Basics of Compiler Design

Extended edition

Torben Ægidius Mogensen

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN

Published through `lulu.com`.

`torbenm@diku.dk`

Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
DENMARK

Book homepage:
`http://www.diku.dk/~torbenm/Basics`

First published 2000
This edition: July 25, 2008

Contents

Chapter 3

Syntax Analysis

3.1 Introduction

Where lexical analysis splits the input 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 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.

[1]The name refers to the fact that derivation is independent of context.

We shall see 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 accepts a wider class of grammars. Sadly, neither of these work for all context-free grammars. Tools that handle all context-free grammars exist, but they can incur a severe speed penalty, which is why most parser generators restrict the class of input grammars.

3.2 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 non-terminals are chosen to denote the language described by the grammar. This is called the *start symbol* of the grammar. 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

$$N \rightarrow X_1 \ldots X_n$$

where N is a nonterminal and $X_1 \ldots X_n$ 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 $X_1 \ldots X_n$. In this setting, a terminal denotes a singleton set, just like alphabet characters in regular expressions. We will, when no confusion is likely, equate a nonterminal with the set of strings it denotes

Some examples:

$$A \rightarrow \mathsf{a}$$

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

$$A \rightarrow \mathsf{a}A$$

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

$$\begin{array}{lcl} B & \rightarrow & \\ B & \rightarrow & \mathtt{a}B \end{array}$$

where the first production indicates that the empty string is part of the set B. Compare this grammar with the definition of s^* in figure 2.1.

Productions with empty right-hand sides are called *empty productions*. These are sometimes 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 section 2.10, 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

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

The second production ensures that the as and bs are paired symmetrically around the middle of the string, ensuring that 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

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

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.

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

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

Figure 3.1: From regular expressions to context free grammars

Sometimes, 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

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

There are still four productions in the grammar, even though the arrow symbol $\rightarrow$ is only used twice.

3.2.1 How to write context free grammars

As hinted above, a regular expression can systematically be rewritten as a 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 figure 3.1. So, if we can think of a way of expressing a language as a regular expression, it is easy to make a grammar for it. However, we will also want to use grammars to describe non-regular languages. An example is the kind of arithmetic expressions

$$\begin{array}{lcl}
Exp & \rightarrow & Exp \texttt{+} Exp \\
Exp & \rightarrow & Exp \texttt{-} Exp \\
Exp & \rightarrow & Exp \texttt{*} Exp \\
Exp & \rightarrow & Exp \texttt{/} Exp \\
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \texttt{(} Exp \texttt{)}
\end{array}$$

Grammar 3.2: Simple expression grammar

that are part of most programming languages (and also found on electronic calculators). Such expressions can be described by grammar 3.2. Note that, as mentioned in section 2.10, the matching parentheses can't be described by regular expressions, as these can't "count" the number of unmatched opening parentheses at a particular point in the string. However, if we didn't have parentheses in the language, it could be described by the regular expression

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

Even so, the regular description isn't useful if you want operators to have different precedence, as it treats the expression as a flat string rather than as having structure. We will look at structure in sections 3.3.1 and 3.4.

Most constructions from programming languages are easily expressed by context free grammars. In fact, most modern languages are designed this way.

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 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 express properties of names used in other parts of the program.

Each syntactic category is denoted by a main nonterminal, *e.g.*, *Exp* from grammar 3.2. More nonterminals might be needed to describe a

$$\begin{array}{lcl} \mathit{Stat} & \rightarrow & \mathbf{id} := \mathit{Exp} \\ \mathit{Stat} & \rightarrow & \mathit{Stat}\ ;\ \mathit{Stat} \\ \mathit{Stat} & \rightarrow & \mathbf{if}\ \mathit{Exp}\ \mathbf{then}\ \mathit{Stat}\ \mathbf{else}\ \mathit{Stat} \\ \mathit{Stat} & \rightarrow & \mathbf{if}\ \mathit{Exp}\ \mathbf{then}\ \mathit{Stat} \end{array}$$

Grammar 3.3: Simple statement grammar

syntactic category or provide structure to it, as we shall see, and 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 3.3, which refers to the *Exp* nonterminal from grammar 3.2.

3.3 Derivation

So far, we have just appealed to intuitive notions of recursion when we describe the set of strings that a grammar produces. Since the productions are similar to recursive set equations, we might expect to use the techniques from section 2.6.1 to find the set of strings denoted by a grammar. However, though these methods in theory apply to infinite sets by considering limits of chains of sets, they are only practically useful when the sets are finite. Instead, we below introduce the concept of *derivation*. An added advantage of this approach 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 production in which 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

$$\begin{array}{llcll} 1. & \alpha N \beta & \Rightarrow & \alpha\gamma\beta & \text{if there is a production } N \rightarrow \gamma \\ 2. & \alpha & \Rightarrow & \alpha & \\ 3. & \alpha & \Rightarrow & \gamma & \text{if there is a } \beta \text{ such that } \alpha \Rightarrow \beta \text{ and } \beta \Rightarrow \gamma \end{array}$$

where α, β and γ are (possibly empty) sequences of grammar symbols (terminals and nonterminals). The first rule states that using a pro-

$$\begin{array}{lcl} T & \rightarrow & R \\ T & \rightarrow & \mathtt{a}T\mathtt{c} \\ R & \rightarrow & \\ R & \rightarrow & R\mathtt{b}R \end{array}$$

Grammar 3.4: Example grammar

duction 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].

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

Definition 3.1 *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 strings 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 3.4 generates the string `aabbbcc` by the derivation shown in figure 3.5. We have, for clarity, in each sequence of symbols underlined the nonterminal that is 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 doesn't matter. So, we might as well decide to always rewrite the leftmost nonterminal, as shown in figure 3.6.

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*.

3.3.1 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

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

$$\begin{array}{ll}
 & \underline{T} \\
\Rightarrow & \mathtt{a}\underline{T}\mathtt{c} \\
\Rightarrow & \mathtt{aa}\underline{T}\mathtt{cc} \\
\Rightarrow & \mathtt{aa}\underline{R}\mathtt{cc} \\
\Rightarrow & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{cc} \\
\Rightarrow & \mathtt{aa}\underline{R}\mathtt{bcc} \\
\Rightarrow & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{bcc} \\
\Rightarrow & \mathtt{aa}R\mathtt{b}\underline{R}\mathtt{b}R\mathtt{bcc} \\
\Rightarrow & \mathtt{aa}\underline{R}\mathtt{bb}R\mathtt{bcc} \\
\Rightarrow & \mathtt{aabb}\underline{R}\mathtt{bcc} \\
\Rightarrow & \mathtt{aabbbcc}
\end{array}$$

Figure 3.5: Derivation of the string `aabbbcc` using grammar 3.4

$$\begin{array}{ll}
 & \underline{T} \\
\Rightarrow & \mathtt{a}\underline{T}\mathtt{c} \\
\Rightarrow & \mathtt{aa}\underline{T}\mathtt{cc} \\
\Rightarrow & \mathtt{aa}\underline{R}\mathtt{cc} \\
\Rightarrow & \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{cc} \\
\Rightarrow & \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc} \\
\Rightarrow & \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{cc} \\
\Rightarrow & \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc} \\
\Rightarrow & \mathtt{aabb}\underline{R}\mathtt{b}R\mathtt{cc} \\
\Rightarrow & \mathtt{aabbb}\underline{R}\mathtt{cc} \\
\Rightarrow & \mathtt{aabbbcc}
\end{array}$$

Figure 3.6: Leftmost derivation of the string `aabbbcc` using grammar 3.4

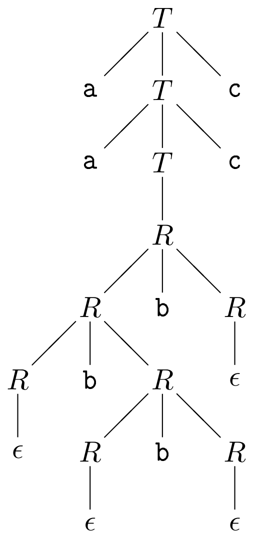


Figure 3.7: Syntax tree for the string `aabbbcc` using grammar 3.4

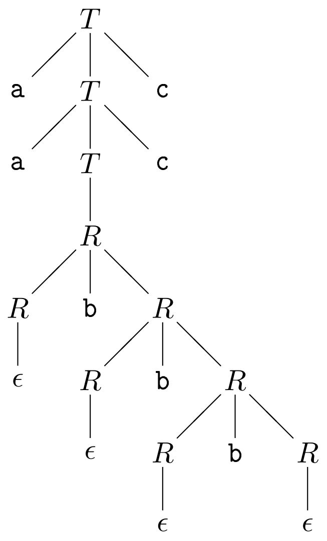


Figure 3.8: Alternative syntax tree for the string `aabbbcc` using grammar 3.4

$$\begin{array}{lcl} T & \rightarrow & R \\ T & \rightarrow & \mathtt{a}T\mathtt{c} \\ R & \rightarrow & \\ R & \rightarrow & \mathtt{b}R \end{array}$$

Grammar 3.9: Unambiguous version of grammar 3.4

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, form the derived string. If a nonterminal is rewritten using an empty production, an ϵ is shown as its child. This is also a leaf node, but is ignored when reading the string from the leaves of the tree.

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 figures 3.5 and 3.6 yield the same syntax tree, which is shown in figure 3.7.

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

For compilation, we do the derivation 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 doesn't matter when constructing a syntax tree, the *choice* of production for that 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, figure 3.8 shows an alternative syntax tree for the same string that was derived in figure 3.7.

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, ambiguity isn't 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 desireable 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, or external rules can be applied to decide which of the many possible syntax trees is the "right one". An unambiguous version of grammar 3.4 is shown in figure 3.9.

How do we know if a grammar is ambiguous? If we can find a string and show two alternative syntax trees for it, this is a proof of ambiguity. It may, however, be hard to find such a string and, when the grammar is unambiguous, even harder to show that this is the case. 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, any grammar that has a production of the form

$$N \rightarrow N\alpha N$$

where α is any sequence of grammar symbols, is ambiguous. This is, for example, the case with grammars 3.2 and 3.4.

We will, in sections 3.11 and 3.13, 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 also fail on certain unambiguous grammars, so they can not be used to prove 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. 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 (though they may impose different structures on the strings of the language).

Given two grammars, it would be nice to be able to tell if they are equivalent. Unfortunately, no known method is able to decide this in all cases, but, unlike ambiguity, it is not (at the time of writing) known if such a method may or may not theoretically exist. Sometimes, equivalence can be proven *e.g.* by induction over the set of strings that the grammars produce. The converse 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.

3.4 Operator precedence

As mentioned in section 3.2.1, we can describe traditional arithmetic expressions by grammar 3.2. Note that **num** is a terminal that denotes all integer constants and that, here, the parentheses are terminal symbols

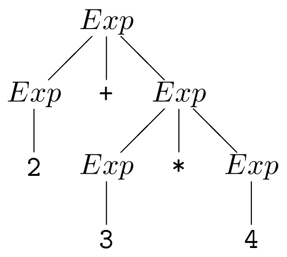


Figure 3.10: Preferred syntax tree for 2+3*4 using grammar 3.2

(unlike in regular expressions, where they are used to impose structure on the regular expressions).

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

$$Exp \rightarrow Exp + Exp$$

which has the form that in section 3.3.1 was claimed to imply ambiguity. This ambiguity is not 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 in figure 3.10, 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 possible way of resolving the ambiguity is to use precedence rules during syntax analysis to select among the possible syntax trees. Many parser generators allow this approach, as we shall see in section 3.15. However, some parsing methods require the grammars to be unambiguous, so we have to express the operator hierarchy in the grammar itself. To clarify this, we first define some concepts:

- 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 doesn't matter if we calculate from left to right or from right to left. However, to avoid ambiguity we have to choose one of these. By convention (and similarity to `-` and `/`) we choose to let these be left-associative as well. 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 (like Pascal), comparison operators (like $<$ or $>$) are non-associative, *i.e.*, you are not allowed to write $\mathtt{2} < \mathtt{3} < \mathtt{4}$.

3.4.1 Rewriting ambiguous expression grammars

If we have an ambiguous grammar

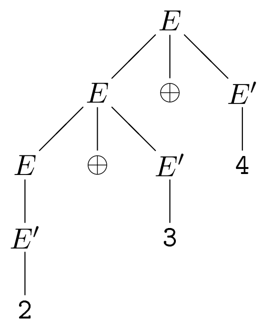
$$\begin{array}{lcl} E & \rightarrow & E \oplus E \\ E & \rightarrow & \mathbf{num} \end{array}$$

we can rewrite this to an unambiguous grammar that generates the correct structure. As this depends on the 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:

$$\begin{array}{lcl} E & \rightarrow & E \oplus E' \\ E & \rightarrow & E' \\ E' & \rightarrow & \mathbf{num} \end{array}$$

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



We get a slightly more complex syntax tree than in figure 3.10, but not enormously so.

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

$$\begin{array}{lcl} E & \rightarrow & E' \oplus E \\ E & \rightarrow & E' \\ E' & \rightarrow & \mathbf{num} \end{array}$$

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

$$\begin{array}{lcl} E & \rightarrow & E' \oplus E' \\ E & \rightarrow & E' \\ E' & \rightarrow & \mathbf{num} \end{array}$$

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 `-`:

$$\begin{array}{lcl} E & \rightarrow & E \texttt{+} E' \\ E & \rightarrow & E \texttt{-} E' \\ E & \rightarrow & E' \\ E' & \rightarrow & \mathbf{num} \end{array}$$

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

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

Grammar 3.11: Unambiguous expression grammar

$$\begin{array}{lcl}
E & \rightarrow & E \texttt{+} E' \\
E & \rightarrow & E' \oplus E \\
E & \rightarrow & E' \\
E' & \rightarrow & \textbf{num}
\end{array}$$

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 doesn't 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$\oplus$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 3.11 shows how these rules are used to make an unambiguous version of grammar 3.2. Figure 3.12 show the syntax tree for 2+3*4 using this grammar.

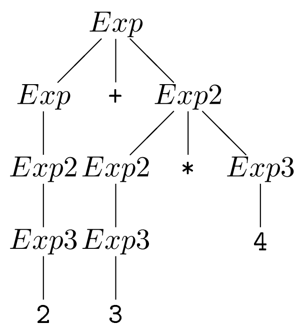


Figure 3.12: Syntax tree for `2+3*4` using grammar 3.11

3.5 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 section 3.4. 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 3.3. The problem is that it isn't clear how to parse, for example,

```
if p then if q then s1 else s2
```

According to the grammar, the `else` can equally well match either `if`. The usual convention 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` as a kind of right-associative operators, as this would make them group to the right, making an `if-then` match the closest `else`. However, the grammar transformations shown in section 3.4 can't directly be applied to grammar 3.3, as the productions for conditionals don't 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 3.13. This grammar also

$$
\begin{array}{lcl}
Stat & \rightarrow & Stat2 \; \texttt{;} \; Stat \\
Stat & \rightarrow & Stat2 \\
Stat2 & \rightarrow & Matched \\
Stat2 & \rightarrow & Unmatched \\
Matched & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Matched\ \texttt{else}\ Matched \\
Matched & \rightarrow & \mathbf{id} \texttt{:=} Exp \\
Unmatched & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Matched\ \texttt{else}\ Unmatched \\
Unmatched & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Stat2
\end{array}
$$

Grammar 3.13: Unambiguous grammar for statements

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 precedence rules during parsing. We shall look into this in section 3.15.

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

3.6 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 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 guess right. 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.

3.7 Predictive parsing

If we look at the left-derivation in figure 3.6, we see that, to the left of the rewritten nonterminals, there are only terminals. These terminals 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 on the next unmatched input symbol.

If we look at the third line in figure 3.6, we have already read two as 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

$$T \rightarrow \mathtt{a}T\mathtt{c}$$

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

$$T \rightarrow R$$

We are not quite as lucky in the next step. None of the productions for R start with a terminal symbol, so we can't immediately choose a production based on this. As the grammar (grammar 3.4) is ambiguous, it should not be a surprise that we can't always choose uniquely. If we instead use the unambiguous grammar (grammar 3.9) we can immediately choose the second production for R. When all the bs 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 on the next input symbol, we are able to do this kind of predictive parsing.

3.8 *Nullable* and *FIRST*

In simple cases, like the above, all but one of the productions for a nonterminal start with distinct terminals and the remaining production does not start with a terminal. However, the method can be applied also for grammers that don't have this property: Even if several productions start with nonterminals, we can choose among these if the strings these productions can derive begin with symbols from known disjoint sets. 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 3.2 *A symbol c is in FIRST(α) if and only if $\alpha \Rightarrow c\beta$ for some sequence β of grammar symbols.*

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 3.3 *A sequence α of grammar symbols is* Nullable *(we write this as Nullable(α)) if and only if $\alpha \Rightarrow \epsilon$.*

A production $N \rightarrow \alpha$ is called nullable if *Nullable(α)*. We describe calculation of *Nullable* by case analysis over the possible forms of sequences of grammar symbols:

Algorithm 3.4

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

where a *is a terminal, N is a nonterminal, α and β are sequences of grammar symbols and ϵ represents 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, but a nonterminal is nullable if any production has a nullable.

Note that this is a recursive definition since *Nullable* for a nonterminal is defined in terms of *Nullable* for its right-hand sides, which may contain that same nonterminal. We can solve this in much the same way that we solved set equations in section 2.6.1. We have, however, now booleans instead of sets and several equations instead of one. Still, the method is essentially the same: We have a set of boolean equations:

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

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 equations are satisfied. In section 2.6.1, 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 true, the result will also be more true (*i.e.*, it may stay unchanged, change from *false* to *true*, but never change from *true* to *false*).

If we look at grammar 3.9, we get these equations for nonterminals and right-hand sides:

$$\begin{array}{lcl} \mathit{Nullable}(T) & = & \mathit{Nullable}(R) \vee \mathit{Nullable}(\mathtt{a}T\mathtt{c}) \\ \mathit{Nullable}(R) & = & \mathit{Nullable}(\epsilon) \vee \mathit{Nullable}(\mathtt{b}R) \\ & & \\ \mathit{Nullable}(R) & = & \mathit{Nullable}(R) \\ \mathit{Nullable}(\mathtt{a}T\mathtt{c}) & = & \mathit{Nullable}(\mathtt{a}) \wedge \mathit{Nullable}(T) \wedge \mathit{Nullable}(\mathtt{c}) \\ \mathit{Nullable}(\epsilon) & = & \mathit{true} \\ \mathit{Nullable}(\mathtt{b}R) & = & \mathit{Nullable}(\mathtt{b}) \wedge \mathit{Nullable}(R) \end{array}$$

In a fixed-point calculation, we initially assume that *Nullable* is false for all nonterminals and use this as a basis for calculating *Nullable* for first the right-hand sides and then the nonterminals. We repeat recalculating these until there is no change between two iterations. Figure 3.14 shows the fixed-point iteration for the above equations. In each iteration, we first evaluate the formulae for the right-hand sides and then use the results of this to evaluate the nonterminals. The right-most column shows the final result.

We can calculate *FIRST* in a similar fashion to *Nullable*:

Right-hand side	Initialisation	Iteration 1	Iteration 2	Iteration 3
R	*false*	*false*	*true*	*true*
$\mathtt{a}T\mathtt{c}$	*false*	*false*	*false*	*false*
ϵ	*false*	*true*	*true*	*true*
$\mathtt{b}R$	*false*	*false*	*false*	*false*
Nonterminal				
T	*false*	*false*	*true*	*true*
R	*false*	*true*	*true*	*true*

Figure 3.14: Fixed-point iteration for calculation of *Nullable*

Algorithm 3.5

$$\begin{array}{lcl}
\mathit{FIRST}(\epsilon) & = & \emptyset \\
\mathit{FIRST}(\mathtt{a}) & = & \{\mathtt{a}\} \\
\mathit{FIRST}(\alpha\,\beta) & = & \left\{\begin{array}{ll} \mathit{FIRST}(\alpha) \cup \mathit{FIRST}(\beta) & \textit{if Nullable}(\alpha) \\ \mathit{FIRST}(\alpha) & \textit{if not Nullable}(\alpha) \end{array}\right. \\
\mathit{FIRST}(N) & = & \mathit{FIRST}(\alpha_1) \cup \ldots \cup \mathit{FIRST}(\alpha_n) \\
 & & \textit{where the productions for } N \textit{ are} \\
 & & N \rightarrow \alpha_1, \quad \ldots \quad, N \rightarrow \alpha_n
\end{array}$$

where a *is a terminal, N is a nonterminal, α and β are sequences of grammar symbols and ϵ represents the empty sequence of grammar symbols.*

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 anything in *FIRST*$(\beta)$ is also in *FIRST*$(\alpha\beta)$.

The set-equations are solved in the same general way as the boolean equations for *Nullable*, but since we work with sets, we initailly assume every set to be empty. For grammar 3.9, we get the following equations:

$$\begin{array}{lcl}
\mathit{FIRST}(T) & = & \mathit{FIRST}(R) \cup \mathit{FIRST}(\mathtt{a}T\mathtt{c}) \\
\mathit{FIRST}(R) & = & \mathit{FIRST}(\epsilon) \cup \mathit{FIRST}(\mathtt{b}R) \\
 & & \\
\mathit{FIRST}(R) & = & \mathit{FIRST}(R) \\
\mathit{FIRST}(\mathtt{a}T\mathtt{c}) & = & \mathit{FIRST}(\mathtt{a}) \\
\mathit{FIRST}(\epsilon) & = & \emptyset \\
\mathit{FIRST}(\mathtt{b}R) & = & \mathit{FIRST}(\mathtt{b})
\end{array}$$

Right-hand side	Initialisation	Iteration 1	Iteration 2	Iteration 3
R	$\emptyset$	$\emptyset$	$\{\mathtt{b}\}$	$\{\mathtt{b}\}$
$\mathtt{a}T\mathtt{c}$	$\emptyset$	$\{\mathtt{a}\}$	$\{\mathtt{a}\}$	$\{\mathtt{a}\}$
ϵ	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathtt{b}R$	$\emptyset$	$\{\mathtt{b}\}$	$\{\mathtt{b}\}$	$\{\mathtt{b}\}$
Nonterminal				
T	$\emptyset$	$\{\mathtt{a}\}$	$\{\mathtt{a}, \mathtt{b}\}$	$\{\mathtt{a}, \mathtt{b}\}$
R	$\emptyset$	$\{\mathtt{b}\}$	$\{\mathtt{b}\}$	$\{\mathtt{b}\}$

Figure 3.15: Fixed-point iteration for calculation of *FIRST*

The fixed-point iteration is shown in figure 3.15.

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 go through the motions of solving the equations. When working by hand, it is often useful to simplify the equations before the fixed-point iteration, *e.g.*, reduce $\mathit{FIRST}(\mathtt{a}T\mathtt{c})$ to $\{\mathtt{a}\}$.

3.9 Predictive parsing revisited

We are now ready to construct predictive parsers for a wider class of grammars: If the right-hand sides of the productions for a nonterminal have disjoint *FIRST* sets, we can use the next input symbol to choose among the productions.

In section 3.7, we picked the empty production (if any) on any symbol that was not in the *FIRST* sets of the non-empty productions for the same nonterminal. We must actually do this for any production that is *Nullable*. Hence, at most one production for a nonterminal may be nullable, as otherwise we would not be able to choose deterministically between the two.

We said in section 3.3.1 that our syntax analysis methods will detect ambiguous grammars. However, this isn't true with the method as stated above: We will get unique choice of production even for some ambiguous grammars, including grammar 3.4. The syntax analysis will in this case 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 for some 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 3.9 to

$$T \rightarrow \mathtt{a}T\mathtt{b}$$

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`.

Hence, we add to our construction of predictive parsers a test that will reject ambiguous grammars and those unambiguous grammars that can cause the parser to fail erroneously.

We have so far simply chosen a nullable production if and only if no other choice is possible. However, we should extend this to say that we choose a production $N \rightarrow \alpha$ on symbol c if one of the two conditions below are satisfied:

1) $c \in \mathit{FIRST}(\alpha)$.

2) *Nullable*(α) and c can validly follow N in a derivation.

This makes us choose nullable productions more often than before. This, in turn, leads to more cases where we can not choose uniquely, including the example above with the modified grammar 3.9 (since `b` can follow R in valid derivations) and all ambiguous grammars that are not caught by the original method.

3.10 *FOLLOW*

For the purpose of rejecting grammars that are problematical for predictive parsing, we introduce *FOLLOW* sets for nonterminals.

Definition 3.6 *A terminal symbol* `a` *is in FOLLOW(N) if and only if there is a derivation from the start symbol S of the grammar such that $S \Rightarrow \alpha N \mathtt{a} \beta$, 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.

To correctly handle end-of-string conditions, we want to detect if $S \Rightarrow \alpha N$, *i.e.*, if there are derivations where N can be followed by the end of input. It turns out to be easy 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 representing 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. A production

$$M \rightarrow \alpha N \beta$$

generates the constraint *FIRST*(β) $\subseteq$ *FOLLOW*(N), since β, obviously, can follow N. Furthermore, if *Nullable*(β) the production also generates the constraint
FOLLOW(M) $\subseteq$ *FOLLOW*(N) (note the direction of the inclusion). The reason is that, if a symbol c is in *FOLLOW*(M), then there (by definition) is a derivation $S' \Rightarrow \gamma M c \delta$. But since $M \rightarrow \alpha N \beta$ and β is nullable, we can continue this by $\gamma M c \delta \Rightarrow \gamma \alpha N c \delta$, so c is also in *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 into different αs, Ns and βs. For example, the production $A \rightarrow BcB$ generates the constraint $\{c\} \subseteq$ *FOLLOW*(B) by splitting after the first B and the constraint *FOLLOW*(A) $\subseteq$ *FOLLOW*(B) by "splitting" after the last B.

We solve the constraints in the following fashion:

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

The steps taken to calculate the follow sets of a grammar are, hence:

1. Add a new nonterminal $S' \rightarrow S\$$, where S is the start symbol for the original grammar. S' is the start symbol for the extended grammar.

2. For each nonterminal N, locate all occurrences of N on the right-hand sides of productions. For each occurrence do the following:

   2.1 Let β be the rest of the right-hand side after the occurrence of N. Note that β may be empty.

   2.2 Let $m = \mathit{FIRST}(\beta)$. Add the constraint $m \subseteq \mathit{FOLLOW}(N)$ to the set of constraints. If β is empty, you can omit the constraint, as it doesn't add anything.

   2.3 If $\mathit{Nullable}(\beta)$, find the nonterminal M at the left-hand side of the production and add the constraint $\mathit{FOLLOW}(M) \subseteq \mathit{FOLLOW}(N)$. If $M = N$, you can omit the constraint, as it doesn't add anything. Note that if β is empty, $\mathit{Nullable}(\beta)$ is true.

3. Solve the constraints using the following steps:

   3.1 Start with empty sets for $\mathit{FOLLOW}(N)$ for all nonterminals N (not including S').

   3.2 For each constraint of the form $m \subseteq \mathit{FOLLOW}(N)$ constructed in step 2.1, add the contents of m to $\mathit{FOLLOW}(N)$.

   3.3 Iterating until a fixed-point is reached, for each constraint of the form $\mathit{FOLLOW}(M) \subseteq \mathit{FOLLOW}(N)$, add the contents of $\mathit{FOLLOW}(M)$ to $\mathit{FOLLOW}(N)$.

We can take grammar 3.4 as an example of this. We first add the production

$$T' \rightarrow T\$$$

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

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

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 use the constraints that involve these *FIRST* sets:

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

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

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

Which is the final values for the *FOLLOW* sets.

If we return to the question of predictive parsing of grammar 3.4, we see that for the nonterminal R we should choose the empty production on the symbols in *FOLLOW(R)*, *i.e.*, $\{\$, \mathtt{c}, \mathtt{b}\}$ and choose the non-empty production on the symbols in *FIRST(R*b*R)*, *i.e.*, $\{\mathtt{b}\}$. Since these sets overlap (on the symbol 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.

3.11 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

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

If we can always choose a production uniquely by using these rules, this is called 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. A grammar that can be parsed 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 structure of a program, and a table-based approach that encodes the decision process in a table.

3.11.1 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 section 3.11. 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, figure 3.16 shows pseudo-code for a recursive descent parser for grammar 3.9. We have constructed this program by the following process:

We have first added a production $T' \rightarrow T\$$ 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 next input symbol anyway, so we can report an error if it isn't in *FIRST(T')*. This is shown in the function `parseT'`.

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(*`a`T`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 function `match` takes as argument a symbol, which it tests for equality with the next input symbol. If they are equal, the following symbol is read into the variable `next`. We assume `next` is initialised to the first input symbol before `parseT'` is called.

The program in figure 3.16 only checks if the input is valid. It can easily be extended to construct a syntax tree by letting the parse functions return the sub-trees for the parts of input that they parse.

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

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

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

Figure 3.16: Recursive descent parser for grammar 3.9

3.11.2 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.

The table is cross-indexed by nonterminal and terminal and contains for each such pair the production (if any) that is chosen for that nonterminal when that terminal is the next input symbol. This decision is made just as for recursive descent parsing: The production $N \rightarrow \alpha$ is in the table at (N,a) if a is in *FIRST(α)* or if both *Nullable(α)* and a is in *FOLLOW(N)*.

For grammar 3.9 we get the table shown in figure 3.17.

The program that uses this table is shown in figure 3.18. 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

	a	b	c	$
T'	$T' \rightarrow T\$$	$T' \rightarrow T\$$		$T' \rightarrow T\$$
T	$T \rightarrow \texttt{a}T\texttt{c}$	$T \rightarrow R$	$T \rightarrow R$	$T \rightarrow R$
R		$R \rightarrow \texttt{b}R$	$R \rightarrow$	$R \rightarrow$

Figure 3.17: LL(1) table for grammar 3.9

```
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),next) = empty then
    reportError
  else
    rhs := rightHandSide(table(top(stack),next)) ;
    pop(stack) ;
    pushList(rhs,stack)
```

Figure 3.18: Program for table-driven LL(1) parsing

with the next input symbol. If the table-entry is empty, we report an error. If 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 of these will be at the top of the stack.

As an example, figure 3.19 shows the input and stack at each step during parsing of the string `aabbbcc$` using the table in figure 3.17. The top of the stack is to the left.

The program in figure 3.18, like the one in figure 3.16, only checks if the input is valid. It, too, can be extended to build a syntax tree. This can be done by letting each nonterminal on the stack point to its node in the partially built syntax tree. When the nonterminal is replaced by one of its right-hand sides, nodes for the symbols on the right-hand side are added as children to the node.

3.11.3 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.

input	stack
aabbbcc\$	T'
aabbbcc\$	$T$\$
aabbbcc\$	a$T$c\$
abbbcc\$	$T$c\$
abbbcc\$	a$T$cc\$
bbbcc\$	$T$cc\$
bbbcc\$	$R$cc\$
bbbcc\$	b$R$cc\$
bbcc\$	$R$cc\$
bbcc\$	b$R$cc\$
bcc\$	$R$cc\$
bcc\$	b$R$cc\$
cc\$	$R$cc\$
cc\$	cc\$
c\$	c\$
\$	\$

Figure 3.19: Input and stack during table-driven LL(1) parsing

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 3.11). 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 the grammar used to describe it isn't.

3.12 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 grammars can be rewritten to allow LL(1) parsing. In these cases stronger parsing techniques must be used.

3.12.1 Eliminating left-recursion

As mentioned above, the unambiguous expression grammar (grammar 3.11) 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 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 direct left-recursion.

When we have a nonterminal with some left-recursive and some non-left-recursive productions, *i.e.*,

$$\begin{array}{lcl} N & \to & N\alpha_1 \\ & \vdots & \\ N & \to & N\alpha_m \\ N & \to & \beta_1 \\ & \vdots & \\ N & \to & \beta_n \end{array}$$

where the β_i do not start with N, we observe that this is equivalent to the regular expression $(\beta_1 \,|\, \dots \,|\, \beta_n)(\alpha_1 \,|\, \dots \,|\, \alpha_m)^*$. We can generate the same set of strings by the grammar

$$\begin{array}{lcl} N & \rightarrow & \beta_1 N' \\ & \vdots & \\ N & \rightarrow & \beta_n N' \\ N' & \rightarrow & \alpha_1 N' \\ & \vdots & \\ N' & \rightarrow & \alpha_m N' \\ N' & \rightarrow & \end{array}$$

where N' is a new nonterminal.

Note that, since the β_i do not start with N, there is no direct left-recursion in this grammar. There may, however, still be instances of indirect left-recursion. We will briefly look at indirect left-recursion in section 3.12.1.

Rewriting the grammar like above will change 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 describe. We will return to this in section 3.16.

As an example of left-recursion removal, we take the unambiguous expression grammar 3.11. This has left recursion in both *Exp* and *Exp*2, so we apply the transformation to both of these to obtain grammar 3.20. The resulting grammar 3.20 is now LL(1).

Indirect left-recursion

The transformation shown in section 3.12.1 only serves in the simple case where there is no *indirect left-recursion*. Indirect left-recursion can have several faces:

1. There are productions

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

$$\begin{array}{lcl}
Exp & \rightarrow & Exp2\ Exp' \\
Exp' & \rightarrow & \texttt{+}\ Exp2\ Exp' \\
Exp' & \rightarrow & \texttt{-}\ Exp2\ Exp' \\
Exp' & \rightarrow & \\
Exp2 & \rightarrow & Exp3\ Exp2' \\
Exp2' & \rightarrow & \texttt{*}\ Exp3\ Exp2' \\
Exp2' & \rightarrow & \texttt{/}\ Exp3\ Exp2' \\
Exp2' & \rightarrow & \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & \texttt{(}\ Exp\ \texttt{)}
\end{array}$$

Grammar 3.20: Removing left-recursion from grammar 3.11

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. 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. We will not go into this here, as in practise most cases of left-recursion are direct left-recursion. Details can be found in [4].

3.12.2 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 3.3 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. Seeleft-recursion-elimination grammar 3.21. In this grammar[3], we can uniquely choose one of the productions for *Stat* based on one input token.

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

$$
\begin{array}{lll}
\mathit{Stat} & \rightarrow & \mathbf{id} := \mathit{Exp} \\
\mathit{Stat} & \rightarrow & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Stat}\ \mathit{Elsepart} \\
\\
\mathit{Elsepart} & \rightarrow & \texttt{else}\ \mathit{Stat} \\
\mathit{Elsepart} & \rightarrow &
\end{array}
$$

Grammar 3.21: Left-factorised grammar for conditionals

For most grammars, combining productions with common prefix will solve the problem. However, in this particular example the grammar still isn't LL(1): We can't 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 shouldn't be a surprise to us, since, after all, the grammar is ambiguous and ambiguous grammars can't be LL(1). The equivalent unambiguous grammar (grammar 3.13) can't easily be rewritten to a form suitable for LL(1), so in practice grammar 3.21 is used anyway and the conflict is handled by choosing the non-empty production for *Elsepart* whenever the symbol `else` is encountered, as this gives the desired behaviour of letting an `else` match the nearest `if`. Very few LL(1) conflicts caused by ambiguity can be removed in this way, however, without also changing the language recognized by the grammar. For example, operator precedence ambiguity can not be resolved by deleting conflicting entries in the LL(1) table.

3.12.3 Construction of LL(1) parsers summarized

1. Eliminate ambiguity

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:

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

This choice is encoded either in a table or a recursive-descent program.

3.13 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.

A class of bottom-up methods for parsing called *LR parsers* exist which accept a much larger class of grammars (though still not all grammars). The main advantage of LR parsing is that less rewriting is required to get a grammar in acceptable form, but there are also languages for which there exist LR-acceptable grammars but no LL(1) grammars. Furthermore, as we shall see in section 3.15, LR parsers allow external declaration of operator precedences for resolving ambiguity instead of requiring the grammar itself to be unambiguous.

We will look at a simple form of LR-parsing called SLR parsing. While most parser generators use a somewhat more complex method called LALR(1) parsing, we limit the discussion to SLR for the following reasons:

- It is simpler.
- In practice, LALR(1) handles few grammars that are not also handled by SLR.
- When a grammar is in the SLR class, the parse-tables produced by SLR are identical to those produced by LALR(1).
- Understanding of SLR principles is sufficient to know how a grammar should be rewritten when a LALR(1) parser generator rejects it.

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 right-derivation is built.

LR parsers are table-driven bottom-up parsers and use two kinds of “actions” involving the input stream and a stack:

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

reduce: On the stack, a number of symbols that are identical to the right-hand side of a production are replaced by the left-hand side of that production. Contrary to LL 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.

When all of the input is read, the stack will have a single element, which will be the start symbol of the grammar.

LR parsers are also called *shift-reduce parsers*. 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 achieve this, we construct a DFA. Conceptually, this DFA reads the contents of the stack, starting from the bottom. If the DFA is in an accepting state when it reaches the top of the stack, it will cause reduction by a production that is determined by the state and the next input symbol. If the DFA is not in an accepting state, it will cause a shift. Hence, at every step, the action can be determined by letting the DFA read the stack from bottom to top.

Letting the DFA read the entire stack at every action is not very efficient, so, instead, we keep track of the DFA state every time we push an element on the stack, storing the state as part of the stack element.

When the DFA has indicated a shift, the course of action is easy: We get the state from the top of the stack and follow the transition marked with the next input symbol to find the next DFA state.

If the DFA indicated a reduce, we pop the right-hand side of the production off the stack. We then read the DFA state from the new stack top. When we push the nonterminal that is the left-hand side of the production, we make a transition from this DFA state on the nonterminal.

With these optimisations, the DFA only has to inspect a terminal or nonterminal at the time it is pushed on the stack. At all other times, it just need to read the DFA state that is stored with the stack element. Hence, we can forget about what the actual symbols are as soon as the DFA has made the transition. There is, thus, no reason to keep the symbols on the stack, so we let a stack element just contain the DFA state. We still use the DFA to determine the next action, but it now only needs to look at the current state (stored at the top of the stack) and the next input symbol (at a shift action) or nonterminal (at a reduce action).

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:	Read next input symbol, push state n on the stack.
go n:	Push state n on the stack.
reduce p:	Reduce with the production numbered p.
accept:	Parsing has completed successfully.
error:	A syntax error has been detected.

Note that the current state is always found at the top of the stack. *Shift* and *reduce* actions are found when a state is cross-indexed with a terminal symbol. *Go* actions are found when a state is cross-indexed with a nonterminal. *Go* actions are only used immediately after a reduce, but we can't put them next to the *reduce* actions in the table, as the destination state of a *go* depends on the state on top of the stack *after* the right-hand side of the reduced production is popped off: A *reduce* in the current state is immediately followed by a *go* in the state that is found when the stack is popped.

An example SLR table is shown in figure 3.22. The table has been produced from grammar 3.9 by the method shown below in section 3.14. The actions have been abbreviated to their first letters and *error* is shown as a blank entry.

The algorithm for parsing a string using the table is shown in figure 3.23. As written, the 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: Each stack element holds (in addition to the state number) a portion of a syntax tree. When doing a *reduce* action, a new (partial) syntax tree is built by using the nonterminal from the reduced production as root and the syntax trees attached to the popped-off stack elements as children. The new tree is then attached to the stack element that is pushed.

Figure 3.24 shows an example of parsing the string `aabbbcc` using the table in figure 3.22. The stack grows from left to right.

3.14 Constructing SLR parse tables

An SLR parse table has as its core a DFA. Constructing this DFA from the grammar is not much different from constructing a DFA from a regular expression as shown in chapter 2: We first construct an NFA using techniques similar to those in section 2.4 and then convert this into a DFA using the construction shown in section 2.5.

	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		

Figure 3.22: SLR table for grammar 3.9

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

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

    accept:   terminate with success

    error:    reportError
endloop
```

Figure 3.23: Algorithm for SLR parsing

input	stack	action
`aabbbcc$`	0	s3
`abbbcc$`	03	s3
`bbbcc$`	033	s4
`bbcc$`	0334	s4
`bcc$`	03344	s4
`cc$`	033444	r3 ($R \rightarrow$) ; g6
`cc$`	0334446	r4 ($R \rightarrow \mathtt{b}R$) ; g6
`cc$`	033446	r4 ($R \rightarrow \mathtt{b}R$) ; g6
`cc$`	03346	r4 ($R \rightarrow \mathtt{b}R$) ; g2
`cc$`	0332	r1 ($T \rightarrow R$) ; g5
`cc$`	0335	s7
`c$`	03357	r2 ($T \rightarrow \mathtt{a}T\mathtt{c}$) ; g5
`c$`	035	s7
`$`	0357	r2 ($T \rightarrow \mathtt{a}T\mathtt{c}$) ; g1
`$`	01	accept

Figure 3.24: Example SLR parsing

$$
\begin{array}{rlcl}
0: & T' & \rightarrow & T \\
1: & T & \rightarrow & R \\
2: & T & \rightarrow & \mathtt{a}T\mathtt{c} \\
3: & R & \rightarrow & \\
4: & R & \rightarrow & \mathtt{b}R
\end{array}
$$

Grammar 3.25: Example grammar for SLR-table construction

However, before we do this, we extend the grammar with a new starting production. Doing this to grammar 3.9 yields grammar 3.25.

The next step is to make an NFA for each production. This is done exactly like in section 2.4, treating both terminals and nonterminals as alphabet symbols. The accepting state of each NFA is labelled with the number of the corresponding production. The result is shown in figure 3.26. Note that we have used the optimised construction for ϵ (the empty production) as shown in figure 2.6.

The NFAs in figure 3.26 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

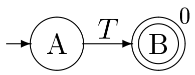
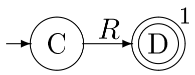
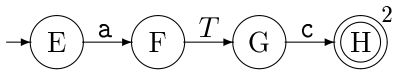
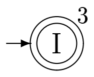
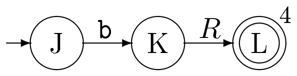
Production	NFA
$T' \rightarrow T$	
$T \rightarrow R$	
$T \rightarrow \texttt{a}T\texttt{c}$	
$R \rightarrow$	
$R \rightarrow \texttt{b}R$	

Figure 3.26: NFAs for the productions in grammar 3.25

state	epsilon-transitions
A	C, E
C	I, J
F	C, E
K	I, J

Figure 3.27: Epsilon-transitions added to figure 3.26

after a reduction, whereby some elements of the stack (corresponding to the right-hand side of a production) are replaced by a nonterminal (corresponding to the left-hand side of that production). However, before we can do this, the symbols that form the right-hand side must be on the stack.

To achieve this we must, whenever a transition by a nonterminal is possible, also allow transitions on the symbols on the right-hand side of a production for that nonterminal so these eventually can be reduced to the nonterminal. We do this by adding epsilon-transitions to the NFAs in figure 3.26: Whenever there is a transition from state s to state t on a nonterminal N, we add epsilon-transitions from s to the initial states of all the NFAs for productions with N on the left-hand side. Adding these graphically to figure 3.26 would make a very cluttered picture, so instead we simply note the transitions in a table, shown in figure 3.27.

Together with these epsilon-transitions, the NFAs in figure 3.26 form

DFA	NFA	Transitions				
state	states	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					

Figure 3.28: SLR DFA for grammar 3.9

a single, combined NFA. This NFA has the starting state A (the starting state of the NFA for the added start production) and an accepting state for each production in the grammar. We must now convert this NFA into a DFA using the subset construction shown in section 2.5. Instead of showing the resulting DFA graphically, we construct a table where transitions on terminals are shown as *shift* actions and transitions on nonterminals as *go* actions. This will make the table look similar to figure 3.22, except that no *reduce* or *accept* actions are present yet. Figure 3.28 shows the DFA constructed from the NFA made by adding epsilon-transitions in 3.27 to figure 3.26. The set of NFA states that forms each DFA state is shown in the second column of the table in figure 3.28. We will need these below for adding *reduce* and *accept* actions, but once this is done we will not need then anymore, and we can remove then from the final table.

To add *reduce* and *accept* actions, we first need to compute the *FOLLOW* sets for each nonterminal, as described in section 3.10. For purpose of calculating *FOLLOW*, we add yet another extra start production: $T'' \rightarrow T'\$$, to handle end-of-text conditions as described in section 3.10. This gives us the following result:

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

We then 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 on production 0 (the extra start production that was added before constructing the NFA) is written as *accept*.

In figure 3.28, 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. We add this at the symbol \$ (*FOLLOW(T′)*). 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 figure 3.22.

Figure 3.29 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. For each state s that 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. Convert the combined NFA to a DFA. Use the starting state of the NFA for the production added in step 1 as the starting state for the combined NFA.

5. Build a table cross-indexed by the DFA states and grammar symbols (terminals including \$ and nonterminals). Add *shift* actions at transitions on terminals and *go* actions on transitions on nonterminals.

6. Calculate *FOLLOW* for each nonterminal. For this purpose, we add one more start production: $S'' \rightarrow S'\$$.

7. When a DFA state contains an NFA state that accepts the right-hand side of the production numbered p, add *reduce* p at all symbols in *FOLLOW(N)*, where N is the nonterminal on the left of production p. If production p is the production added in step 1, the action is *accept* instead of *reduce* p.

Figure 3.29: Summary of SLR parse-table construction

3.14.1 Conflicts in SLR parse-tables

When *reduce* actions are added to SLR parse-tables, we might add one to a place where there is already a *shift* action, or we may add *reduce* actions for several different productions to the same place. When either of this happens, 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 may occur in the same place.

Conflicts are often caused by ambiguous grammars, but (as is the case for LL-parsers) even some non-ambiguous grammars may 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 sections 3.4 and 3.5. Alternatively, operator precedence declarations may be used to disambiguate an ambiguous grammar, as we shall see in section 3.15.

But even unambiguous grammars may in some cases generate conflicts in SLR-tables. In some cases, it is still possible to rewrite the grammar to get around the problem, but in a few cases the language simply isn't SLR. Rewriting an unambiguous grammar to eliminate conflicts is somewhat of an art. Investigation 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 isn't a problem for SLR-parsers, as it is for LL(1)-parsers.

3.15 Using precedence rules in LR parse tables

We saw in section 3.12.2, 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 3.2 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 section 3.4.1.
- 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 3.2 will generate several conflicts:

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

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.

1) This conflict arises from expressions like `a+b+c`. After having read `a+b`, the next input symbol is a `+`. 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 `+` is left-associative, we prefer the first of these options and hence eliminate the shift-action from the table and keep the reduce-action.
2) The offending expressions here have the form `a*b+c`. Since we want multiplication to bind stronger than addition, we, again, prefer reduction over shifting.
3) In expressions of the form `a+b*c`, we, as before, want multiplication to group stronger, so we do 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 a left-associative operator conflicts with itself and is likewise handled 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. The operator associated with a reduce-action is the operator used in the production that is reduced.

b) If the conflict is between operators of the same priority, the associativity (which must be the same, as noted in section 3.4.1) of the operators is used: If the operators are left-associative, the shift-action is eliminated and the reduce-action retained. 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.

c) If there are several operators with declared precedence in the production that is used in a reduce-action, the last of these is used to determine the precedence of the reduce-action.[4]

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 (section 3.5) can also be eliminated using precedence rules: Giving `else` a higher precedence than `then` or giving them the same precedence and making them right-associative will handle the problem, as either of these will make the parser shift on `else` instead of reducing $Stat \rightarrow$ `if` Exp `then` $Stat$ when this is followed by `else`.

Not all conflicts should be eliminated by precedence rules. Excessive use of precedence rules may cause the parser to accept only a subset

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

of the intended language (*i.e.*, if a necessary action is eliminated by a precedence rule). So, unless you know what you are doing, you should limit the use of precedence declarations to operators in expressions.

3.16 Using LR-parser generators

Most LR-parser generators use an extended version of the SLR construction called LALR(1). In practice, however, there is little difference between these, so a LALR(1) parser generator can be used with knowledge of SLR only.

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).

3.16.1 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).

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 reduced production 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 some kind of syntax representation during parsing and then later operate on this representation.

3.16.2 Abstract syntax

The syntax trees described in section 3.3.1 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, *abstract syntax* is commonly used.

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 figure 3.12 may be represented by the following abstract syntax tree:

$$\begin{tikzcd}
 & \mathit{PlusExp} \arrow[dl, dash] \arrow[dr, dash] & & \\
\mathit{NumExp(2)} & & \mathit{MulExp} \arrow[dl, dash] \arrow[dr, dash] & \\
 & \mathit{NumExp(3)} & & \mathit{NumExp(4)}
\end{tikzcd}$$

Here the names *PlusExp*, *MulExp* and *NumExp* may be constructors in a data-type or they may be elements from an enumerated type used as tags in a union-type. The names indicate which production is chosen, so there is no need to keep the subtrees that are implied by the choice of production, such as the subtree from figure refexpression-tree2 that holds the symbol +. Likewise, the sequence of nodes Exp, $Exp2$, $Exp3$, 2 at the left of figure refexpression-tree2 are combined to a single node *NumExp(2)* that includes both the choice of productions for Exp, $Exp2$ and $Exp3$ and the value of the terminal node.

A compiler designer has much freedom in the choice of abstract syntax. Some use abstract syntax that retain alls 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, skipping parenthesis and other (for this purpose) irrelevant structure.

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 return value of the action.

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, each 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.

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 non-trivially 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

$$\begin{array}{lcl} E & \rightarrow & E + \mathbf{num} \\ E & \rightarrow & \mathbf{num} \end{array}$$

gets transformed by left-recursion elimination into

$$\begin{array}{lcl} E & \rightarrow & \mathbf{num}\, E' \\ E' & \rightarrow & + \mathbf{num}\, E' \\ E' & \rightarrow & \end{array}$$

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' should 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' 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 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.

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

We now want to make actions for the transformed grammar that will produce the same abstract syntax trees as this 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:

$$\begin{array}{lll l} E & \rightarrow & \textbf{num}\, E' & \{\ \texttt{\$2(NumExp(\$1))}\ \} \\ E' & \rightarrow & + \,\textbf{num}\, E' & \{\ \lambda\texttt{x.\$3(PlusExp(x,NumExp(\$2)))}\ \} \\ E' & \rightarrow & & \{\ \lambda\texttt{x.x}\ \} \end{array}$$

where $\lambda x.e$ is a nameless 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 Haskell as `\x -> e` and in Scheme as `(lambda (x) e)`.
The imperative version of the actions in the original grammar is

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

In this setting, `NumExp` and `PlusExp` aren't constructors but functions that allocate and build node and return pointers to these. Unnamed functions of the kind used in the above solution for functional languages can not be built in most imperative languages, 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

$$\begin{array}{lcll} E & \rightarrow & \textbf{num}\, E' & \{ \texttt{\$2->hole = NumExp(\$1);} \\ & & & \;\;\texttt{\$0 = \$2.tree} \} \\ E' & \rightarrow & + \textbf{num}\, E' & \{ \texttt{\$0.hole = makeHole();} \\ & & & \;\;\texttt{\$3->hole = PlusExp(\$0.hole,NumExp(\$2));} \\ & & & \;\;\texttt{\$0.tree = \$3.tree} \} \\ E' & \rightarrow & & \{ \texttt{\$0.hole = makeHole();} \\ & & & \;\;\texttt{\$0.tree = \$0.hole} \} \end{array}$$

This may look bad, but when using LR-parser generators, left-recursion removal is rarely needed, and parser generators based on LL(1) often do left-recursion removal automatically and transform the actions appropriately.

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.

3.16.3 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

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 .`

Figure 3.30: Textual representation of NFA states

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 conflicts, it will tell in which state in the table these occur. 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 the NFA corresponds to a position in a production in the grammar. If we, for example, look at the NFA states in figure 3.26, these would be written as shown in figure 3.30. Note that a '.' is used to indicate the position of the state in the production. State 4 of the table in figure 3.28 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.

If all efforts to get a grammar through a parser generator fails, a practical solution may be to change the grammar so it 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).

Some 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 typical solution is to parse all operators using the same precedence and then restructure the syntax tree afterwards, but see also exercise 3.20.

3.17 Properties of context-free languages

In section 2.10, 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.

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. 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 repeating a string twice 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, the language consisting of all subsequences of a context-free language is actually regular. 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.

3.18 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 [13], 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 computer, so these represent all computable languages.

Context-free grammars are actually too weak to describe natural languages, but were adopted for defining the Algol60 programming language [15]. Since then, variants of this notation has been used for defining or describing almost all programming languages.

Some languages have been designed with specific parsing methods in mind: Pascal [19] has been designed for LL(1) parsing while C [23] was originally designed to fit LALR(1) parsing, but this property was lost in subsequent versions of the language.

Most parser generators are based on LALR(1) parsing, but a few use LL(1) parsing. An example of this is ANTLR (`http://www.antlr.org/`).

"The Dragon Book" [4] tells more about parsing methods than the present book.

Several textbooks exist that describe properties of context-free languages, e.g., [18].

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 handling arbitrary precedences of infix, prefix and postfix operators is presented in [1].

Exercises

Exercise 3.1

Figures 3.7 and 3.8 show two different syntax trees for the string `aabbbcc` using grammar 3.4. Draw a third, different syntax tree for `aabbbcc` using the same grammar and show the left-derivation that corresponds to this syntax tree.

Exercise 3.2

Draw the syntax tree for the string `aabbbcc` using grammar 3.9.

Exercise 3.3

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

ϵ (*i.e.* the empty string)
()
(())
()()
(()(()))

but none of the following

(
)
)(
(()
()())

Exercise 3.4

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 3.5

We extend the language of balanced parentheses from exercise 3.3 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 3.6

Show that the grammar

$$\begin{array}{lcl} A & \rightarrow & -A \\ A & \rightarrow & A - \mathbf{id} \\ A & \rightarrow & \mathbf{id} \end{array}$$

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 the syntax trees using the new grammars for the string you used to prove the original grammar ambiguous.

Exercise 3.7

In grammar 3.2, 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 section 3.4.1. Show the syntax tree for $2 : 3 < 4 + 5 : 6 * 7$ using the unambiguous grammar.

Exercise 3.8

Extend grammar 3.13 with the productions

$$\begin{array}{lcl} Exp & \rightarrow & \textbf{id} \\ Matched & \rightarrow & \end{array}$$

then calculate *Nullable* and *FIRST* for every production in the grammar.

Add an extra start production as described in section 3.10 and calculate *FOLLOW* for every nonterminal in the grammar.

Exercise 3.9

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

$$\begin{array}{lcl} A & \rightarrow & BAa \\ A & \rightarrow & \\ B & \rightarrow & bBc \\ B & \rightarrow & AA \end{array}$$

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

Exercise 3.10

Eliminate left-recursion from grammar 3.2.

Exercise 3.11

Calculate *Nullable* and *FIRST* for every production in grammar 3.20.

Exercise 3.12

Add a new start production $Exp' \rightarrow Exp\,\$$ to the grammar produced in exercise 3.10 and calculate *FOLLOW* for all nonterminals in the resulting grammar.

Exercise 3.13

Make a LL(1) parser-table for the grammar produced in exercise 3.12.

Exercise 3.14

Consider the following grammar for postfix expressions:

$$\begin{array}{lcl} E & \rightarrow & E\,E\,+ \\ E & \rightarrow & E\,E\,* \\ E & \rightarrow & \textbf{num} \end{array}$$

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 3.15

Extend grammar 3.11 with a new start production as shown in section 3.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in section 3.10.

Exercise 3.16

Make NFAs (as in figure 3.26) for the productions in grammar 3.11 (after extending it as shown in section 3.14) and show the epsilon-transitions as in figure 3.27. Convert the combined NFA into an SLR DFA like the one in figure 3.28. Finally, add reduce and accept actions based on the *FOLLOW* sets calculated in exercise 3.15.

Exercise 3.17

Extend grammar 3.2 with a new start production as shown in section 3.14 and calculate *FOLLOW* for every nonterminal. Remember to add an extra start production for the purpose of calculating *FOLLOW* as described in section 3.10.

Exercise 3.18

Make NFAs (as in figure 3.26) for the productions in grammar 3.2 (after extending it as shown in section 3.14) and show the epsilon-transitions as in figure 3.27. Convert the combined NFA into an SLR DFA like the one in figure 3.28. Add reduce actions based on the *FOLLOW* sets calculated in exercise 3.17. Eliminate the conflicts in the table by using operator precedence rules as described in section 3.15. Compare the size of the table to that from exercise 3.16.

Exercise 3.19

Consider the grammar

$$\begin{array}{lcl} T & \rightarrow & T \texttt{ -> } T \\ T & \rightarrow & T * T \\ T & \rightarrow & \textbf{int} \end{array}$$

where `->` is considered a single terminal symbol.

a) Add a new start production as shown in section 3.14.

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 3.20

In section 3.16.3 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 3.21

Consider the grammar

$$\begin{array}{lcl} A & \rightarrow & a\ A\ a \\ A & \rightarrow & b\ A\ b \\ A & \rightarrow & \end{array}$$

a) Describe the language that the grammar defines.

b) Is the grammar ambiguous? Justify your answer.

c) Construct a SLR parse table for the grammar.

d) Can the conflicts in the table be eliminated?

Exercise 3.22

The following ambiguous grammar describes boolean expressions:

$$\begin{array}{lcl} B & \rightarrow & \textbf{true} \\ B & \rightarrow & \textbf{false} \\ B & \rightarrow & B \lor B \\ B & \rightarrow & B \land B \\ B & \rightarrow & \neg B \end{array}$$

a) Given that negation ($\neg$) binds tighter than conjunction ($\land$) which binds tighter than disjunction ($\lor$) and that conjunction and disjunction are both right-associative, rewrite the grammar to be unambiguous.

b) Write a grammar that generates only true boolean expressions. Hint: Use the answer from question a) and add an additional nonterminal for false boolean expressions.

Bibliography

[1] A. Aasa. Precedences in specification and implementations of programming languages. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 183–194. Springer Verlag, 1991.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996. Also downloadable from `http://mitpress.mit.edu/sicp/full-text/sicp/book/`.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers; Principles, Techniques and Tools.* Addison-Wesley, 2007. Newer edition of [5].

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers; Principles, Techniques and Tools.* Addison-Wesley, 1986. Rereleased in extended form as [4].

[6] Hassan Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction.* MIT Press, 1991.

[7] John R. Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann, 2001.

[8] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[9] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[10] H. Bratman. An alternative form of the 'uncol' diagram. *Communications of the ACM*, 4(3):142, 1961.

[11] Preston Briggs. *Register Allocation via Graph Coloring, Tech. Rept. CPC-TR94517-S.* PhD thesis, Rice University, Center for Research on Parallel Computation, Apr. 1992.

[12] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 1(4):481–494, 1964.

[13] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, IT-2(3):113–124, 1956.

[14] J. Earley and H. Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13:607–617, 1970.

[15] Peter Naur (ed.). Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[16] John Hatcliff, Torben Mogensen, and Peter Thiemann (Eds.). *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science.* Springer Verlag, 1999.

[17] Raymond J. Hookway and Mark A. Herdeg. Digital fx!32: Combining emulation and binary translation.
`http://www.cs.tufts.edu/comp/150PAT/optimization/DTJP01PF.pdf`, 1997.

[18] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation, 2nd ed.* Addison-Wesley, 2001.

[19] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report (2nd ed.).* Springer-Verlag, 1975.

[20] Neil D. Peyton Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[21] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages – A Tutorial.* Prentice Hall, 1992.

[22] J. P. Keller and R. Paige. Program derivation with verified transformations – a case study. *Communications in Pure and Applied Mathematics*, 48(9–10), 1996.

[23] B. W. Kerninghan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[24] M. E. Lesk. Lex: a Lexical Analyzer Generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J., 1975.

[25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd ed.* Addison-Wesley, Reading, Massachusetts, 1999.

[26] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[27] Robin Milner. A theory of type polymorphism in programming. *Journal of Computational Systems Science*, 17(3):348–375, 1978.

[28] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[29] Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors. *The essence of computation: complexity, analysis, transformation.* Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[30] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[31] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[32] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[33] David A. Patterson and John L. Hennessy. *Computer Organization & Design, the Hardware/Software Interface.* Morgan Kaufmann, 1998.

[34] Vern Paxson. Flex, version 2.5, a fast scanner generator. `http://www.gnu.org/software/flex/manual/html_mono/flex.html`, 1995.

[35] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[36] Niklaus Wirth. The design of a pascal compiler. *Software - Practice and Experience*, 1(4):309–333, 1971.

Index