productions for a nonterminal corresponding to a particular precedence level refers only to nonterminals that correspond to the same or higher precedence levels, unless parentheses or similar bracketing constructs disambiguate the use of these. Grammar 2.12 shows how these rules are used to make an unambiguous version of Grammar 2.2. Figure 2.13 shows the syntax tree for `2+3*4` using this grammar and the corresponding reduced tree. Note that the nonterminals *Exp*, *Exp*2, and *Exp*3 all represent the same syntactic category (expressions), so in the reduced tree we can equate *Exp* and *Exp*2, making it equivalent to the reduced tree in Fig. 2.11.

Suggested exercises: 2.6.

Grammar 2.12
Unambiguous expression grammar

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp2 \\
Exp &\rightarrow Exp - Exp2 \\
Exp &\rightarrow Exp2 \\
Exp2 &\rightarrow Exp2 * Exp3 \\
Exp2 &\rightarrow Exp2 \,/\, Exp3 \\
Exp2 &\rightarrow Exp3 \\
Exp3 &\rightarrow \textbf{num} \\
Exp3 &\rightarrow (\ Exp\ )
\end{aligned}
$$

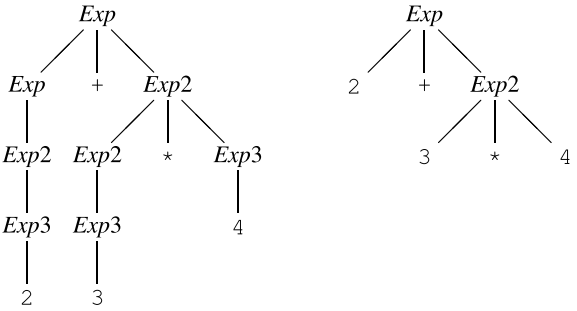


Fig. 2.13 Syntax tree for `2+3*4` using Grammar 2.12, and the corresponding fully reduced tree

2.4 Other Sources of Ambiguity

Most of the potential ambiguity in grammars for programming languages comes from expression syntax and can be handled by exploiting precedence rules as shown in Sect. 2.3. Another classical example of ambiguity is the "dangling-else" problem.

Imperative languages like Pascal or C often let the else-part of a conditional be optional, like shown in Grammar 2.3. The problem is that it is not clear how to parse, for example,

```
if p then if q then x := 1 else x := 2
```

According to the grammar, the `else` can equally well match either `if`. The usual convention used in programming languages is that an `else` matches the closest not previously matched `if`, which, in the example, will make the `else` match the second `if`.

How do we make this clear in the grammar? We can treat `if`, `then` and `else` in the same way as right-associative operators, as this will make them group to the right, making an `if-then` match the closest `else`. However, the grammar transformations shown in Sect. 2.3 can not directly be applied to Grammar 2.3, as the productions for conditionals do not have the right form.

Instead we use the following observation: When an `if` and an `else` match, all `if`s that occur between these must have matching `else`s. This can easily be proven by assuming otherwise and concluding that this leads to a contradiction.

Hence, we make two nonterminals: One for matched (i.e. with `else`-part) conditionals and one for unmatched (i.e. without `else`-part) conditionals. The result is shown in Grammar 2.14. This grammar also resolves the associativity of semicolon (right) and the precedence of `if` over semicolon.

An alternative to rewriting grammars to resolve ambiguity is to use an ambiguous grammar and resolve conflicts by using external precedence rules during parsing. We shall look into this in Sect. 2.15.

All cases of ambiguity must be treated carefully: It is not enough that we eliminate ambiguity, we must do so in a way that results in the desired structure: The structure

Grammar 2.14
Unambiguous grammar for statements

Stat → *Stat*2 **;** *Stat*
Stat → *Stat*2
*Stat*2 → *Matched*
*Stat*2 → *Unmatched*
Matched → `if` *Exp* `then` *Matched* `else` *Matched*
Matched → **id** **:=** *Exp*
Unmatched → `if` *Exp* `then` *Matched* `else` *Unmatched*
Unmatched → `if` *Exp* `then` *Stat*2

of arithmetic expressions is significant, and it makes a difference to which `if` an `else` is matched.

Suggested exercises: 2.3 (focusing now on making the grammar unambiguous).

2.5 Syntax Analysis

The syntax analysis phase of a compiler will take a string of tokens produced by the lexer, and from this construct a syntax tree for the string by finding a derivation of the string from the start symbol of the grammar.

This can be done by guessing derivations (i.e., choosing productions randomly) until the right one is found, but random guessing is hardly an effective method. Even so, some parsing techniques are based on "guessing" derivations. However, these make sure, by looking at the string, that they will always pick the right production. These are called *predictive* parsing methods. Predictive parsers always build the syntax tree from the root down to the leaves and are hence also called (deterministic) top-down parsers.

Other parsers go the other way: They search for parts of the input string that matches right-hand sides of productions and rewrite these to the left-hand nonterminals, at the same time building pieces of the syntax tree. The syntax tree is eventually completed when the string has been rewritten (by inverse derivation) to the start symbol. Also here, we wish to make sure that we always pick the "right" rewrites, so we get deterministic parsing. Such methods are called *bottom-up* parsing methods.

We will in the next sections first look at predictive parsing and later at a bottom-up parsing method called SLR parsing.

2.6 Predictive Parsing

If we look at the left-derivation in Fig. 2.6, we see that, when we replace a nonterminal by the right-hand side of a production, all symbols to the left of this nonterminal are terminals. In other words, we always rewrite the leftmost nonterminal. The terminals to the left of this nonterminal correspond to a prefix of the string that is being parsed.

In a parsing situation, this prefix will be the part of the input that has already been read. The job of the parser is now to choose the production by which the leftmost unexpanded nonterminal should be rewritten. Our aim is to be able to make this choice deterministically based only on the next unmatched input symbol.

If we look at the third line in Fig. 2.6, we have already read two `a`s and (if the input string is the one shown in the bottom line) the next symbol is a `b`. Since the right-hand side of the production

$$T \to \texttt{a}T\texttt{c}$$

starts with an `a`, we obviously can not use this. Hence, we can only rewrite T using the production

$$T \to R$$

We are not quite as lucky in the next step. None of the productions for R start with a terminal symbol, so we can not immediately choose a production based on this. As the grammar (Grammar 2.4) is ambiguous, it should not be a surprise that we can not always choose uniquely. If we, instead, use the unambiguous grammar (Grammar 2.9) we can, when the next input symbol is a `b`, immediately choose the second production for R. When all the `b`s are read and we are at the following `c`, we choose the empty production for R and match the remaining input with the rest of the derived string.

If we can always choose a unique production based only on the next input symbol, we are able to do predictive parsing without backtracking. We will, below, investigate when we are able to make such unique choices.

2.7 *Nullable* and *FIRST*

In simple cases, like the above, the right-hand sides of all productions for any given nonterminal start with distinct terminals, except at most one production whose right-hand side does not start with a terminal (i.e., it is an empty production, or the right-hand side of the production starts with a nonterminal). We chose the production whose right-hand side does not start with a terminal whenever the input symbol does not match any of the terminal symbols that start the right-hand sides other productions. In the example above (using (Grammar 2.9)), we choose the second production for T as long as the next input symbol is `a`, and the first production otherwise. Similarly, we choose the second production for R when the next input symbol is `b`, and the first production otherwise.

We can extend the method to work also for grammars where more than one production for a given nonterminal have right-hand sides that do not start with terminals. We just need to be able to select between these productions based on the input symbol, even when the right-hand sides do not start with terminal symbols. To do this, we for each right-hand side find the set of strings that the right-hand side can derive.

We then, for each production, find the set of initial characters of the strings in these sets. These sets are called the *FIRST* sets of the productions.

If the *FIRST* sets of different productions for the same nonterminal are disjoint, we can, given an input symbol, choose the production whose *FIRST* set contains this symbol. If the input symbol is not in any of the *FIRST* sets, and there is an empty production, we choose this. Otherwise, we report a syntax error message.

Hence, we define the function *FIRST*, which given a sequence of grammar symbols (e.g., the right-hand side of a production) returns the set of symbols with which strings derived from that sequence can begin:

Definition 2.2 A symbol c is in $FIRST(\alpha)$ if and only if $\alpha \Rightarrow c\beta$ for some (possibly empty) sequence β of grammar symbols.

We extend this definitions to productions, so $FIRST(N \rightarrow \alpha) = FIRST(\alpha)$, and to nonterminals by defining $FIRST(N)$ to be the union of the first sets for all productions for N.

To calculate *FIRST*, we need an auxiliary function *Nullable*, which for a sequence α of grammar symbols indicates whether or not that sequence can derive the empty string:

Definition 2.3 A sequence α of grammar symbols is *Nullable* (we write this as $Nullable(\alpha)$) if and only if $\alpha \Rightarrow \varepsilon$, where ε indicates the empty string.

A production $N \rightarrow \alpha$ is called nullable if $Nullable(\alpha)$.

We describe calculation of *Nullable* by case analysis over the possible forms of sequences of grammar symbols:

Algorithm 2.4

$$
\begin{array}{ll}
Nullable() & = true \\
Nullable(\mathtt{a}) & = false \\
Nullable(\alpha\ \beta) & = Nullable(\alpha) \wedge Nullable(\beta) \\
Nullable(N) & = Nullable(\alpha_1) \vee \ldots \vee Nullable(\alpha_n), \\
 & \quad \textit{where the productions for } N \textit{ are} \\
 & \quad N \rightarrow \alpha_1, \ldots, N \rightarrow \alpha_n
\end{array}
$$

where $\mathtt{a}$ *is a terminal,* N *is a nonterminal,* α *and* β *are sequences of grammar symbols. Note that the first rule handles the empty sequence of grammar symbols.*

The equations are quite natural: Any occurrence of a terminal on a right-hand side makes *Nullable* false for that right-hand side, and a nonterminal is nullable if any production has a nullable right-hand side.

Note that this is a recursive definition since *Nullable* for a nonterminal is defined in terms of *Nullable* for its right-hand sides, some of which may contain that same nonterminal. We can solve this in much the same way that we solved set equations in Sect. 1.5.1. We have, however, now booleans instead of sets, and we have several

equations instead of one. Still, the method is essentially the same: We have a set of Boolean equations:

$$\begin{aligned} X_1 &= F_1(X_1, \ldots, X_n) \\ &\vdots \\ X_n &= F_n(X_1, \ldots, X_n) \end{aligned}$$

We initially assume $X_1, \ldots, X_n$ to be all *false*. We then, in any order, calculate the right-hand sides of the equations and update the variable on the left-hand side by the calculated value. We continue until all right-hand sides evaluate to the values on the corresponding left-hand sides. This implies that the equations are solved.

In the Appendix and Sect. 1.5.1, we required the functions to be monotonic with respect to subset. Correspondingly, we now require the Boolean functions to be monotonic with respect to truth: If we make more arguments of a function true, the result will also be more true (i.e., it may stay unchanged or change from *false* to *true*, but never change from *true* to *false*). We also have a property similar to the distributivity property we observed in Sect. 1.5.1: $F_i(X_1, \ldots, (p \vee q), \ldots, X_n) = F_i(X_1, \ldots, p, \ldots, X_n) \vee F_i(X_1, \ldots, q, \ldots, X_n)$ for any i and j. We can also observe that if we represent *false* by the empty set and *true* by the set $\{1\}$, $\vee$ is set union and $\wedge$ is set intersection, and we can solve the equations as set equations.

If we look at Grammar 2.9, we get the following equations by applying the rules for *Nullable* on the right-hand of the productions, and reducing the results using simple logical identities:

$$\begin{aligned} \mathit{Nullable}(T) &= \mathit{Nullable}(R) \vee \mathit{Nullable}(\texttt{a}T\texttt{c}) \\ &= \mathit{Nullable}(R) \vee (\mathit{Nullable}(\texttt{a}) \wedge \mathit{Nullable}(T) \wedge \mathit{Nullable}(\texttt{c})) \\ &= \mathit{Nullable}(R) \vee (\mathit{false} \wedge \mathit{Nullable}(T) \wedge \mathit{false}) \\ &= \mathit{Nullable}(R) \\ \mathit{Nullable}(R) &= \mathit{Nullable}() \vee \mathit{Nullable}(\texttt{b}R) \\ &= \mathit{true} \vee (\mathit{Nullable}(\texttt{b}) \wedge \mathit{Nullable}(R)) \\ &= \mathit{true} \end{aligned}$$

In a fixed-point calculation, we initially assume that *Nullable* is false for all nonterminals, and use this as a basis for calculating the right-hand sides of the equations. We repeat recalculating these until there is no change between two iterations. Figure 2.15 shows the fixed-point iteration for the above equations. In each iteration, we evaluate the right-hand sides of the equations using the values from the previous iteration. We continue until two iterations yield the same results. The right-most column shows the final result. Fixed-point iteration is a bit of overkill for this simple example, but it is needed in the general case.

Fig. 2.15 Fixed-point iteration for calculation of *Nullable*

Nonterminal	Initialisation	Iteration 1	Iteration 2	Iteration 3
T	*false*	*false*	*true*	*true*
R	*false*	*true*	*true*	*true*

To choose productions for predictive parsing, we need to know if the individual productions are nullable. When we know which nonterminals are nullable, it is easy enough to calculate this for the right-hand side of each production using the formulae in Algorithm 2.4. For Grammar 2.9 we get:

Production	*Nullable*
$T \to R$	*true*
$T \to \mathtt{a}T\mathtt{c}$	*false*
$R \to$	*true*
$R \to \mathtt{b}R$	*false*

We can calculate *FIRST* in a fashion similar to the calculation of *Nullable*, i.e., by using formulas for sequences of grammar symbols and recursively defining equations for the nonterminals:

Algorithm 2.5

$$
\begin{aligned}
FIRST(\,) &= \emptyset \\
FIRST(\mathtt{a}) &= \{\mathtt{a}\} \\
FIRST(\alpha\,\beta) &= \begin{cases} FIRST(\alpha) \cup FIRST(\beta) & \textit{if } Nullable(\alpha) \\ FIRST(\alpha) & \textit{if not } Nullable(\alpha) \end{cases} \\
FIRST(N) &= FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n) \\
&\quad \textit{where the productions for } N \textit{ are} \\
&\quad N \to \alpha_1, \ldots, N \to \alpha_n
\end{aligned}
$$

where $\mathtt{a}$ *is a terminal,* N *is a nonterminal, and* α *and* β *are sequences of grammar symbols.* $\emptyset$ *denotes the empty set.*

The only nontrivial equation is that for $\alpha\beta$. Obviously, anything that can start a string derivable from α can also start a string derivable from $\alpha\beta$. However, if α is nullable, a derivation may proceed as $\alpha\beta \Rightarrow \beta \Rightarrow \cdots$, so if α is nullable, anything in $FIRST(\beta)$ is also in $FIRST(\alpha\beta)$. So we have special cases for when α is nullable, and when it is not.

The set-equations are solved in the same general way as the Boolean equations for *Nullable*, but since we work with sets, we initially assume every set to be empty. For Grammar 2.9, we get the following equations:

$$
\begin{aligned}
FIRST(T) &= FIRST(R) \cup FIRST(\mathtt{a}T\mathtt{c}) \\
&= FIRST(R) \cup FIRST(\mathtt{a}) \\
&= FIRST(R) \cup \{\mathtt{a}\} \\
FIRST(R) &= FIRST(\,) \cup FIRST(\mathtt{b}R) \\
&= \emptyset \cup FIRST(\mathtt{b}) \\
&= \{\mathtt{b}\}
\end{aligned}
$$

The fixed-point iteration is shown in Fig. 2.16. As before, we use the values from the previous iteration when calculating the right-hand sides of the equations.

Fig. 2.16 Fixed-point iteration for calculation of *FIRST*

Nonterminal	Initialisation	Iteration 1	Iteration 2	Iteration 3
T	$\emptyset$	{a}	{a, b}	{a, b}
R	$\emptyset$	{b}	{b}	{b}

As for *Nullable*, we need *FIRST* for every production, which we can find by using the formulae in Algorithm 2.5 and the values of *FIRST* for the nonterminals:

Production	*FIRST*
$T \to R$	{b}
$T \to \mathtt{a}T\mathtt{c}$	{a}
$R \to$	$\emptyset$
$R \to \mathtt{b}R$	{b}

We note that the two productions for T have disjoint *FIRST* sets, so we can uniquely choose a production based on the input symbol. Since the first production for T is nullable, we choose this also on symbols other than b, in fact we choose it on all other symbols than a, where we choose the second production. The productions for R also have disjoint *FIRST* sets. We choose the empty production for R when the input symbol is not b.

When working with grammars by hand, it is usually quite easy to see for most productions if they are nullable and what their *FIRST* sets are. For example, a production is not nullable if its right-hand side has a terminal anywhere, and if the right-hand side starts with a terminal, the *FIRST* set consists of only that symbol. Sometimes, however, it is necessary to use fixed-point iteration to solve the equations.

Suggested exercises: 2.8 (*Nullable* and *FIRST* only).

2.8 Predictive Parsing Revisited

We have up to now used the following rule for predictive parsing: If the right-hand sides of the productions for a nonterminal have disjoint *FIRST* sets, and the next input symbol is in one of these sets, we choose the corresponding production. If the next input symbol is not in any of these sets, and there is an empty production, we choose this.

We can generalise the case for the empty production, so we in the case where the next input symbol is not found in any *FIRST* set, can select a production if it is *Nullable*. The idea is that a *Nullable* production can derive the empty string, so we can extend the rule for empty productions to cover nullable productions as well. Note that a nullable production can have a non-empty *FIRST* set, so it can be chosen both when the next input symbol is in its *FIRST* set, and when the next input symbol is not in the *FIRST* set of the nonterminal (i.e., not in the *FIRST* set of any of the productions for the nonterminal).

But if there are several *Nullable* productions, we have no way of choosing between them. So, for predictive parsing, a nonterminal can have at most one *Nullable* production.

We said in Sect. 2.2.1 that our syntax analysis methods will detect ambiguous grammars. However, this is not true with the method as stated above: We can get unique choice of production even for some ambiguous grammars, including Grammar 2.4. In the best case, the syntax analysis will just choose one of several possible syntax trees for a given input string. In many cases, we do not consider such behaviour acceptable. In fact, we would very much like our parser construction method to tell us if we by mistake write an ambiguous grammar.

Even worse, the rules for predictive parsing as presented here might—even for unambiguous grammars—give deterministic choice of production, but reject strings that actually belong to the language described by the grammar. If we, for example, change the second production in Grammar 2.9 to

$$T \to \texttt{a}T\texttt{b}$$

this will not change the choices made by the predictive parser for nonterminal R. However, always choosing the last production for R on a `b` will lead to erroneous rejection of many strings, including `ab`.

This kind of behaviour is clearly unacceptable. We should, at least, get a warning that this might occur, so we can rewrite the grammar or choose another syntax analysis method.

Hence, we add to our construction of predictive parsers a test that will reject all ambiguous grammars and those unambiguous grammars that can cause the parser to fail even though a valid parse exists. The test can not tell us in which of these two categories a grammar belongs, though, but in either case we need to rewrite the grammar (or choose a different parsing method).

We have, so far, simply chosen a nullable production if and only if the next input symbol is not in the *FIRST* set of the nonterminal, i.e., if no other choice is valid. But this does not imply that choosing the nullable production is always valid when no other choice is valid. It could well be the case that no choice is valid—which implies that the string we are parsing is not in the language of the grammar. The right thing to do in such cases is to issue an error.

So we must change the rules for choosing productions in such a way that we choose a nullable production only if this is meaningful. So we choose a production $N \to \alpha$ on symbol c if at least one of the two conditions below are satisfied:

1) $c \in \mathit{FIRST}(\alpha)$, or
2) α is nullable, and the sequence Nc can occur somewhere in a derivation starting from the start symbol of the grammar.

The first rule is obvious, but the second requires a bit of explanation: If α is nullable, we can construct a syntax tree for N without reading any input, so it seems like a nullable production could be a valid choice regardless of the next input symbol.

Predictive parsing makes a leftmost derivation, so we always rewrite the leftmost nonterminal N in the current sequence of grammar symbols. If we look at the part of the current sequence of grammar symbols that start with this N, it has the form $N\beta$, where β is any (possibly empty) sequence of grammar symbols. If the next input symbol is c, it must be in $FIRST(N\beta)$, otherwise we can never derive $N\beta$ to a string that starts with c. If c is not in $FIRST(N)$, then N must be nullable and c must be in $FIRST(\beta)$. But β is not necessarily the right-hand side of any production, so we will need to find $FIRST(\beta)$ in some other way. The next section will cover this.

Even with this restriction on choosing nullable productions, we can still have situations where both nullable and non-nullable productions are valid choices. This includes the example above with the modified Grammar 2.9 (since $R\flat$ can occur in a derivation). An ambiguous grammar will have either:

1. two or more nullable productions for a given nonterminal, or
2. overlapping *FIRST* sets for the productions of a nonterminal, or
3. a *FIRST* set for a non-nullable production that overlaps with the set of characters that makes a nullable production for the same nonterminal a valid choice.

Note that while *absence* of such conflicts proves that a grammar is unambiguous, *presence* of such conflicts does not prove that a grammar is ambiguous.

2.9 Follow

To determine when we can select a nullable production during predictive parsing, we introduce *FOLLOW* sets for nonterminals.

Definition 2.6 A terminal symbol c is in $FOLLOW(N)$ if and only if there is a derivation from the start symbol S of the grammar such that $S \Rightarrow \alpha N c \beta$, where α and β are (possibly empty) sequences of grammar symbols.

In other words, a terminal c is in $FOLLOW(N)$ if c may follow N at some point in a derivation. Unlike $FIRST(N)$, this is not a property of the productions for N, but of the productions that (directly or indirectly) use N on their right-hand side.

To correctly handle end-of-string conditions, we also want to detect when $S \Rightarrow \alpha N$, i.e., if there are derivations where N can be followed by the end of input. It turns out to be easiest to do this by adding an extra production to the grammar:

$$S' \rightarrow S\$$$

where S' is a new nonterminal that replaces S as start symbol, and \$ is a new terminal symbol that represents the end of input. Hence, in the new grammar, \$ will be in $FOLLOW(N)$ exactly if $S' \Rightarrow \alpha N\$$, which is the case exactly when $S \Rightarrow \alpha N$.

The easiest way to calculate *FOLLOW* is to generate a collection of *set constraints*, which are subsequently solved to find the smallest sets that obey the constraints. A production

$$M \to \alpha N \beta$$

generates the constraint $FIRST(\beta) \subseteq FOLLOW(N)$, since β, obviously, can follow N. Furthermore, if $Nullable(\beta)$ the production also generates the constraint $FOLLOW(M) \subseteq FOLLOW(N)$ (note the direction of the inclusion). The reason is that, if there is a derivation $S' \Rightarrow \gamma M \delta$, then because $M \to \alpha N \beta$, and β is nullable, we derive $S' \Rightarrow \gamma M \delta \Rightarrow \gamma \alpha N \beta \delta \Rightarrow \gamma \alpha N \delta$, so $FIRST(\delta)$ is also in $FOLLOW(N)$. This is true for any such δ, so $FOLLOW(M) \subseteq FOLLOW(N)$.

If a right-hand side contains several occurrences of nonterminals, we add constraints for all occurrences, i.e., splitting the right-hand side with different choices of α, N and β. For example, the production $A \to BcB$ generates the constraint $\{c\} \subseteq FOLLOW(B)$ by splitting after the first B, and, by splitting after the last B, we also get the constraint $FOLLOW(A) \subseteq FOLLOW(B)$.

We solve the generated constraints in the following fashion:

We start by assuming empty *FOLLOW* sets for all nonterminals. First, we then handle the constraints of the form $FIRST(\beta) \subseteq FOLLOW(N)$: We compute $FIRST(\beta)$ and add this to $FOLLOW(N)$. Next, we handle the second type of constraints: For each constraint $FOLLOW(M) \subseteq FOLLOW(N)$, we add all elements of $FOLLOW(M)$ to $FOLLOW(N)$. We iterate these last steps until no further changes happen.

In summary, the steps taken to calculate the *FOLLOW* sets of a grammar are:

1. Extend the grammar by adding a new nonterminal $S' \to S\$$, where S is the start symbol for the original grammar. S' is the start symbol for the extended grammar.
2. For every occurrence of a nonterminal N on the right-hand side of a production, i.e., when there is a production $M \to \alpha N \beta$, where α and β are (possibly empty) sequences of grammar symbols, and N may or may not be equal to M, do the following:
   2.1. Let $m = FIRST(\beta)$. If $m \neq \emptyset$, add the constraint $m \subseteq FOLLOW(N)$ to the set of constraints.
   2.2. If $M \neq N$ and $Nullable(\beta)$, add the constraint $FOLLOW(M) \subseteq FOLLOW(N)$. Note that if β is empty, $Nullable(\beta)$ is trivially true.

   Note that if a production has several occurrences of nonterminals on its right-hand side, step 2 is done for all of these.
3. Solve the constraints using the following steps:
   3.1. Start with empty sets for $FOLLOW(N)$ for all nonterminals N (except S', which doesn't have a *FOLLOW* set).
   3.2. For each constraint of the form $m \subseteq FOLLOW(N)$ constructed in step 2.1, add the contents of m to $FOLLOW(N)$.
   3.3. Iterating until a fixed-point is reached, for each constraint of the form $FOLLOW(M) \subseteq FOLLOW(N)$, add the contents of $FOLLOW(M)$ to $FOLLOW(N)$.

We can take Grammar 2.4 as an example of this. We first add the production

$$T' \to T\$$$

to the grammar to handle end-of-text conditions. The table below shows the constraints generated by each production.

Production	Constraints
$T' \to T\$$	$\{\$\} \subseteq FOLLOW(T)$
$T \to R$	$FOLLOW(T) \subseteq FOLLOW(R)$
$T \to \mathtt{a}T\mathtt{c}$	$\{\mathtt{c}\} \subseteq FOLLOW(T)$
$R \to$	
$R \to R\mathtt{b}R$	$\{\mathtt{b}\} \subseteq FOLLOW(R)$

In the above table, we have already calculated the required *FIRST* sets, so they are shown as explicit lists of terminals. To initialise the *FOLLOW* sets, we first use the constraints that involve these *FIRST* sets:

$$\begin{aligned} FOLLOW(T) &\supseteq \{\$, \mathtt{c}\} \\ FOLLOW(R) &\supseteq \{\mathtt{b}\} \end{aligned}$$

and then iterate calculation of the subset constraints. The only such constraint is $FOLLOW(T) \subseteq FOLLOW(R)$, so we get

$$\begin{aligned} FOLLOW(T) &\supseteq \{\$, \mathtt{c}\} \\ FOLLOW(R) &\supseteq \{\$, \mathtt{c}, \mathtt{b}\} \end{aligned}$$

Now all constraints are satisfied, so we can replace subset with equality:

$$\begin{aligned} FOLLOW(T) &= \{\$, \mathtt{c}\} \\ FOLLOW(R) &= \{\$, \mathtt{c}, \mathtt{b}\} \end{aligned}$$

If we return to the question of predictive parsing of Grammar 2.4, we see that, for the nonterminal R, we should choose the empty production on any symbol in $FOLLOW(R)$, i.e., $\{\$, \mathtt{c}, \mathtt{b}\}$, and choose the non-empty production on the symbols in $FIRST(R\mathtt{b}R)$, i.e., $\{\mathtt{b}\}$. Since these sets overlap (on the symbol $\mathtt{b}$), we can not uniquely choose a production for R based on the next input symbol. Hence, the revised construction of predictive parsers (see below) will reject this grammar as possibly ambiguous.

2.10 A Larger Example

The above examples of calculating *FIRST* and *FOLLOW* are rather small, so we show a somewhat more substantial example. The following grammar describes even-length strings of as and bs that are *not* of the form ww where w is any string of as and bs.

In other words, a string can *not* consist of two identical halves, but otherwise any even-length sequence of `a`s and `b`s is accepted.

$$
\begin{aligned}
N &\rightarrow A\ B \\
N &\rightarrow B\ A \\
A &\rightarrow \texttt{a} \\
A &\rightarrow C\ A\ C \\
B &\rightarrow \texttt{b} \\
B &\rightarrow C\ B\ C \\
C &\rightarrow \texttt{a} \\
C &\rightarrow \texttt{b}
\end{aligned}
$$

The grammar is based on the observation that, if the string does not consist of two identical halves, there must be a point in the first part that has an `a` where the equivalent point in the second part has a `b`, or vice-versa. The grammar states that one of these is the case. The grammar is ambiguous, so we can not use predictive parsing, but it is used as a nontrivial example of calculation of *FIRST* and *FOLLOW* sets.

First, we note that there are no empty productions in the grammar, so no production can be *Nullable*. So we immediately set up the equations for *FIRST*:

$$
\begin{aligned}
\mathit{FIRST}(N) &= \mathit{FIRST}(A\ B) \cup \mathit{FIRST}(B\ A) \\
&= \mathit{FIRST}(A) \cup \mathit{FIRST}(B) \\
\mathit{FIRST}(A) &= \mathit{FIRST}(\texttt{a}) \cup \mathit{FIRST}(C\ A\ C) \\
&= \{\texttt{a}\} \cup \mathit{FIRST}(C) \\
\mathit{FIRST}(B) &= \mathit{FIRST}(\texttt{b}) \cup \mathit{FIRST}(C\ B\ C) \\
&= \{\texttt{b}\} \cup \mathit{FIRST}(C) \\
\mathit{FIRST}(C) &= \mathit{FIRST}(\texttt{a}) \cup \mathit{FIRST}(\texttt{b}) \\
&= \{\texttt{a}, \texttt{b}\}
\end{aligned}
$$

which we solve by fixed-point iteration. We initially set the *FIRST* sets for the nonterminals to the empty sets, and iterate evaluation:

Nonterminal	Iteration 1	Iteration 2	Iteration 3
N	$\emptyset$	{a, b}	{a, b}
A	{a}	{a, b}	{a, b}
B	{b}	{a, b}	{a, b}
C	{a, b}	{a, b}	{a, b}

The last iteration did not add anything, so the fixed-point is reached. We now add the production $N' \rightarrow N\$$, and set up the constraints for calculating *FOLLOW* sets:

Production	Constraints
$N' \to N\$$	$\{\$\} \subseteq FOLLOW(N)$
$N \to A\ B$	$FIRST(B) \subseteq FOLLOW(A)$, $FOLLOW(N) \subseteq FOLLOW(B)$
$N \to B\ A$	$FIRST(A) \subseteq FOLLOW(B)$, $FOLLOW(N) \subseteq FOLLOW(A)$
$A \to \texttt{a}$	
$A \to C\ A\ C$	$FIRST(A) \subseteq FOLLOW(C)$, $FIRST(C) \subseteq FOLLOW(A)$, $FOLLOW(A) \subseteq FOLLOW(C)$
$B \to \texttt{b}$	
$B \to C\ B\ C$	$FIRST(B) \subseteq FOLLOW(C)$, $FIRST(C) \subseteq FOLLOW(B)$, $FOLLOW(B) \subseteq FOLLOW(C)$
$C \to \texttt{a}$	
$C \to \texttt{b}$	

We first use the constraint $\{\$\} \subseteq FOLLOW(N)$ and the constraints of the form $FIRST(\cdots) \subseteq FOLLOW(\cdots)$ to get the initial sets:

$$\begin{aligned} FOLLOW(N) &\supseteq \{\$\} \\ FOLLOW(A) &\supseteq \{\texttt{a}, \texttt{b}\} \\ FOLLOW(B) &\supseteq \{\texttt{a}, \texttt{b}\} \\ FOLLOW(C) &\supseteq \{\texttt{a}, \texttt{b}\} \end{aligned}$$

and then use the constraints of the form $FOLLOW(\cdots) \subseteq FOLLOW(\cdots)$. If we do this in top-down order, we get after one iteration:

$$\begin{aligned} FOLLOW(N) &\supseteq \{\$\} \\ FOLLOW(A) &\supseteq \{\texttt{a}, \texttt{b}, \$\} \\ FOLLOW(B) &\supseteq \{\texttt{a}, \texttt{b}, \$\} \\ FOLLOW(C) &\supseteq \{\texttt{a}, \texttt{b}, \$\} \end{aligned}$$

Another iteration does not add anything, so the final result is

$$\begin{aligned} FOLLOW(N) &= \{\$\} \\ FOLLOW(A) &= \{\texttt{a}, \texttt{b}, \$\} \\ FOLLOW(B) &= \{\texttt{a}, \texttt{b}, \$\} \\ FOLLOW(C) &= \{\texttt{a}, \texttt{b}, \$\} \end{aligned}$$

Suggested exercises: 2.8 (*FOLLOW* only).

2.11 LL(1) Parsing

We have, in the previous sections, looked at how we can choose productions based on *FIRST* and *FOLLOW* sets, i.e., using the rule that we choose a production $N \to \alpha$ on input symbol c if either

- $c \in FIRST(\alpha)$, or
- $Nullable(\alpha)$ and $c \in FOLLOW(N)$.

If we can always choose a production uniquely by using these rules, this is called LL(1) parsing—the first L indicates the reading direction (left-to-right), the second L indicates the derivation order (left), and the (1) indicates that there is a one-symbol lookahead, i.e., that decisions require looking only at one input symbol (the next input symbol). A grammar where strings can be unambiguously parsed or rejected using LL(1) parsing is called an LL(1) grammar.

In the rest of this section, we shall see how we can implement LL(1) parsers as programs. We look at two implementation methods: Recursive descent, where grammar structure is directly translated into the program structure, and a table-based approach that encodes the production choices in a table, so a simple grammar-independent program can use the table to do parsing.

2.11.1 Recursive Descent

As the name indicates, *recursive descent* uses recursive functions to implement predictive parsing. The central idea is that each nonterminal in the grammar is implemented by a function in the program.

Each such function looks at the next input symbol in order to choose one of the productions for the nonterminal, using the criteria shown in the beginning of Sect. 2.11. The right-hand side of the chosen production is then used for parsing in the following way:

- A terminal on the right-hand side is matched against the next input symbol. If they match, we move on to the following input symbol and the next symbol on the right hand side, otherwise an error is reported.
- A nonterminal on the right-hand side is handled by calling the corresponding function and, after this call returns, continuing with the next symbol on the right-hand side.

When there are no more symbols on the right-hand side, the function returns.

As an example, Fig. 2.17 shows pseudo-code for a recursive descent parser for Grammar 2.9. We have constructed this program by the following process:

We have first added a production $T' \to T\$$ and calculated *FIRST* and *FOLLOW* for all productions.

T' has only one production, so the choice is trivial. However, we have added a check on the input symbol anyway, so we can report an error if it is not in $FIRST(T')$. This is shown in the function `parseT'`. The function `match` takes as argument a symbol, which it tests for equality with the symbol in the variable `input`. If they are equal, the following input symbol is read into the variable `input`. We assume `input` is initialised to the first input symbol before `parseT'` is called.

```
function parseT'() =
  if input = 'a' or input = 'b' or input = '$' then
    parseT() ; match('$')
  else reportError()

function parseT() =
  if input = 'b' or input = 'c' or input = '$' then
    parseR()
  else if input = 'a' then
    match('a') ; parseT() ; match('c')
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    (* do nothing, just return *)
  else if input = 'b' then
    match('b') ; parseR()
  else reportError()
```

Fig. 2.17 Recursive descent parser for Grammar 2.9

For the `parseT` function, we look at the productions for T. As $FIRST(R) = \{\texttt{b}\}$, the production $T \rightarrow R$ is chosen on the symbol `b`. Since R is also *Nullable*, we must choose this production also on symbols in $FOLLOW(T)$, i.e., `c` or \$. $FIRST(\texttt{a}T\texttt{c}) = \{\texttt{a}\}$, so we select $T \rightarrow \texttt{a}T\texttt{c}$ on an `a`. On all other symbols we report an error.

For `parseR`, we must choose the empty production on symbols in $FOLLOW(R)$ (`c` or \$). The production $R \rightarrow \texttt{b}R$ is chosen on input `b`. Again, all other symbols produce an error.

The program in Fig. 2.17 does not build a syntax tree—it only checks if the input is valid. It can be extended to construct a syntax tree by letting the parse functions return the sub-trees for the parts of input that they parse. Pseudo-code for this is shown in Fig. 2.18. Note that, while decisions are made top-down, the syntax tree is built bottom-up by combining sub-trees from recursive calls. We use the functions `tNode` and `nNode` to build nodes in the syntax tree. `tNode` takes as argument a terminal symbol and builds a leaf node equal to that terminal. `nNode` takes as arguments the name of a nonterminal and a list of subtrees and builds a tree with the nonterminal as root and the subtrees as children. Lists are shown in square brackets with elements separated by commas.

2.11.2 Table-Driven LL(1) Parsing

In table-driven LL(1) parsing, we encode the selection of productions into a table instead of in the program text. A simple non-recursive program uses this table and a stack to perform the parsing.

```
function parseT'() =
  if input = 'a' or input = 'b' or input = '$' then
     let tree = parseT() in
        match('$');
        return tree
  else reportError()

function parseT() =
  if input = 'b' or input = 'c' or input = '$' then
    let tree = parseR() in
      return nNode('T', [tree])
  else if input = 'a' then
    match('a') ;
    let tree = parseT() in
      match('c') ;
      return nNode('T', [tNode('a'),tree,tNode('c')])
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    return nNode('R', [])
  else if input = 'b' then
    match('b') ;
    let tree = parseR() in
      return nNode('R', [tNode('b'),tree])
  else reportError()
```

Fig. 2.18 Tree-building recursive descent parser for Grammar 2.9

Nonterminal	a	b	c	$
T'	$T' \to T\$$	$T' \to T\$$		$T' \to T\$$
T	$T \to \mathtt{a}T\mathtt{c}$	$T \to R$	$T \to R$	$T \to R$
R		$R \to \mathtt{b}R$	$R \to$	$R \to$

Fig. 2.19 LL(1) table for Grammar 2.9

The table is cross-indexed by nonterminal N and terminal a and contains for each such pair the production (if any) that is chosen for N when a is the next input symbol. This decision is made just as for recursive descent parsing: The production $N \to \alpha$ is written in the table at position (N,a) if either $\mathtt{a} \in \mathit{FIRST}(\alpha)$, or if both $\mathit{Nullable}(\alpha)$ and $\mathtt{a} \in \mathit{FOLLOW}(N)$. For Grammar 2.9 we get the table shown in Fig. 2.19.

Fig. 2.20 shows a program that uses this table to parse a string. It uses a stack, which at any time (read from top to bottom) contains the part of the current derivation that has not yet been matched to the input. When this eventually becomes empty, the parse is finished. If the stack is non-empty, and the top of the stack contains a terminal, that terminal is matched against the input and popped from the stack. Otherwise, the top of the stack must be a nonterminal, which we cross-index in the table with the next input symbol. If the table-entry is empty, we report an error. If

```
stack := empty ; push(T',stack)
while stack <> empty do
  if top(stack) is a terminal then
    match(top(stack)) ; pop(stack)
  else if table(top(stack),input) = empty then
    reportError
  else
    rhs := rightHandSide(table(top(stack),input)) ;
    pop(stack) ;
    pushList(rhs,stack)
```

Fig. 2.20 Program for table-driven LL(1) parsing

input	stack
aabbbcc$	T'
aabbbcc$	$T$$
aabbbcc$	aTc$
abbbcc$	Tc$
abbbcc$	aTcc$
bbbcc$	Tcc$
bbbcc$	Rcc$
bbbcc$	bRcc$
bbcc$	Rcc$
bbcc$	bRcc$
bcc$	Rcc$
bcc$	bRcc$
cc$	Rcc$
cc$	cc$
c$	c$
$	$

Fig. 2.21 Input and stack during table-driven LL(1) parsing

not, we pop the nonterminal from the stack and replace this by the right-hand side of the production in the table entry. The list of symbols on the right-hand side are pushed such that the first symbol will end up at the top of the stack.

As an example, Fig. 2.21 shows the input and stack at each step during parsing of the string `aabbbcc$` using the table in Fig. 2.19. The stack is shown horizontally with the top to the left.

The program in Fig. 2.20, like the one in Fig. 2.17, only checks if the input is valid. It, too, can be extended to build a syntax tree. Figure 2.22 shows pseudo-code for this. The stack now holds nodes in the syntax tree instead of grammar symbols. The `match` function now matches a terminal (leaf) node with the next input symbol. The function `makeNodes` takes a list of grammar symbols and creates a list of nodes, one for each grammar symbol in its argument. A nonterminal node is created with an *uninitialised* mutable field for the list of its children. The `T'-node` that is pushed to the initial stack is a nonterminal node corresponding to the nonterminal `T'`. Note

```
stack := empty ; push(T'-node,stack)
while stack <> empty do
  node := pop(stack) ;
  if node is a terminal node then
    match(node) ; pop(stack)
  else
    nonterminal := node.nonterminal ;
    if table(nonterminal,input) = empty then
       reportError
    else
      rhs := rightHandSide(table(nonterminal,input)) ;
      node.children := makeNodes(rhs);
      pushList(node.children,stack)
```

Fig. 2.22 Tree-building program for table-driven LL(1) parsing

that the tree is built by first (in a top-down manner) creating nonterminal nodes with uninitialised fields and later (in a bottom-up manner) overwriting these uninitialised fields with subtrees.

2.11.3 Conflicts

When a symbol `a` allows several choices of production for nonterminal N we say that there is a *conflict* on that symbol for that nonterminal. Conflicts may be caused by ambiguous grammars (indeed all ambiguous grammars will cause conflicts) but there are also unambiguous grammars that cause conflicts. An example of this is the unambiguous expression grammar (Grammar 2.12). We will in the next section see how we can rewrite this grammar to avoid conflicts, but it must be noted that this is not always possible: There are languages for which there exist unambiguous context-free grammars but where no grammar for the language generates a conflict-free LL(1) table. Such languages are said to be non-LL(1). It is, however, important to note the difference between a non-LL(1) language and a non-LL(1) grammar: A language may well be LL(1) even though a grammar used to describe it is not. This just means that there is another grammar (which is LL(1)) for the same language. Our goal is to take a a non-LL(1) grammar for a LL(1) language and transform it into an equivalent LL(1) grammar.

2.12 Rewriting a Grammar for LL(1) Parsing

In this section we will look at methods for rewriting grammars such that they are more palatable for LL(1) parsing. In particular, we will look at *elimination of left-recursion* and at *left factorisation*.

It must, however, be noted that not all unambiguous grammars can be rewritten to allow LL(1) parsing. In these cases stronger parsing techniques must be used. We will not cover parsing of ambiguous grammars in this book.

2.12.1 Eliminating Left-Recursion

As mentioned above, the unambiguous expression grammar (Grammar 2.12) is not LL(1). The reason is that all productions in *Exp* and *Exp*2 have the same *FIRST* sets. Overlap like this will always happen when there are directly or indirectly left-recursive productions in the grammar, as the *FIRST* set of a left-recursive production will include the *FIRST* set of the nonterminal itself and hence be a superset of the *FIRST* sets of all the other productions for that nonterminal. To solve this problem, we must avoid left-recursion in the grammar. We start by looking at elimination of direct left-recursion.

When we have a nonterminal with some left-recursive productions and some productions that are not left-recursive, i.e.,

$$\begin{array}{l} N \rightarrow N\,\alpha_1 \\ \quad\vdots \\ N \rightarrow N\,\alpha_m \\ N \rightarrow \beta_1 \\ \quad\vdots \\ N \rightarrow \beta_n \end{array}$$

where the β_i do not start with N, we observe that the nonterminal N generates all sequences that start with one of the β_i and continue with any number (including 0) of the α_j. In other words, the grammar is equivalent to the regular expression $(\beta_1 \mid \ldots \mid \beta_n)(\alpha_1 \mid \ldots \mid \alpha_m)^*$. Some LL(1) parser generators accept grammars with right-hand sides of this form. When using such parser generators, no further rewriting is required. When using simple grammar notation, more rewriting is required, which we will look at below.

We saw in Fig. 2.1 a method for converting regular expressions into context-free grammars that generate the same set of strings. By following this procedure and simplifying a bit afterwards, we get this equivalent grammar:

$$\begin{array}{l} N \rightarrow \beta_1\,N_* \\ \quad\vdots \\ N \rightarrow \beta_n\,N_* \\ \\ N_* \rightarrow \alpha_1\,N_* \\ \quad\vdots \\ N_* \rightarrow \alpha_m\,N_* \\ N_* \rightarrow \end{array}$$

where N_* is a new nonterminal that generates a (possibly empty) sequence of αs.

Grammar 2.23 Removing left-recursion from Grammar 2.12

$$
\begin{array}{lll}
Exp & \rightarrow & Exp2\ Exp_* \\
Exp_* & \rightarrow & +\ Exp2\ Exp_* \\
Exp_* & \rightarrow & -\ Exp2\ Exp_* \\
Exp_* & \rightarrow & \\
Exp2 & \rightarrow & Exp3\ Exp2_* \\
Exp2_* & \rightarrow & *\ Exp3\ Exp2_* \\
Exp2_* & \rightarrow & /\ Exp3\ Exp2_* \\
Exp2_* & \rightarrow & \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & (\ Exp\ )
\end{array}
$$

Note that, since the β_i do not start with N, there is no direct left-recursion in the first n productions. Since N_* is a new nonterminal, no α_j can start with this, so the last m productions can't be directly left-recursive either.

There may, however, still be *indirect* left-recursion: If an α_j is nullable, the corresponding production for N_* is indirectly left-recursive. If a β_i can derive something starting with N, the corresponding production for N is indirectly left-recursive. We will briefly look at indirect left-recursion below.

While we have eliminated direct left-recursion, we have also changed the syntax trees that are built from the strings that are parsed. Hence, after parsing, the syntax tree must be re-structured to obtain the structure that the original grammar describes. We will return to this in Sect. 2.16.

As an example of left-recursion removal, we take the unambiguous expression Grammar 2.12. This has left recursion in both *Exp* and *Exp*2, so we apply the transformation to both of these to obtain Grammar 2.23. The resulting Grammar 2.23 is now LL(1), which can be verified by generating an LL(1) table for it.

Indirect Left-Recursion

The transformation shown in Sect. 2.12.1 is only applicable in the simple case where there only *direct left-recursion*. Indirect left-recursion can have several forms:

1. There are mutually left-recursive productions

$$
\begin{array}{lll}
N_1 & \rightarrow & N_2\alpha_1 \\
N_2 & \rightarrow & N_3\alpha_2 \\
 & \vdots & \\
N_{k-1} & \rightarrow & N_k\alpha_{k-1} \\
N_k & \rightarrow & N_1\alpha_k
\end{array}
$$

2. There is a production $N \rightarrow \alpha N \beta$ where α is *Nullable*.

or any combination of the two. More precisely, a grammar is (directly or indirectly) left-recursive if there is a non-empty derivation sequence $N \Rightarrow N\alpha$, i.e., if a nonterminal derives a sequence of grammar symbols that start by that same nonterminal.

Grammar 2.24
Left-factorised grammar for conditionals

Stat → **id** := *Exp*
Stat → `if` *Exp* `then` *Stat ElsePart*

ElsePart → `else` *Stat*
ElsePart →

If there is indirect left-recursion, we must first rewrite the grammar to make the left-recursion direct and then use the transformation above.

Rewriting a grammar to turn indirect left-recursion into direct left-recursion can be done systematically, but the process is a bit complicated. Details can be found in [2]. We will not go into this here, as in practice most cases of left-recursion are direct left-recursion.

2.12.2 Left-Factorisation

If two productions for the same nonterminal begin with the same sequence of symbols, they obviously have overlapping *FIRST* sets. As an example, in Grammar 2.3 the two productions for `if` have overlapping prefixes. We rewrite this in such a way that the overlapping productions are made into a single production that contains the common prefix of the productions and uses a new auxiliary nonterminal for the different suffixes. See Grammar 2.24. In this grammar,[3] we can uniquely choose one of the productions for *Stat* based on one input token.

For most grammars, combining productions with common prefix will solve the problem. However, in this particular example the grammar still is not LL(1): We can not uniquely choose a production for the auxiliary nonterminal *ElsePart*, since `else` is in *FOLLOW*(*ElsePart*) as well as in the *FIRST* set of the first production for *ElsePart*. This should not be a surprise to us, since, after all, the grammar is ambiguous and ambiguous grammars can not be LL(1). The equivalent unambiguous grammar (Grammar 2.14) can not easily be rewritten to a form suitable for LL(1), so in practice Grammar 2.24 is used anyway and the conflict is handled not by rewriting to an unambiguous grammar, but by using an ambiguous grammar and resolving the conflict by prioritising productions. If the non-empty production for *ElsePart* has higher priority than the empty production, we will choose the non-empty production when the next input symbol is `else`. This gives the desired behaviour of letting an `else` match the nearest `if`.

Whenever an LL(1) table would have multiple choices of production for the same nonterminal/terminal pair, we use the priorities to select a single production. Most LL(1) parser generators prioritise productions by the order in which they are written, so Grammar 2.24 will give the desired behaviour. Unfortunately, few conflicts in

[3] We have omitted the production for semicolon, as that would only muddle the issue by introducing more ambiguity.

LL(1) tables can be removed by prioritising productions without also changing the language recognised by the grammar. For example, operator precedence ambiguity can not be resolved by prioritising productions. Attempting to do so will cause parse errors for some valid expressions.

2.12.3 Construction of LL(1) Parsers Summarised

Constructing an LL(1) parser from a given grammar is done in the following steps.

1. Eliminate ambiguity that can not be resolved by prioritising productions.
2. Eliminate left-recursion.
3. Perform left factorisation where required.
4. Add an extra start production $S' \rightarrow S\$$ to the grammar.
5. Calculate *FIRST* for every production and *FOLLOW* for every nonterminal.
6. For nonterminal N and input symbol c, choose production $N \rightarrow \alpha$ when:

   - $c \in FIRST(\alpha)$, or
   - $Nullable(\alpha)$ and $c \in FOLLOW(N)$.

   This choice is encoded either in a table or a recursive-descent program.
7. Use production priorities to eliminate conflicts where appropriate.

Suggested exercises: 2.14.

2.13 SLR Parsing

A problem with LL(1) parsing is that most grammars need extensive rewriting to get them into a form that allows unique choice of production. Even though this rewriting can, to a large extent, be automated, there are still a large number of grammars that can not be automatically transformed into LL(1) grammars.

LR parsers is a class of bottom-up methods for parsing that can solve the parsing problem for a much larger class of grammars than LL(1) parsing, though still not all grammars. The main advantage of LR parsing is that less rewriting is required to get a grammar in acceptable form for LR parsing than is the case for LL(1) parsing. Furthermore, as we shall see in Sect. 2.15, LR parsers allow external declarations for resolving operator precedences, instead of requiring the grammars themselves to be rewritten.

We will look at a simple form of LR-parsing called SLR parsing. The letters "SLR" stand for "Simple", "Left" and "Right". "Left" indicates that the input is read from left to right and the "Right" indicates that a rightmost derivation is built.

LR parsers are also called *shift-reduce parsers*. They are table-driven bottom-up parsers and use two kinds of "actions" involving the input stream and a stack:

Fig. 2.25 Example shift-reduce parsing

stack	input	action
	aabbbcc	shift 5 times
aabbb	cc	reduce with $R \to$
aabbb$R$	cc	reduce with $R \to$ bR
aabb$R$	cc	reduce with $R \to$ bR
aab$R$	cc	reduce with $R \to$ bR
aa$R$	cc	reduce with $T \to R$
aa$T$	cc	shift
aa$T$c	c	reduce with $T \to$ aTc
aT	c	shift
aTc		reduce with $T \to$ aTc
T		

shift: A terminal symbol is read from the input and pushed on the stack.

reduce: The top n elements of the stack hold symbols identical to the n symbols on the right-hand side of a specified production. These n symbols are by the reduce action replaced by the nonterminal at the left-hand side of the specified production. Contrary to LL(1) parsers, the stack holds the right-hand-side symbols such that the *last* symbol on the right-hand side is at the top of the stack.

If the input text does not conform to the grammar, there will at some point during the parsing be no applicable actions, and the parser will stop with an error message. Otherwise, the parser will read through all the input and leave a single element (the start symbol of the grammar) on the stack. To illustrate shift-reduce parsing, Fig. 2.25 shows a sequence of shift and reduce actions corresponding to parsing the string aabbbcc using Grammar 2.9. The stack is shown growing left to right, so the rightmost stack element is the top. In this example, we do not explain how we select between shift and reduce, but we want to make well-informed choices and to detect potential ambiguity. This is done using SLR parsing.

As with LL(1), our aim is to make the choice of action depend only on the next input symbol and the symbol on top of the stack. To help make this choice, we use a DFA. Conceptually, this DFA reads the contents of the stack (which contains both terminals and nonterminals), starting from the bottom up to the top. The state of the DFA when the top of the stack is reached is, together with the next input symbol, used to determine the next action. Like in LL(1) parsing, this is done using a table, but we use a DFA state instead of a nonterminal to select the row in the table, and the table entries are not productions but actions.

If the action is a shift action, there is no need to start over from the bottom of the stack to find the next action: We just push the input symbol and follow a transition from the current DFA state on this symbol, which gives us the DFA state we need for the next choice.

If the action is a reduce action, we pop off the stack symbols corresponding to the right-hand side of the selected production, and then push the nonterminal on the left-hand side of this production. To make a DFA transition on this nonterminal, we need to know the state of the DFA when reading the stack from bottom to the new

top. To avoid having to start over from the bottom, we remember the transitions we have already made, so the stack holds not only grammar symbols but also states. When we pop a symbol off the stack, we can find the previous DFA state on the new stack top. This is similar to how a lexical analyser remembers past states so it can find the most recent accepting state if a transition fails (see Sect. 1.8). So, when a reduce action pops off the symbols corresponding to the right-hand side of a production, we can find the DFA state that we use to make a transition on the nonterminal on the left-hand side of the production. This nonterminal is pushed on to the new stack together with the new state.

With these optimisations, the DFA only has to make one transition when an action is made: A transition on a terminal when a shift action is made, and a transition on a nonterminal when a reduce action is made.

We represent the DFA as a table, where we cross-index a DFA state with a symbol (terminal or nonterminal) and find one of the following actions:

shift n:	Push the current input symbol and then state *n* on the stack, and read the next input symbol. This corresponds to a transition on a terminal.
go n:	Push the nonterminal indicated by the column and then state *n* on the stack. This corresponds to a transition on a nonterminal.
reduce p:	Reduce with the production numbered *p*: Pop symbols (interleaved with state numbers) corresponding to the right-hand side of the production off the stack. This is always followed by a *go* action on the left-hand side nonterminal using the DFA state that is found *after* popping the right-hand side off the stack.
accept:	Parsing has completed successfully.
error:	A syntax error has been detected. This happens when no *shift*, *accept* or *reduce* action is defined for the input symbol.

Note that the current state is always found at the top of the stack.

An example SLR table is shown in Fig. 2.26. The table has been produced from Grammar 2.9 by the method shown below in Sect. 2.14. The actions have been abbreviated to their first letters and *error* is shown as a blank entry. The rows are indexed by DFA states (0 to 7) and the columns are indexed by terminals (including \$) or nonterminals.

The algorithm for parsing a string using the table is shown in Fig. 2.27. The shown algorithm just determines if a string is in the language generated by the grammar. It can, however, easily be extended to build a syntax tree: Instead of grammar symbols, the stack contains syntax trees. When performing a *reduce* action, a new syntax tree is built by using the nonterminal from the reduced production as root and the syntax trees stored at the popped-off stack elements as children. The new tree is pushed on the stack instead of just pushing the nonterminal.

Figure 2.28 shows an example of parsing the string `aabbbcc` using the table in Fig. 2.26. The "stack" column represents the stack contents with the stack bottom shown to the left and the stack top to the right. We interleave grammar symbols and states on the stack, always leaving the current state on the top (at the right). At each step, we look at the current input symbol (at the left end of the string in the input column) and the state at the top of the stack (at the right end of the sequence in the

State	a	b	c	$	*T*	*R*
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g2
4		s4	r3	r3		g6
5			s7			
6			r4	r4		
7			r2	r2		

Fig. 2.26 SLR table for Grammar 2.9

```
stack := empty ; push(0,stack) ; read(input)
loop
  case table[top(stack),input] of
    shift s:  push(input,stack) ;
              push(s,stack) ;
              read(input)

    reduce p: n := the left-hand side of production p ;
              r := the number of symbols
                        on the right-hand side of p ;
              pop 2*r elements from the stack ;
              let go s = table[top(stack),n];
              push(n,stack) ;
              push(s,stack)

    accept:   terminate with success

    error:    reportError
endloop
```

Fig. 2.27 Algorithm for SLR parsing

stack column). We look up the pair of input symbol and state in the table and find the action (shown in the action column) that leads to the stack and input shown in next row. When the action is a *reduce* action, we also show the reduction used (in parentheses), and after a semicolon also the *go* action that is performed after the reduction. At the end, the root nonterminal T is found as the second stack element. If a syntax tree is built, this will be placed here.

2.14 Constructing SLR Parse Tables

An SLR parse table has a DFA as its core. Constructing this DFA from the grammar is similar to constructing a DFA from a regular expression, as shown in Chap. 2: We

stack	input	action
0	aabbbcc$	s3
0a3	abbbcc$	s3
0a3a3	bbbcc$	s4
0a3a3b4	bbcc$	s4
0a3a3b4b4	bcc$	s4
0a3a3b4b4b4	cc$	r3 ($R \to$) ; g6
0a3a3b4b4b4*R*6	cc$	r4 ($R \to$ bR) ; g6
0a3a3b4b4*R*6	cc$	r4 ($R \to$ bR) ; g6
0a3a3b4*R*6	cc$	r4 ($R \to$ bR) ; g2
0a3a3*R*2	cc$	r1 ($T \to R$) ; g5
0a3a3*T*5	cc$	s7
0a3a3*T*5c7	c$	r2 ($T \to$ aTc) ; g5
0a3*T*5	c$	s7
0a3*T*5c7	$	r2 ($T \to$ aTc) ; g1
0*T*1	$	accept

Fig. 2.28 Example SLR parsing

$$
\begin{aligned}
&0\text{:}\ T' \to T\\
&1\text{:}\ T \to R\\
&2\text{:}\ T \to \mathtt{a}T\mathtt{c}\\
&3\text{:}\ R \to\\
&4\text{:}\ R \to \mathtt{b}R
\end{aligned}
$$

Grammar 2.29 Example grammar for SLR-table construction

first construct an NFA using techniques similar to those in Sect. 1.3 and then convert this into a DFA using the construction shown in Sect. 1.5.

Before we construct the NFA, we extend the grammar with a new starting production. Doing this to Grammar 2.9 yields Grammar 2.29.

The next step is to make an NFA for each production. This is done as in Sect. 1.3, treating both terminals and nonterminals as alphabet symbols. The accepting state of each NFA is labeled with the number of the corresponding production. The result is shown in Fig. 2.30. Note that we have used the optimised construction for ε (the empty production) as shown in Fig. 1.6. For identification purposes, we label the states with letters.

The NFAs in Fig. 2.30 make transitions both on terminals and nonterminals. Transitions by terminal corresponds to *shift* actions and transitions on nonterminals correspond to *go* actions. A *go* action happens after a reduction, so before we can make a transition on a nonterminal, we must on the stack have symbols corresponding to a right-hand side of a production for that nonterminal. So whenever an NFA can make a transition on a nonterminal, we add epsilon transitions to the NFAs for the right-hand sides of the productions for that nonterminal. This way, we can make transitions for a right-hand side, make a reduction, and then a transition on the nonterminal.

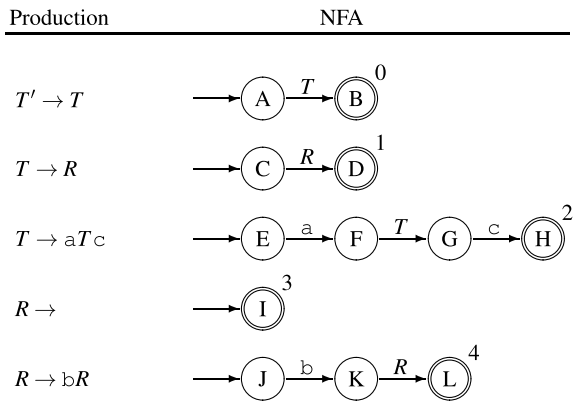


Fig. 2.30 NFAs for the productions in Grammar 2.29

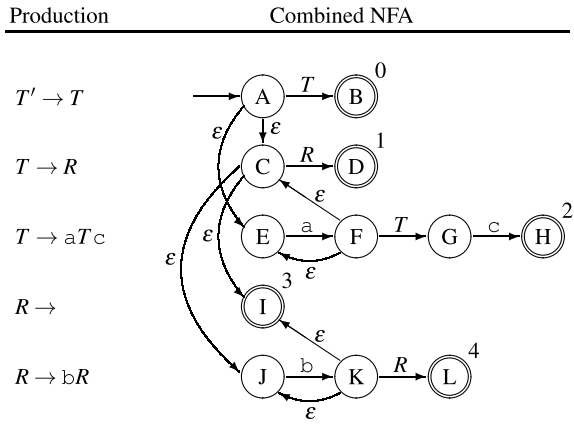


Fig. 2.31 Combined NFA for Grammar 2.29: epsilon transitions are added, and A is the only start state

Finally, we combine the NFAs to a single NFA by letting A (the start state of the production for the added start symbol T') be the only initial state. The result is shown in Fig. 2.31.

We must now convert this NFA into a DFA using the subset construction shown in Sect. 1.5.2. The result is shown in Fig. 2.32. The states are labelled with the sets of NFA states that are combined into the DFA states.

From this DFA, we construct a table where transitions on terminals are shown as *shift* actions and transitions on nonterminals as *go* actions. We use state number 0 for the starting state of the DFA. The order of the other states is not important, but we have numbered them in the order they were generated using the work-list algorithm from Sect. 1.5.2. The table looks similar to Fig. 2.26, except that it has an extra column for sets of NFA states and that no *reduce* or *accept* actions are present yet. Figure 2.33 shows the table constructed from the DFA in Fig. 2.32. The sets of NFA states that form each DFA state is shown in the second column. We will need these below for adding *reduce* and *accept* actions, but once this is done, we will not need them anymore, so we can remove them from the final table.

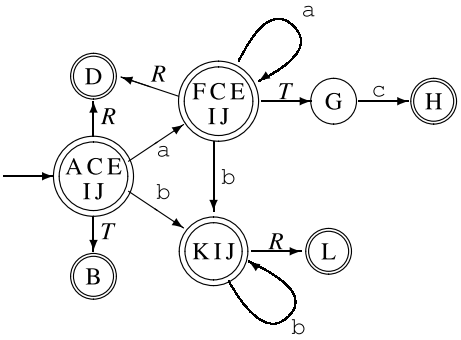


Fig. 2.32 DFA constructed from the NFA in Fig. 2.31

Fig. 2.33 DFA table for Grammar 2.9, equivalent to the DFA in Fig. 2.32

DFA state	NFA states	Transitions a	b	c	T	R
0	A, C, E, I, J	s3	s4		g1	g2
1	B					
2	D					
3	F, C, E, I, J	s3	s4		g5	g2
4	K, I, J		s4			g6
5	G			s7		
6	L					
7	H					

To add *reduce* and *accept* actions, we first need to compute the *FOLLOW* sets for each nonterminal, as described in Sect. 2.9. For purpose of calculating *FOLLOW*, we add yet another extra start production: $T'' \rightarrow T'\$$, to handle end-of-text conditions as described in Sect. 2.9. This gives us the following result:

$$\begin{aligned} FOLLOW(T') &= \{\$\} \\ FOLLOW(T) &= \{\mathtt{c}, \$\} \\ FOLLOW(R) &= \{\mathtt{c}, \$\} \end{aligned}$$

We now add *reduce* actions by the following rule: If a DFA state s contains the accepting NFA state for a production $p : N \rightarrow \alpha$, we add *reduce p* as action to s on all symbols in $FOLLOW(N)$. Reduction for production 0 (the extra start production that was added before constructing the NFA) on the \$ symbol is written as *accept*.

In Fig. 2.33, state 0 contains NFA state I, which accepts production 3. Hence, we add r3 as actions at the symbols c and \$ (as these are in $FOLLOW(R)$). State 1 contains NFA state B, which accepts production 0. Since $FOLLOW(T') = \{\$\}$, we add a reduce action for production 0 at \$. As noted above, this is written as *accept* (abbreviated to "a"). In the same way, we add reduce actions to state 3, 4, 6 and 7. The result is shown in Fig. 2.26.

Figure 2.34 summarises the SLR construction.

1. Add the production $S' \rightarrow S$, where S is the start symbol of the grammar.
2. Make an NFA for the right-hand side of each production.
3. If an NFA state s has an outgoing transition on a nonterminal N, add epsilon-transitions from s to the starting states of the NFAs for the right-hand sides of the productions for N.
4. Make the start state of the NFA for the production $S' \rightarrow S$ the single start state of the combined NFA.
5. Convert the combined NFA to a DFA.
6. Build a table cross-indexed by the DFA states and grammar symbols (terminals including \$ and nonterminals). Add *shift* actions for transitions on terminals and *go* actions for transitions on nonterminals.
7. Calculate *FOLLOW* for each nonterminal. For this purpose, we add one more start production: $S'' \rightarrow S'\$$.
8. When a DFA state contains an accepting NFA state marked with production number p, where the left-hand side nonterminal for p is N, find the symbols in $FOLLOW(N)$ and add a *reduce p* action in the DFA state at all these symbols. If $p = 0$, add an *accept* action instead of a *reduce p* action.

Fig. 2.34 Summary of SLR parse-table construction

2.14.1 Conflicts in SLR Parse-Tables

When *reduce* actions are added to SLR parse-tables, we might add a reduce action where there is already a *shift* action, or we may add *reduce* actions for two or more different productions to the same table entry. When either of these happen, we no longer have a unique choice of action, i.e., we have a *conflict*. The first situation is called a *shift-reduce conflict* and the other case a *reduce-reduce conflict*. Both can occur in the same table entry.

Conflicts are often caused by ambiguous grammars, but (as is the case for LL-parsers) some non-ambiguous grammars can generate conflicts. If a conflict is caused by an ambiguous grammar, it is usually (but not always) possible to find an equivalent unambiguous grammar. Methods for eliminating ambiguity were discussed in Sects. 2.3 and 2.4. Sometimes, operator precedence declarations can be used to disambiguate an ambiguous grammar, as we shall see in Sect. 2.15. In rare cases, a language is simply not SLR, so no language-preserving rewrites or use of precedence declarations will eliminate conflicts.

When a conflict is found, inspection of the NFA states that form the problematic DFA state will often help identifying the exact nature of the problem, which is the first step towards solving it. Sometimes, changing a production from left-recursive to right-recursive may help, even though left-recursion in general is not a problem for SLR-parsers, as it is for LL(1)-parsers. It may also help to rewrite the grammar in the following way: If there are productions of the form

$$
\begin{aligned}
A &\rightarrow \alpha\, B\, \beta \\
A &\rightarrow \alpha\, \gamma_1\, \delta \\
B &\rightarrow \gamma_1 \\
&\vdots \\
B &\rightarrow \gamma_n
\end{aligned}
$$

and there is overlap between $FIRST(\delta)$ and $FOLLOW(B)$, then there will be a shift-reduce conflict after reading $\alpha\, \gamma_1$, as both reduction with $B \rightarrow \gamma_1$ and shifting on any symbol in $FIRST(\delta)$ is possible, which gives a conflict for all symbols in $FIRST(\delta) \cap FOLLOW(B)$. This conflict can be resolved by splitting the first production above into all the possible cases for B:

$$
\begin{aligned}
A &\rightarrow \alpha\, \gamma_1\, \beta \\
&\vdots \\
A &\rightarrow \alpha\, \gamma_n\, \beta \\
A &\rightarrow \alpha\, \gamma_1\, \delta
\end{aligned}
$$

The shift-reduce conflict we had when having read $\alpha\, \gamma_1$ is now gone, as we have postponed reduction until we have read more input, which can determine if we should reduce by the production $A \rightarrow \alpha\, \gamma_1\, \beta$ or by the production $A \rightarrow \alpha\, \gamma_1\, \delta$. See also Sects. 2.15 and 2.16.1.

Suggested exercises: 2.16.

2.15 Using Precedence Rules in LR Parse Tables

We saw in Sect. 2.12.2, that the conflict arising from the dangling-else ambiguity could be removed by removing one of the entries in the LL(1) parse table. Resolving ambiguity by deleting conflicting actions can also be done in SLR-tables. In general, there are more cases where this can be done successfully for SLR-parsers than for LL(1)-parsers. In particular, ambiguity in expression grammars like Grammar 2.2 can be eliminated this way in an SLR table, but not in an LL(1) table. Most LR-parser generators allow declarations of precedence and associativity for tokens used as infix-operators. These declarations are then used to eliminate conflicts in the parse tables.

There are several advantages to this approach:

- Ambiguous expression grammars are more compact and easier to read than unambiguous grammars in the style of Sect. 2.3.1.
- The parse tables constructed from ambiguous grammars are often smaller than tables produced from equivalent unambiguous grammars.
- Parsing using ambiguous grammars is (slightly) faster, as fewer reductions of the form $Exp2 \rightarrow Exp3$ etc. are required.

Using precedence rules to eliminate conflicts is very simple. Grammar 2.2 will generate several conflicts:

1) A conflict between shifting on + and reducing by the production $Exp \rightarrow Exp + Exp$.
2) A conflict between shifting on + and reducing by the production $Exp \rightarrow Exp * Exp$.
3) A conflict between shifting on * and reducing by the production $Exp \rightarrow Exp + Exp$.
4) A conflict between shifting on * and reducing by the production $Exp \rightarrow Exp * Exp$.

And several more of similar nature involving - and /, for a total of 16 conflicts. Let us take each of the four conflicts above in turn and see how precedence rules can be used to eliminate them. We use the usual convention that + and * are both left-associative and that * binds more strongly than +.

1) This conflict arises from expressions like `a+b+c`. After having read `a+b`, the next input symbol is +. We can now either choose to reduce `a+b`, grouping around the first addition before the second, or shift on the plus, which will later lead to `b+c` being reduced, and hence grouping around the second addition before the first. Since the convention is that + is left-associative, we prefer the first of these options and, hence, eliminate the shift-action from the table and keep only the reduce-action.
2) The offending expressions here have the form `a*b+c`. Since convention make multiplication bind stronger than addition, we, again, prefer reduction over shifting.
3) In expressions of the form `a+b*c`, the convention, again, makes multiplication bind stronger, so we prefer a shift to avoid grouping around the + operator and, hence, eliminate the reduce-action from the table.
4) This case is identical to case 1, where an operator that by convention is left-associative conflicts with itself. We, as in case 1, handle this by eliminating the shift.

In general, elimination of conflicts by operator precedence declarations can be summarised into the following rules:

a) If the conflict is between two operators of different priority, eliminate the action with the lowest priority operator in favour of the action with the highest priority. In a reduce action, the operator associated with a reduce-action is an operator used in the production that is reduced. If several operators are used in the same production, the operator that is closest to the end of the production is used.[4]
b) If the conflict is between operators of the same priority, the associativity (which must be the same, as noted in Sect. 2.3.1) of the operators is used: If the operators are left-associative, the shift-action is eliminated and the reduce-action retained.

[4] Using several operators with declared priorities in the same production should be done with care.

If the operators are right-associative, the reduce-action is eliminated and the shift-action retained. If the operators are non-associative, both actions are eliminated (which will reduce the set of accepted programs).

Prefix and postfix operators can be handled similarly. Associativity only applies to infix operators, so only the precedence of prefix and postfix operators matters.

Note that only shift-reduce conflicts are eliminated by the above rules. Some parser generators allow also reduce-reduce conflicts to be eliminated by precedence rules (in which case the production with the highest-precedence operator is preferred), but this is not as obviously useful as the above.

The dangling-else ambiguity (Sect. 2.4) can also be eliminated using precedence rules. If we have read `if` *Exp* `then` *Stat* and the next symbol is a `else`, we want to shift on `else`, so the `else` will be associated with the `then`. Giving `else` a higher precedence than `then` or giving them both the same precedence and making them right-associative will ensure that a shift is made on `else` when we need it.

Not all conflicts should be eliminated by precedence rules. If you blindly add precedence rules until no conflicts are reported, you risk eliminating actions that are required to parse certain strings, so the parser will accept only a subset of the intended language. Normally, you should only use precedence declarations to specify operator hierarchies, unless you have analysed the parser actions carefully and found that there is no undesirable consequences of adding the precedence rules.

Suggested exercises: 2.18.

2.16 Using LR-Parser Generators

Most LR-parser generators use an extended version of the SLR construction called LALR(1). The "LA" in the abbreviation is short for "lookahead" and the (1) indicates that the lookahead is one symbol, i.e., the next input symbol. LALR(1) parser tables have fewer conflicts that SLR parser tables.

We have chosen to present the SLR construction instead of the LALR(1) construction for several reasons:

- It is simpler.
- In practice, SLR parse tables rarely have conflicts that would not also be conflicts in LALR(1) tables.
- When a grammar is in the SLR class, the parse-table produced by an SLR parser generator is identical to the table produced by an LALR(1) parser generator.
- If you use an LALR(1) parser generator and you do not get any conflicts, you do not need to worry about the difference.
- If you use an LALR(1) parser generator and you *do* get conflicts, understanding SLR parsing is sufficient to deal with the conflicts (by adding precedence declarations or by rewriting the grammar).

In short, the practical difference is small, and knowledge of SLR parsing is sufficient when using LALR(1) parser generators.

Most LR-parser generators organise their input in several sections:

- Declarations of the terminals and nonterminals used.
- Declaration of the start symbol of the grammar.
- Declarations of operator precedence.
- The productions of the grammar.
- Declaration of various auxiliary functions and data-types used in the actions (see below).

2.16.1 Conflict Handling in Parser Generators

For all but the simplest grammars, the user of a parser generator should expect conflicts to be reported when the grammar is first presented to the parser generator. These conflicts can be caused by ambiguity or by the limitations of the parsing method. In any case, the conflicts can normally be eliminated by rewriting the grammar or by adding precedence declarations.

Most parser generators can provide information that is useful to locate where in the grammar the problems are. When a parser generator reports a conflict, it will tell in which state in the table the conflict occur. Information about this state can be written out in a (barely) human-readable form as a set of NFA-states. Since most parser generators rely on pure ASCII, they can not actually draw the NFAs as diagrams. Instead, they rely on the fact that each state in a NFA corresponds to a position in a production in the grammar. If we, for example, look at the NFA states in Fig. 2.30, these would be written as shown in Fig. 2.35. Note that a '.' is used to indicate the position of the state in the production. State 4 of the table in Fig. 2.33 will hence be written as

```
R -> b . R
R -> .
R -> . bR
```

The set of NFA states, combined with information about on which symbols a conflict occurs, can be used to find a remedy, e.g., by adding precedence declarations. Note that a dot at the end of a production indicates an accepting NFA state (and, hence, a possible reduce action) while a dot before a terminal indicates a possible shift action. That both of these appear (as above) in the same DFA state does not imply a conflict—the symbols on which the reduce action is taken may not overlap the symbols on which shift actions are taken.

If all efforts to eliminate conflicts fail, a practical solution may be to change the grammar so it unambiguously accepts a larger language than the intended language, and then post-process the syntax tree to reject "false positives". This elimination can be done at the same time as type-checking (which, too, may reject programs).

Fig. 2.35 Textual representation of NFA states

NFA-state	Textual representation
A	`T′ -> . T`
B	`T′ -> T .`
C	`T -> . R`
D	`T -> R .`
E	`T -> . aTc`
F	`T -> a . Tc`
G	`T -> aT . c`
H	`T -> aTc .`
I	`R -> .`
J	`R -> . bR`
K	`R -> b . R`
L	`R -> bR .`

Some programming languages allow programs to declare precedence and associativity for user-defined operators. This can make it difficult to handle precedence during parsing, as the precedences are not known when the parser is generated. A possible solution is to parse all operators using the same precedence and associativity, and then restructure the syntax tree afterwards. See Exercise 2.20 for other approaches.

2.16.2 Declarations and Actions

Each nonterminal and terminal is declared and associated with a data-type. For a terminal, the data-type is used to hold the values that are associated with the tokens that come from the lexer, *e.g.*, the values of numbers or names of identifiers. For a nonterminal, the type is used for the values that are built for the nonterminals during parsing (at reduce-actions), typically syntax trees.

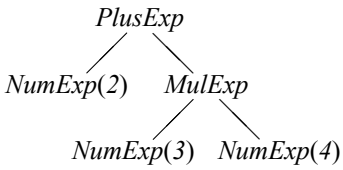
While, conceptually, parsing a string produces a syntax tree for that string, parser generators usually allow more control over what is actually produced. This is done by assigning an *action* to each production. The action is a piece of program text that is used to calculate the value of a production that is being reduced by using the values associated with the symbols on the right-hand side. For example, by putting appropriate actions on each production, the numerical value of an expression may be calculated as the result of parsing the expression. Indeed, compilers can be made such that the value produced during parsing is the compiled code of a program. For all but the simplest compilers it is, however, better to build a (possibly abstract) syntax tree during parsing and then later operate on this representation.

2.16.3 Abstract Syntax

The syntax trees described in Sect. 2.2.1 are not always optimally suitable for compilation. They contain a lot of redundant information: Parentheses, keywords used

for grouping purposes only, and so on. They also reflect structures in the grammar that are only introduced to eliminate ambiguity or to get the grammar accepted by a parser generator (such as left-factorisation or elimination of left-recursion). Hence, actions usually generate *abstract syntax trees* instead of precise syntax trees.

Abstract syntax keeps the essence of the structure of the text but omits the irrelevant details. An *abstract syntax tree* is a tree structure where each node corresponds to one or more nodes in the (concrete) syntax tree. For example, the concrete syntax tree shown in Fig. 2.13 may be represented by the following abstract syntax tree:



Here the names *PlusExp*, *MulExp* and *NumExp* may be constructors in a data-type, they may be elements from an enumerated type used as tags in a union-type or they may be names of subclasses of an *Exp* class. The names indicate which production is chosen, so there is no need to retain the subtrees that are implied by the choice of production, such as the subtree from Fig. 2.13 that holds the symbol +. Likewise, the sequence of nodes *Exp*, *Exp*2, *Exp*3, 2 at the left of Fig. 2.13 are combined to a single node *NumExp*(*2*) that includes both the choice of productions for *Exp*, *Exp*2 and *Exp*3 and the value of the terminal node. In short, each node in the abstract syntax tree corresponds to one or more nodes in the concrete syntax tree.

A designer of a compiler or interpreter has much freedom in the choice of abstract syntax. Some use abstract syntax that retain all of the structure of the concrete syntax trees plus additional positioning information used for error-reporting. Others prefer abstract syntax that contains only the information necessary for compilation or interpretation, skipping parentheses and other (for compilation or interpretation) irrelevant structure, like we did above.

Exactly how the abstract syntax tree is represented and built depends on the parser generator used. Normally, the action assigned to a production can access the values of the terminals and nonterminals on the right-hand side of a production through specially named variables (often called `$1`, `$2`, etc.) and produces the value for the node corresponding to the left-hand-side either by assigning it to a special variable (`$0`) or letting it be the value of an action expression.

The data structures used for building abstract syntax trees depend on the language. Most statically typed functional languages support tree-structured datatypes with named constructors. In such languages, it is natural to represent abstract syntax by one datatype per syntactic category (e.g., *Exp* above) and one constructor for each instance of the syntactic category (e.g., *PlusExp*, *NumExp* and *MulExp* above). In Pascal, a syntactic category can be represented by a variant record type and each instance as a variant of that. In C, a syntactic category can be represented by a union of structs, each struct representing an instance of the syntactic category and the union covering all possible instances. In object-oriented languages such as Java, a syntactic

category can be represented as an abstract class or interface where each instance in a syntactic category is a concrete class that implements the abstract class or interface. Alternatively, a syntactic category can be represented by an enumerated type with properties.

In most cases, it is fairly simple to build abstract syntax using the actions for the productions in the grammar. It becomes complex only when the abstract syntax tree must have a structure that differs nontrivially from the concrete syntax tree.

One example of this is if left-recursion has been eliminated for the purpose of making an LL(1) parser. The preferred abstract syntax tree will in most cases be similar to the concrete syntax tree of the original left-recursive grammar rather than that of the transformed grammar. As an example, the left-recursive grammar

$$\begin{aligned} E &\rightarrow E + \textbf{num} \\ E &\rightarrow \textbf{num} \end{aligned}$$

gets transformed by left-recursion elimination into

$$\begin{aligned} E &\rightarrow \textbf{num}\ E_* \\ E_* &\rightarrow + \textbf{num}\ E_* \\ E_* &\rightarrow \end{aligned}$$

Which yields a completely different syntax tree. We can use the actions assigned to the productions in the transformed grammar to build an abstract syntax tree that reflects the structure in the original grammar.

In the transformed grammar, E_* can return an abstract syntax tree with a *hole*. The intention is that this hole will eventually be filled by another abstract syntax tree:

- The second production for E_* (the empty production) returns just a hole.
- In the first production for E_*, the $+$ and **num** terminals are used to produce a tree for a plus-expression (i.e., a *PlusExp* node) with a hole in place of the first subtree. This tree is itself used to fill the hole in the tree returned by the recursive use of E_*, so the abstract syntax tree is essentially built outside-in. The result is a new tree with a hole.
- In the production for E, the hole in the tree returned by the E_* nonterminal is filled by a *NumExp* node with the number that is the value of the **num** terminal.

The best way of building trees with holes depends on the type of language used to implement the actions. Let us first look at the case where a functional language is used.

The actions shown below for the original grammar will build an abstract syntax tree similar to the one shown in the beginning of this section.

$$\begin{aligned} E &\rightarrow E + \textbf{num} && \{\ \texttt{PlusExp(\$1,NumExp(\$3))}\ \} \\ E &\rightarrow \textbf{num} && \{\ \texttt{NumExp(\$1)}\ \} \end{aligned}$$

We now want to make actions for the transformed grammar that will produce the same abstract syntax trees as the above actions for the original grammar will.

In functional languages, an abstract syntax tree with a hole can be represented by a function. The function takes as argument what should be put into the hole, and returns a syntax tree where the hole is filled with this argument. The hole is represented by the argument variable of the function. We can write this as actions to the transformed grammar:

$$
\begin{array}{ll}
E \rightarrow \textbf{num}\ E_* & \{\ \texttt{\$2(NumExp(\$1))}\ \} \\
E_* \rightarrow +\,\textbf{num}\ E_* & \{\ \texttt{λx.\$3(PlusExp(x,NumExp(\$2)))}\ \} \\
E_* \rightarrow & \{\ \texttt{λx.x}\ \}
\end{array}
$$

where an expression of the form $\lambda x.e$ is an anonymous function that takes x as argument and returns the value of the expression e. The empty production returns the identity function, which works like a top-level hole. The non-empty production for E_* applies the function \$3 returned by the E_* on the right-hand side to a subtree, hence filling the hole in \$3 by this subtree. The subtree itself has a hole x, which is filled when applying the function returned by the right-hand side. The production for E applies the function \$2 returned by E_* to a subtree that has no holes and, hence, returns a tree with no holes.

In SML, $\lambda x.e$ is written as `fn x => e`, in F# as `fun x -> e`, in Haskell as `\x -> e`, and in Scheme as `(lambda (x) e)`.

An imperative version of the actions in the original grammar could be

$$
\begin{array}{ll}
E \rightarrow E + \textbf{num} & \{\ \texttt{\$0 = PlusExp(\$1,NumExp(\$3))}\ \} \\
E \rightarrow \textbf{num} & \{\ \texttt{\$0 = NumExp(\$1)}\ \}
\end{array}
$$

In this setting, `NumExp` and `PlusExp` can be class constructors or functions that allocate and build nodes and return pointers to these. In most imperative languages, anonymous functions of the kind used in the above solution for functional languages can not be built, so holes must be an explicit part of the data-type that is used to represent abstract syntax. These holes will be overwritten when the values are supplied. E_* will, hence, return a record holding both an abstract syntax tree (in a field named `tree`) and a pointer to the hole that should be overwritten (in a field named `hole`). As actions (using C-style notation), this becomes

$E \rightarrow \textbf{num}\ E_*$
```
{ $2->hole = NumExp($1);
  $0 = $2.tree }
```
$E_* \rightarrow +\,\textbf{num}\ E_*$
```
{ $0.hole = makeHole();
  $3->hole = PlusExp($0.hole,NumExp($2));
  $0.tree = $3.tree }
```
$E_* \rightarrow$
```
{ $0.hole = makeHole();
  $0.tree = $0.hole }
```

where `makeHole()` creates a node that can be overwritten. This may look bad, but left-recursion removal is rarely needed when using LR-parser generators.

An alternative approach is to let the parser build an intermediate (semi-abstract) syntax tree from the transformed grammar, and then let a separate pass restructure the intermediate syntax tree to produce the intended abstract syntax. Some LL(1) parser generators can remove left-recursion automatically and will afterwards restructure the syntax tree so it fits the original grammar.

2.17 Properties of Context-Free Languages

In Sect. 1.9, we described some properties of regular languages. Context-free languages share some, but not all, of these.

For regular languages, deterministic (finite) automata cover exactly the same class of languages as nondeterministic automata. This is not the case for context-free languages: Nondeterministic stack automata do indeed cover all context-free languages, but deterministic stack automata cover only a strict subset. The subset of context-free languages that can be recognised by deterministic stack automata are called deterministic context-free languages. Deterministic context-free languages can be recognised by LR parsers (but not necessarily by SLR parsers).

We have noted that the basic limitation of regular languages is finiteness: A finite automaton can not count unboundedly, and hence can not keep track of matching parentheses or similar properties that require counting. Context-free languages are capable of such counting, essentially using the stack for this purpose. Even so, there are limitations: A context-free language can only keep count of one thing at a time, so while it is possible (even trivial) to describe the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ by a context-free grammar, the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not a context-free language. The information kept on the stack follows a strict LIFO order, which further restricts the languages that can be described. It is, for example, trivial to represent the language of palindromes (strings that read the same forwards and backwards) by a context-free grammar, but the language of strings that can be constructed by concatenating a string with itself is not context-free.

Context-free languages are, as regular languages, closed under union: It is easy to construct a grammar for the union of two languages given grammars for each of these. Context-free languages are also closed under prefix, suffix, subsequence and reversal. Indeed, a language that consists of all subsequences of a context-free language is actually a regular language. However, context-free languages are *not* closed under intersection or complement. For example, the languages $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m \mid m, n \geq 0\}$ and $\{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \mid m, n \geq 0\}$ are both context-free while their intersection $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not, and the complement of the language described by the grammar in Sect. 2.10 is not a context-free language.

2.18 Further Reading

Context-free grammars were first proposed as a notation for describing natural languages (e.g., English or French) by the linguist Noam Chomsky [4], who defined this as one of three grammar notations for this purpose. The qualifier "context-free" distinguishes this notation from the other two grammar notations, which were called "context-sensitive" and "unconstrained". In context-free grammars, derivation of a nonterminal is independent of the context in which the terminal occurs, whereas the context can restrict the set of derivations in a context-sensitive grammar. Unrestricted grammars can use the full power of a universal (Turing-complete) computer, so unrestricted grammars can represent all languages with decidable membership.

Context-free grammars are too weak to describe natural languages, but were adopted for defining the Algol 60 programming language [3], using a notation which is now called Backus-Naur form. Since then, variants of context-free grammars have been used for defining or describing almost all programming languages.

Some languages have been designed with specific parsing methods in mind: Pascal [6] was designed for LL(1) parsing while C [7] was originally designed to fit LALR(1) parsing. This property was lost in later versions of the language, which have more complex grammars.

Most parser generators are based on LALR(1) parsing, but some use LL parsing. An example of this is ANTLR [8].

"The Dragon Book" [2] tells more about parsing methods than the present book.

Several textbooks, e.g., [5] describe properties of context-free languages.

The methods presented here for rewriting grammars based on operator precedence uses only infix operators. If prefix or postfix operators have higher precedence than all infix operators, the method presented here will work (with trivial modifications), but if there are infix operators that have higher precedence than some prefix or postfix operators, it breaks down. A method for rewriting grammars with arbitrary precedences of infix, prefix and postfix operators to unambiguous form is presented in [1], along with a proof of correctness of the transformation.

2.19 Exercises

Exercise 2.1 Figures 2.7 and 2.8 show two different syntax trees for the string `aabbbcc` using Grammar 2.4. Draw a third, different syntax tree for `aabbbcc` using the same grammar, and show the left-derivation that corresponds to this syntax tree.

Exercise 2.2 Draw the syntax tree for the string `aabbbcc` using Grammar 2.9.

Exercise 2.3 Write an unambiguous grammar for the language of balanced parentheses, i.e., the language that contains (among other) the equences

ε (i.e., the empty string)
```
( )
( ( ) )
( ) ( )
( ( ) ( ( ) ) )
```

but *no* unbalanced sequences such as

```
(
)
) (
( ( )
( ) ( ) )
```

Exercise 2.4 Write grammars for each of the following languages:

a) All sequences of `a`s and `b`s that contain the same number of `a`s and `b`s (in any order).
b) All sequences of `a`s and `b`s that contain strictly more `a`s than `b`s.
c) All sequences of `a`s and `b`s that contain a different number of `a`s and `b`s.
d) All sequences of `a`s and `b`s that contain twice as many `a`s as `b`s.

Exercise 2.5 We extend the language of balanced parentheses from Exercise 2.3 with two symbols: `[` and `]`. `[` corresponds to exactly two normal opening parentheses and `]` corresponds to exactly two normal closing parentheses. A string of mixed parentheses is legal if and only if the string produced by replacing `[` by `((` and `]` by `))` is a balanced parentheses sequence. Examples of legal strings are

```
ε
( ) ( )
( ( ]
[ ]
[ ) ( ]
[ ( ] )
```

a) Write a grammar that recognises this language.
b) Draw the syntax trees for `[ ) ( ]` and `[ ( ] )`.

Exercise 2.6 Show that the grammar

$$\begin{aligned} A &\to -A \\ A &\to A - \mathbf{id} \\ A &\to \mathbf{id} \end{aligned}$$

is ambiguous by finding a string that has two different syntax trees.

Now make two different unambiguous grammars for the same language:

a) One where prefix minus binds stronger than infix minus.
b) One where infix minus binds stronger than prefix minus.

Show, using the new grammars, syntax trees for the string you used to prove the original grammar ambiguous. Show also fully reduced syntax trees.

Exercise 2.7 In Grammar 2.2, replace the operators $-$ and $/$ by $<$ and :. These have the following precedence rules:

$<$ is non-associative and binds less tightly than $+$ but more tightly than :.
: is right-associative and binds less tightly than any other operator.

Write an unambiguous grammar for this modified grammar using the method shown in Sect. 2.3.1. Show the syntax tree and the fully reduced syntax tree for $2 : 3 < 4 + 5 : 6 * 7$ using the unambiguous grammar.

Exercise 2.8 Extend Grammar 2.14 with the productions

$$\begin{array}{ll} \mathit{Exp} & \rightarrow \mathbf{id} \\ \mathit{Matched} & \rightarrow \end{array}$$

then calculate *Nullable* and *FIRST* for every production in the grammar. Add an extra start production as described in Sect. 2.9 and calculate *FOLLOW* for every nonterminal in the grammar.

Exercise 2.9 Calculate *Nullable*, *FIRST* and *FOLLOW* for the nonterminals A and B in the grammar

$$\begin{array}{l} A \rightarrow B\,A\,a \\ A \rightarrow \\ B \rightarrow b\,B\,c \\ B \rightarrow A\,A \end{array}$$

Remember to extend the grammar with an extra start production when calculating *FOLLOW*.

Exercise 2.10 Eliminate left-recursion from Grammar 2.2.

Exercise 2.11 Calculate *Nullable* and *FIRST* for every production in Grammar 2.23.

Exercise 2.12 Add a new start production $\mathit{Exp}' \rightarrow \mathit{Exp}\,\$$ to the grammar produced in Exercise 2.10 and calculate *FOLLOW* for all nonterminals in the resulting grammar.

Exercise 2.13 Make a LL(1) parser-table for the grammar produced in Exercise 2.12.

Exercise 2.14 Consider the following grammar for postfix expressions:

$$\begin{array}{l} E \rightarrow E\,E\,+ \\ E \rightarrow E\,E\,* \\ E \rightarrow \mathbf{num} \end{array}$$

a) Eliminate left-recursion in the grammar.
b) Do left-factorisation of the grammar produced in question a.
c) Calculate *Nullable*, *FIRST* for every production and *FOLLOW* for every nonterminal in the grammar produced in question b.
d) Make a LL(1) parse-table for the grammar produced in question b.

Exercise 2.15 Extend Grammar 2.12 with a new start production as shown in Sect. 2.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in Sect. 2.9.

Exercise 2.16 Make NFAs (as in Fig. 2.30) for the productions in Grammar 2.12 (after extending it as shown in Sect. 2.14) and add epsilon-transitions as in Fig. 2.31. Convert the combined NFA into an SLR DFA like the one in Fig. 2.33. Finally, add reduce and accept actions based on the *FOLLOW* sets calculated in Exercise 2.15.

Exercise 2.17 Extend Grammar 2.2 with a new start production as shown in Sect. 2.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in Sect. 2.9.

Exercise 2.18 Make NFAs (as in Fig. 2.30) for the productions in Grammar 2.2 (after extending it as shown in Sect. 2.14) and add epsilon-transitions as in Fig. 2.31. Convert the combined NFA into an SLR DFA like the one in Fig. 2.33. Add reduce actions based on the *FOLLOW* sets calculated in Exercise 2.17. Eliminate the conflicts in the table by using operator precedence rules as described in Sect. 2.15. Compare the size of the table to that from Exercise 2.16.

Exercise 2.19 Consider the grammar

$$
\begin{aligned}
T &\rightarrow T - > \ T \\
T &\rightarrow T \ * \ T \\
T &\rightarrow \textbf{int}
\end{aligned}
$$

where -> is considered a single terminal symbol.

a) Add a new start production as shown in Sect. 2.14.
b) Calculate $FOLLOW(T)$. Remember to add an extra start production.
c) Construct an SLR parser-table for the grammar.
d) Eliminate conflicts using the following precedence rules:

- * binds tighter than ->.
- * is left-associative.
- -> is right-associative.

Exercise 2.20 In Sect. 2.16.1 it is mentioned that user-defined operator precedences in programming languages can be handled by parsing all operators with a single fixed precedence and associativity and then using a separate pass to restructure the syntax tree to reflect the declared precedences. Below are two other methods that have been used for this purpose:

a) An ambiguous grammar is used and conflicts exist in the SLR table. Whenever a conflict arises during parsing, the parser consults a table of precedences to resolve this conflict. The precedence table is extended whenever a precedence declaration is read.
b) A terminal symbol is made for every possible precedence and associativity combination. A conflict-free parse table is made either by writing an unambiguous grammar or by eliminating conflicts in the usual way. The lexical analyser uses a table of precedences to assign the correct terminal symbol to each operator it reads.

Compare all three methods. What are the advantages and disadvantages of each method?

Exercise 2.21 Consider the grammar

$$
\begin{aligned}
A &\rightarrow a\ A\ a \\
A &\rightarrow b\ A\ b \\
A &\rightarrow
\end{aligned}
$$

a) Describe the language that the grammar defines.
b) Is the grammar ambiguous? Justify your answer.
c) Construct an SLR parse table for the grammar.
d) Can the conflicts in the table be eliminated without changing the language? Justify your answer.

Exercise 2.22 The following ambiguous grammar describes Boolean expressions:

$$
\begin{aligned}
B &\rightarrow \textbf{true} \\
B &\rightarrow \textbf{false} \\
B &\rightarrow B\ \vee\ B \\
B &\rightarrow B\ \wedge\ B \\
B &\rightarrow \neg\ B \\
B &\rightarrow (\ B\ )
\end{aligned}
$$

a) Given that negation ($\neg$) binds tighter than conjunction ($\wedge$) which binds tighter than disjunction ($\vee$) and that conjunction and disjunction are both right-associative, rewrite the grammar to be unambiguous.
b) Write a grammar that accepts the subset of Boolean expressions that are equivalent to **true** (i.e., tautologies). Hint: Modify the answer from question a) and add an additional nonterminal F for false Boolean expressions.

References

1. Aasa, A.: Precedences in specifications and implementations of programming languages. Theor. Comput. Sci. **142**(1), 3–26 (1995). http://dx.doi.org/10.1016/0304-3975(95)90680-J, http://www.sciencedirect.com/science/article/pii/030439759590680J

2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers; Principles. Addison-Wesley, Techniques and Tools (2007)
3. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vaquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language. Algol **60** (1963)
4. Chomsky, N.: Three models for the description of language. IRE Trans. Inf. Theory **IT-2**(3), 113–124 (1956)
5. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation, 2nd edn. Addison-Wesley (2001)
6. Jensen, K., Wirth, N.: Pascal User Manual and Report, 2nd edn. Springer-Verlag (1975)
7. Kerninghan, B.W., Ritchie, D.M.: The C Programming Language. Prentice-Hall (1978)
8. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages, 1st edn. Pragmatic Programmers, Pragmatic Bookshelf (2007)