Course Script

INF 5110: Compiler construction

INF5110, spring 2022

Martin Steffen

Contents

Chapter 4 Parsing

What is it about?

Learning Targets of this Chapter

1. top-down and bottom-up parsing
2. look-ahead
3. first and follow-sets
4. different classes of parsers (LL, LALR)

Contents

4.1 Introduction

We have introduced the general concept of context-free grammars, but apart from shortly mentioning some simple sanitary conditions on the form of the grammar and apart from discussing ambiguity as a unwelcome property of a grammar, we did not really impose restrictions on grammars. That will change in this chapter. For parsing, the full expressivity of CFGs and languages is not used. Instead one works with specific subclasses and/or restrictions on the representations of grammars.

Since almost by definition, the *syntax* of a language are those aspects covered by a context-free grammar, a **syntax error** thereby is a violation of the grammar, something the parser has to detect and deal with. Given a CFG, typically given in BNF resp. implemented by a tool supporting a BNF variant, the parser (in combination with the lexer) must generate an AST *exactly* for those programs that adhere to the grammar and must *reject* all others. One says, the parser *recognizes* the given grammar. An important practical part when rejecting a program is to generate a meaningful *error message*, giving hints about locations of the error and potential reasons. In the most minimal way, the parser should inform the programmer where the parser tripped, i.e., telling how far, from left to right, it was able to proceed and informing where it stumbled: "parser error in line xxx/at character position yyy"). One typically has higher expectations for a real parser than just the line number, but that's the basics.

It may be noted that also the subsequent phase, the *semantic analysis*, which takes the abstract syntax tree as input, may report errors. Those are no longer syntax errors but a more complex kind of errors. One typical kind of error in the semantic phase is a *type error*. Also there, the minimal requirement is to indicate the probable location(s) where

the error occurs. To do so, in basically all compilers, the nodes in an abstract syntax tree will contain information concerning the position in the original file the resp. node corresponds to (like line-numbers, character positions). If the parser would not add that information into the AST, the semantic analysis would have no way to relate potential errors it finds to the original, concrete code in the input. Remember: the compiler goes in *phases*, and once the parsing phase is over, there's no going back to scan or parse the file *again*.

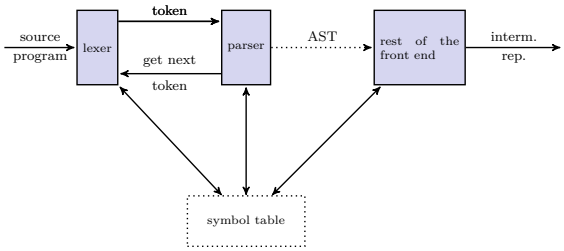


Figure 4.1: Lexer, parser, and the rest

The symbol table is an important data structure, shared between phases. It will be covered (conceptually) in a later chapter. It deals with keeping information about "symbols". For instance, names or identifiers. It's like a "data base" with relevant information in connection with, say, variable names. It allows to efficiently store, update, and look-up information about, for instance, variables. Relevant information about variables includes its type, once the type checker has figured out which type is connected to which variable.

One could store that information also in the abstract syntax tree, stuffing it directly to the nodes, but that's impractical for various reasons. We will discuss symbol tables later.

A central distinction for parsers is **top-down vs. bottom-up** parsing. All parsers (together with lexers) work from *left-to-right*. A sequence of tokens is syntactically correct if there exists a parse-tree (and for an unambigous grammar, there exists at most one). While parser eats through the token stream, it grows, i.e., builds up (at least conceptually) the parse tree.

> A **bottom-up** parser grows parse trees from the leaves to the root. A **top-down** parser grows parse trees from the root to the leaves.

4.1.1 Parsing restricted classes of CFGs

Parsing should better be efficient, or to say it differently: there is no need in making the grammar so complex that it requires inefficient parsing techniques. So, the full complexity of CFLs is not really needed in practice, and by far not. Concerning the need (or the lack of need) for very expressive grammars, one should consider also the following: if a parser has trouble to figure out if a program has a syntax error or not (perhaps using back-tracking), probably humans will have similar problems. So better keep it simple, not just for the sake of the efficiency of parsing, but also for the understandibility of programs. And time in a compiler may be better spent elsewhere (optimization, semantical analysis).

When talking about classification, one can distinguish classes of CF languages vs. classes of CF grammars, but both hang together. For example, the condition to be free of left-recursion-freedom is a condition on a grammar. Things hand together, though. Ambiguity is initially a condition of grammars. A context-free languages is *inherently* ambiguous if there is no unambiguous grammar for it, only ambigous one.

Similarly for the other conditions. For instance, a CF language is top-down parseable, if there exists a grammar that allows top-down parsing ...

As far as grammar classes are conceerned, *maaaany* have been proposed & studied, including their relationships, parsearbility etc. Figure 4.1 mentions a few classes that we will look at in the lecture. The main distinction is that between top-down and bottom-up parsing.

top-down parsing	bottom-up parsing
LL(0)	LR(0)
LL(1)	LR(1)
recursive descent	SLR
	LALR(1)

Table 4.1: Classes of CFG grammars/languages

In practice, one can also classify according to the parser or parser generating tools. A grammar or language that can be parsed by *yacc*, or a grammar which is parseable by parser combinators etc.

Actually, the class LALR(1) is the class that is covered by yacc-style tools.

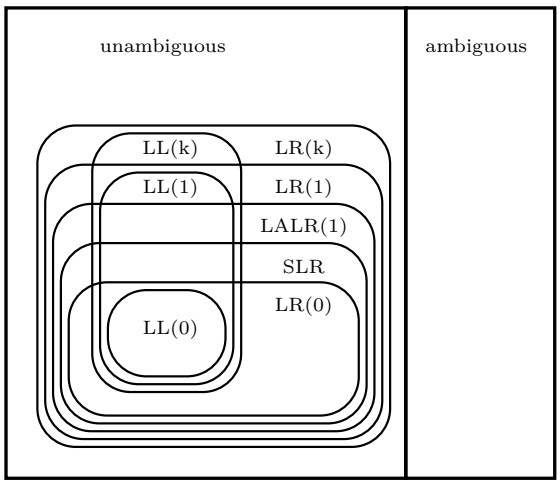


Figure 4.2: Relationship of some grammar (not language) classes (taken from [3])

Figure 4.2 shows the hierarchy of some grammar classes in terms of expressiveness. The picture is about *grammars*, not *languages* (though a similar hierarchy also exists there). The picture is intended to mean that the hierarchy is *real* in the following sense. For instance: it's clear that a grammar that can be parsed top-down with one look-ahead, can be parsed also with a look-ahead of two. In other words, the set of LL(1) grammars

is contained in the set of LL(2) grammars, etc. But also, LL(1) is *properly* contained in LL(2), i.e., there exist an LL(2) which is *not* an LL(1) grammar. Likewise for the other shown inclusions. So the hierarchies are infinite. In other words: adding look-ahead really increases the expressivity of a grammar class.

Another thing one can see on the picture: with the same amout on look-ahead, bottom-up parsing (the LR-kind) is more expressive that top-down parsing (the LL-kind).

The picture mentions SLR and LALR(1), positioned between LR(0) and LR(1). The LR- and the LL-parsers realizes, in a way, the concept of bottom-up, resp. top-down parsing in a pure way. SLR and LALR(1) are variations on top-down parsing, adding some extra tricks and checks to LR-parsers, in particular adding them to LR(0) (giving SLR) resp. modifying LR(1) (giving LALR(1)) to get some extra expressiveness, without actually paying the price of more look-ahead. We will discuss those classes. As said, LALR(1) is the one underlying yacc and friends, so it's a practically important class

4.2 Top-down parsing

Top-down parsing is one of the two classes of parsers we cover; the other one being bottom-up. Parsing is about the following: given a sequence of tokens, i.e., a word where the symbol consist of tokens, determine whether that word can be *derived* in the given grammar. If the word is derivable, i.e., if there exists a derivation, that means, there exists a parse tree. In all but degenerated cases, each word can be derived in different ways; in particular, there can be different order of expansion steps in a derivation of a word. However, for an unambiguous grammar, there will be only one derivation tree or *parse tree* for each derivable word; that's the definition of being unambiguous, as discussed. Top-down parsing builds up this parse tree in a top-down manner, starting with the root of the tree, which is the grammar's start symbol, a non-terminal.

Maybe saying, that top-down parsers "build" the parse-tree is a bit misleading. Normally parsers (top-down or otherwise) don't build or create a data structure called parse-tree. They build an AST, resembling more or less closely the parse-tree. The parse tree is more conceptually traversed, rather than built. Still, the traversal is top-down and the AST is likewise built top-down. For bottom-up parsing, the building and traversal works bottom-up, obviously.

Before we look at how that conceptually works in more detail, let's first have a look at a schematic view of the the parser as machine or automaton (see Figure 4.3).

Note, the machine works on a sequence of *tokens* not characters.

The picture hints at a fact about the kind of "machines" are needed for parsing. For scanning, the machines were finite-state automata; see the very similar corresponding schematic picture for lexers resp. finite state automata we have seen earlier. Best For finite-state automata, as far as expressivity is concerned, there's no difference between the non-deterministic and deterministic variants of FSAs, as discussed.

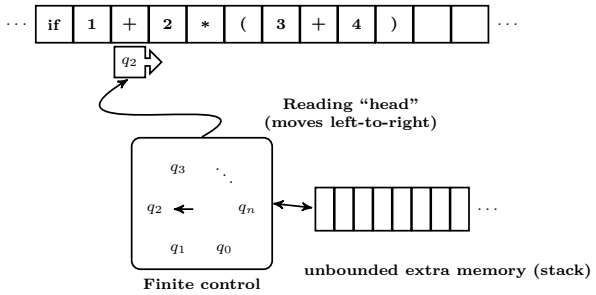


Figure 4.3: Schematic view on a "parser machine"

For recognizing context-free languages, i.e., for parsing, we don't formally define the corresponding machines. They are called **push-down** automata, which are finite-state automata equipped with an additional component of undbounded memory. The memory can be read and written to, but only in stack-like manner. So the extra memory is organized as stack (not as random-access memory). Without going into details, a pushdown automaton, works on the input stream the same way a finite state automaton does, eating it strictly from left to right, and moving along between its states accordingly like a FSA does. The additional power comes from the stack: the machine can consult the stack, actually *only* the top of the stack, it cannot look deeper. That top-most symbol can be used to make decisions into which state to move in step (popping it off). Additionally, symbols can be *pushed* to the stack in a transition.

Such machines can accept CFGs, and each CFGs can be accepted by a pushdown machine. One can analogously as for FSAs define when such a machine is deterministic and when not. However, not everything carries over. For example, non-deterministic stack-machines are strickly more expressive than deterministic ones (which implies, there's no hope to carry over the powerset construction we encountered earlier to such machines now).

As said, we *don't* look into the pushdown automaton formalism. The reason is simple. Parsing in practice does not cover all CFGs in their generality, already ambiguity needs to be excluded. NB: Ambiguity is connected to the notion of non-determinism, though the connection is not one-to-one. Anyway, we are dealing with restricted CFGs. In the section here, for top-down parsing, we will later see how it can be done *recursively*. This is also called *recursive descent* parsing. Using recursion makes, of course, implicitly uses a *stack*, though we don't explicitly operate with a push-down machine. Later, for bottom-up parsing, we will work with DFAs equipped with a stack. Those will be some push-down machines, but we will be more interested in the working and construction of particular ones, rather than the general machine model, so neither there we will introduce the general concept of pushdown automaton.

Remark 4.2.1 (Tail recursion)**.** A final side-remark: the need for a stack, the need for recursion, etc., all ultimately comes from the fact that context-free grammar definitions are *recursive*. We have touched upon the fact, in connection with Chomsky's hierarchy, that regular laguages can be seen as a special resctricted case of context-free languages. Respectively, regular expressions can be (easily) representend as context-free grammars. The connection between context-free grammars and regular expressions can be best seen

in connection with the *extended* version. In some examples when discussing the macro-mechanism for extended regular expression, we stressed that there macros need to be in a "cascade". Numbers can be defined using the simpler concept of digits. Signed number, as still a more complex concept, can be defined using numbers. Floats, in turn, were defined using signed numbers. So there's a clear hierarchy. In particular, there is *no* mutual recursion in the definitions. Allowing that would then be a context-free grammar in BNF.

Does that mean, there is no recursion in regular expressions? Not explicitly (making use of "variables" or non-terminals). Implicitly there is a form of recursion, namely in the form of Kleene-star operation. One can easily expand a regular expression r^* into

$$A \to rA \mid \epsilon \ .$$

(with A not occuring in r, of course). Alternatively, one can use Ar instead. The first production is right-recursive, with A occuring at the end of the right-hand side of the production. The alternative representation is an example of a right-recursive rule. And that's exectly the restriction: regular languages can be captured by the special case of left-recursive grammars (alternatively right-recursive).

But where is the stack? If we look at the recursive descent parsing, to which we come soon, the recusive-ness in the grammar is "translated" to correspoinding recursive procedure calls. In the case of a right-recursive grammar, that will be translated to code, where the calls are *only* done at the *end* of a procedure. That is known as tail-recursion. As you may know, tail-recursion can be done *without* a stack, actually by a simple "goto" or by a loop. That's even a known compiler optimization: replacing tail recursive code by iterative, it's called *tail-call optimization* (TCO). □

The slide version contains some big series of overlays, showing the derivation. This derivation process is not reproduced here (resp. only a few slides later as some big array of steps).

Next we illustrate parsing and derivations using expressions, more precisely a version of the expression grammar using *factors* and *terms*. We encountered such a formulation before, in Example **??**. It was there introduced in discussing the concept of **precedence cascade.** The grammar here, while using factors and terms, is not the mentionned one from Example **??**, and actually we are at the moment not interested in discussing precedence and associativity (again). What is special about the formulation now is that the grammar avoids *left-recursion.* That's important for top-down parsing and will be dicussed later.

Example 4.2.2 (Derivation (factors and terms)). Let's assume the following grammar:

$$\begin{array}{rcl} exp & \to & term\ exp' \\ exp' & \to & addop\ term\ exp' \mid \epsilon \\ addop & \to & + \mid - \\ term & \to & factor\ term' \\ term' & \to & mulop\ factor\ term' \mid \epsilon \\ mulop & \to & * \\ factor & \to & (\ exp\ ) \mid \textbf{number} \end{array} \tag{4.1}$$

Below we show a derivation for the word **number + number * (number + number)**, which could for instance represent the expression $1 + 2 * (4 + 5)$.

$$
\begin{array}{ll}
\underline{exp} & \Rightarrow \\
\underline{term}\ exp' & \Rightarrow \\
\underline{factor}\ term'\ exp' & \Rightarrow \\
\cancel{\textbf{number}}\ term'\ exp' & \Rightarrow \\
\textbf{number}\ \underline{term'}\ exp' & \Rightarrow \\
\textbf{number}\ \cancel{\epsilon}\ exp' & \Rightarrow \\
\textbf{number}\ \underline{exp'} & \Rightarrow \\
\textbf{number}\ \underline{addop}\ term\ exp' & \Rightarrow \\
\textbf{number}\ \cancel{+}\ term\ exp' & \Rightarrow \\
\textbf{number} + \underline{term}\ exp' & \Rightarrow \\
\textbf{number} + \underline{factor}\ term'\ exp' & \Rightarrow \\
\textbf{number} + \cancel{\textbf{number}}\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number}\ \underline{term'}\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number}\ \underline{mulop}\ factor\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number}\ \cancel{*}\ \underline{factor}\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ exp\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * \cancel{(}\ exp\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \underline{exp}\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \underline{term}\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \underline{factor}\ term'\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \cancel{\textbf{number}}\ term'\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number}\ \underline{term'}\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number}\ \cancel{\epsilon}\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number}\ \underline{exp'}\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number}\ \underline{addop}\ term\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number}\ \cancel{+}\ term\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \underline{term}\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \underline{factor}\ term'\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \cancel{\textbf{number}}\ term'\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ \underline{term'}\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ \cancel{\epsilon}\ exp'\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ \underline{exp'}\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ \cancel{\epsilon}\ )\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ \cancel{)}\ term'\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ )\ \underline{term'}\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ )\ \cancel{\epsilon}\ exp' & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ )\ \underline{exp'} & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ )\ \cancel{\epsilon} & \Rightarrow \\
\textbf{number} + \textbf{number} * (\ \textbf{number} + \textbf{number}\ ) &
\end{array}
$$

The derivation starts with the start symbol *expr* and derives the target word of terminals in a finite number of derivation steps. As for the notation used here. The part underlines is the place where the next step takes place. I.e., it indicates the the occurrence of *non-terminal* where the corresponding grammar production is used. We ~~*crosse out*~~ the *terminal* or *token* is considered treated parser. That is when the parser moves on. This will later be r implemented as a `match` or `eat` procedure.

The derivation is a **left-most** derivation. The derivation corresponds to the parse tree of Figure 4.4.

The tree does no longer contain information, which parts have been expanded first. In particular, the information that we have concretely done a left-most derivation when building up the tree in a top-down fashion is not part of the tree (as it is not important). The tree is an example of a *parse tree* as it contains information about the derivation process using rules of the grammar. □

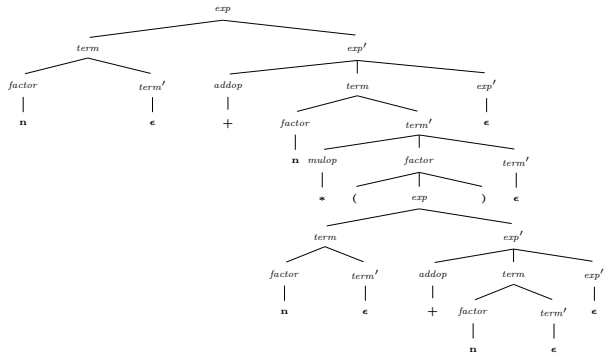


Figure 4.4: Parse tree

In the derivation of the previous example, we mentioned that the derivation is a left-most derivation, without really explaining what that is. It will be easy enough, but before we define it, let's discuss aspects in connection with derivations, which ultimately motivates that definition.

In all but uninteresting cases, the language of a grammar is infinite. I.e., there are infinitely many different derivations, generating infinitely many different words. But parsing is not about genrating words, it's about recognizing a specific one (or rejecting it). So the derivation and the tree is meant not a "free" application of rules in a process of derivation (or expansion, reduction, generation ...) but a reduction of start symbol towards the **target word of terminals**, in the example say

$$exp \;\Rightarrow^* \; \mathbf{1 + 2 * (3 + 4)}$$

i.e.: input stream of tokens "guides" the derivation process, at least it fixes the target. But: how much "guidance" does the target word give?

We know already that for ambigious grammars, there are words that have more than one parse tree. On that case, the input does not give enough guidance to *determine* the derivation tree (and by implication, neither the derivation).

In an unambigous grammar, the input *determines* the parse three, but what about the derivation? In general there are different derivations for each given word, also for unambigous grammars. Some degenerated grammars may have exactly one derivation per word, but it's an aberration and unintesting. In general a parse tree corresponds to multiple derivations, as it does not contain information about the **order** in which the reduction steps have been done that resulted in the tree.

So, degenerated grammars aside, a certain amount of **non-determinism** concerning which step the parser chooses to do next is unavoidable in the process to derive a given word, also for unambiguous grammars.

When performing a parse, and if there are choices to be made, there are three points we want to consider. 1) What kind of decisions are there? 2) If there are alternatives, does it actually matter which one to take? 3 On what can the parser base its decision?

For the following discussions and concepts, remember the notational conventions from Convention **??** on page ??

Coming back to the issues concerning choice: The issues hang together, but let's start with the first one. Given a target word to derive, one can distinguish **two principle sources of non-determinism** in the parsing process.

The general form of a non-empty derivation of a word w, involving at least one step, is as follows:

$$\begin{array}{rcl} S & \Rightarrow^* & \alpha \\ & = & \alpha_1\ A\ \alpha_2 \Rightarrow \alpha_1\ \beta\ \alpha_2 \\ & \Rightarrow^* & w \end{array} \tag{4.2}$$

where the step in focus uses the production $A \to \beta$

There are **2 choices to make**:[a]

1. **where**, i.e., on **which occurrence of a non-terminal** in α to apply a production
2. **which production** to apply (for the chosen non-terminal).

[a] The nitpicking mind could make the argument, that there are 3 choices involved: which non-terminal, which rule, and where in the sentential form. Fair enough, if one makes decisions in that order. But our exposition presents it as two decisions: where in the sentential form, which fixes the non-terminal, and which rule.

Note that α_1 and α_2 may contain non-terminals, including further occurrences of A. However, the word w contains terminals, only.

Let's look at the **where**-flavor of non-determinism in the derivation. Given a partial derivation $S \Rightarrow^* \alpha$, the sentential form may generally contain different non-terminals, including that some terminals may occur multiple times. At any rate, the parser need to decide which non-terminal should be expanded next. The good new is: **it does not matter!** It's easy to see that expanding one non-terminal in one derivation step leaves all other non-terminal untouched and does not prevent them from being expanded later, leading to the same derivation tree. That's the essence of being *context-free*: expanding a non-terminal is independed form the shape of the surrounding letters.

Since it does not matter where to expand, the parser machine could in principle pick randomly. But for the sake of making a consistent, determinisitic decision, it could uniformely take the first non-terminal in α. Or perhaps the last one.

Definition 4.2.3 (Left-most and right-most derivation)**.** A *left-most* derivation of a word or a sentential form α is a derivation $S \Rightarrow^* \alpha$ where each step expands the *left-most* non-terminal in the current sentential form. A *right-most* derivation chooses analogously the right-most non-terminal. We denote a left-most derivation step by $\Rightarrow_l$ and a right-most one by $\Rightarrow_r$.

Let's use the symbols as in Convention **??**. Assume a production a production $A \to \beta$ is used in a derivation $S \Rightarrow^* \alpha_1\ A\ \alpha_2 \Rightarrow \alpha_1\ \beta\ \alpha_2 \Rightarrow^* w$. For a *left-most* derivation, the derivation is more specifically of the form

$$S \Rightarrow_l^* w_1\ A\ \alpha_2 \Rightarrow_l w_1\ \beta\ \alpha_2 \Rightarrow_l^* w\ . \tag{4.3}$$

The prefix w_1 contains only terminals, making A the left-most non-terminal.

As mentioned, the deriviation in Example 4.2.2 was left-most.

That was the *easy* part of non-determinism and partly addressed also the second mentioned issue: for the where-form it does not matter which choice is made, so the parser can, for instance follow a left-most strategy. We can spell it out in a lemma (without proof):

Lemma 4.2.4 (Left or right, who cares)**.** $S \Rightarrow_l^* w$ *iff* $S \Rightarrow_r^* w$ *iff* $S \Rightarrow^* w$.

When saying that it does not matter where to expand, that's correct with respect to the resulting parse-tree(s) for a given word. But it does not mean that all parsers are working with left-most derivations. It will turn out that bottom-up parser will work with right-most derivations. The reasons will be explained when we come there.

Now, what about the other form of non-deterministic choice: *which* rule to use for the chosen non-terminal? Choosing different rules will lead to different parse-trees. Assuming a unambigous grammar, there is only one tree for a given word, however. So if in a partial derivation there is a choice between two productions and assuming that the word of terminals is ultimately derivable, **there is only right choice,** all others would be wrong. The core problem of parsing is: how to come to that right decision?

Let's assume then for the discussion an unambigous one and let's focus on the "which-production-to-apply" non-determinism, because it's the only interesting one.

So assume a derivation in the form of equation (4.3) Now, in the process of parsing, what can influence the decision of choosing $A \to \beta$ over, say $A \to \gamma$?. On the most extreme and unrealistict case, **all** of the target word w.

The parser works itself to the tokens step by step, so if the parser takes the whole word into account, it means, it takes into account the part it has **parsed already**, as well as the part **still to come.** Parts of the input that constitute the future rest of the word is the so-called **look-ahead.**

A parser that bases its decisions on the past as well as all of the future is a parser with an unbounded look-ahead (and unbounded memory to remember arbitrary long past). Long look-aheads are unrealistic, because that's inefficent. Realistic parser uses only a quite short fixed amount of look-ahead, maybe as short as one token. But we can speculate: a parser with unbounded memory and unbounded look-ahead, with all information at hand, can this powerful parser make always the right decision (for an unambiguous grammar)?

> It may seem plausible that a (top-down) parser, constructing a tree following the derivation as from equation (4.3) when eating through the input and with full information, the past as well as the future of its input, can make the right decision at each point, for a given unambigous grammar. However: *that's not true!*

We illustrate that with the following example. The derivation will show a situation where taking into account all information is not enough, but that illustration is not in itself an

intuitive *explanation* of that fact. A derivation like that of the example is sometimes called an **oracular derivation**

Example 4.2.5 (Oracular derivation). Consider the following grammar, another variation on expressions and terms, it's basically a slightly rewritten version of the grammar from equation (**??**).

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{exp} + \mathit{term} \mid \mathit{exp} - \mathit{term} \mid \mathit{term} \\ \mathit{term} & \to & \mathit{term} * \mathit{factor} \mid \mathit{factor} \\ \mathit{factor} & \to & (\ \mathit{exp}\ ) \mid \textbf{number} \end{array} \tag{4.4}$$

Equation (4.5) shows a derivation of the numerical expression `1+2*3`. It's again a left-most derviation.

$$\begin{array}{lll} \underline{\mathit{exp}} & \Rightarrow_1 & \downarrow 1+2*3 \\ \underline{\mathit{exp}} + \mathit{term} & \Rightarrow_3 & \downarrow 1+2*3 \\ \underline{\mathit{term}} + \mathit{term} & \Rightarrow_5 & \downarrow 1+2*3 \\ \underline{\mathit{factor}} + \mathit{term} & \Rightarrow_7 & \downarrow 1+2*3 \\ \textbf{number} + \mathit{term} & & \downarrow 1+2*3 \\ \textbf{number} + \mathit{term} & & 1\downarrow +2*3 \\ \textbf{number} + \underline{\mathit{term}} & \Rightarrow_4 & 1+\downarrow 2*3 \quad \textbf{!} \\ \textbf{number} + \underline{\mathit{term}} * \mathit{factor} & \Rightarrow_5 & 1+\downarrow 2*3 \quad \textbf{!} \\ \textbf{number} + \underline{\mathit{factor}} * \mathit{factor} & \Rightarrow_7 & 1+\downarrow 2*3 \\ \textbf{number} + \textbf{number} * \mathit{factor} & & 1+\downarrow 2*3 \\ \textbf{number} + \textbf{number} * \mathit{factor} & & 1+2\downarrow *3 \\ \textbf{number} + \textbf{number} * \underline{\mathit{factor}} & \Rightarrow_7 & 1+2*\downarrow 3 \\ \textbf{number} + \textbf{number} * \textbf{number} & & 1+2*\downarrow 3 \\ \textbf{number} + \textbf{number} * \textbf{number} & & 1+2*3\downarrow \end{array} \tag{4.5}$$

Again, the *redex*, the place where the step occurs, is underlined. In addition, we show on the right-hand column the input and the current progress on that input. The subscripts on the derivation arrows indicate which rule is chosen in that particular derivation step.

The point of the example is the following: Consider lines 7 and 8, and the steps the parser does. In line 7, it is about to expand *term* which is the left-most terminal. Looking into the "future" the unparsed part is `2 * 3`. In that situation, the parser chooses production 4 (indicated by $\Rightarrow_4$). In the next line, the left-most non-terminal is *term again* and also the non-processed input has not changed. However, in that situation, the "oracular" parser chooses $\Rightarrow_5$.

What does that mean? It means, that the look-ahead did not help the parser! It used all look-ahead there is, namely until the very end of the word. And it *still* cannot make the right decision with all the knowledge available at that given point. Note also: choosing wrongly (like $\Rightarrow_5$ instead of $\Rightarrow_4$ or the other way around) would lead to a failed parse (which would require *backtracking*). That means, it's unparseable without backtracking (and no amount of look-ahead will help), at least we need backtracking, if we do left-derivations and **top-down.**

Right-derivations are not really an option, as typically we want to eat the input left-to-right. Secondly, right-most derivations will suffer from the same problem (perhaps not for the very grammar but in general, so nothing would even be gained.)

On the other hand: bottom-up parsing later works on different principles, so the particular problem illustrated by this example will not bother that style of parsing (but there are other challenges then).

So, what *is* the problem then here? The reason why the parser could not make a uniform decision (for example comparing line 7 and 8) comes from the fact that these two particular lines are connected by $\Rightarrow_4$, which corresponds to the production

$$term \to term * factor$$

there the derivation step replaces the left-most *term* by *term* again without moving ahead with the input. This form of rule is said to be *left-recursive* (with recursion on *term*). This is something that top-down or recursive descent parsers *cannot deal with* (or at least not without doing backtracking, which is *not* an option).

We will learn how to transform grammars automatically to *remove* left-recursion. It's an easy construction. Note, however, that the construction not necessarily results in a grammar that afterwards *is* top-down parsable. It simple removes a "feature" of the grammar which definitely cannot be treated by top-down parsing. □

So, one lesson from the previous example was

> left-recursion destroys top-down parseability

(when based on left-most derivations/left-to-right parsing as it is always done for top-down parsing).

Side remark 4.2.6 (Nit-picking)**.** As side remark, for being super-precise: If a grammar contains left-recursion on a non-terminal which is irrelevant in that no word will ever lead to a parse involving that particular non-terminal or in that the non-terminal will never show up in any derivable sentential form, in that case, obviously, left-recursion does not hurt.

Of course, the grammar in that case would be silly. We in general do not consider grammars which contain such irrelevant symbols (or have other such obviously meaningless defects). But unless we exclude such silly grammars, the take-home lesson from the previous example is not 100% true. □

That left-recursion is problematic for top-down parsing is one lesson, the other one perhaps is that making decisions is not easy: not even a machine with perfect recall and with perfect foresight, i.e., remembering all the past, and working with unbounded look-ahead, can make deterministic decisions in some cases.

But what can be done? We start simply by putting our foot down and rule out grammars where the decision cannot be done (which would effectively require back-tracking and we can't have that).

Let's assume we are working with a left-most derivation (which is what top-down parsing resp. LL-parsing does) and assume the left-most non-terminal A in a derivation is covered by two rules in the grammar:

$$A \to \beta \mid \gamma$$

Now consider again the left-most derivation of a word w from equation (4.3). In the step we focus on replacing the left-most A, the "past" is fixed the "future" is not, and the given target word w is of the form $w = w_1 w_2$, with w_1 being in the past. For the "future", there are two possible continuations:

$$A\alpha_2 \Rightarrow_l \beta\alpha_2 \Rightarrow_l^* w_2 \quad \text{or else} \quad A\alpha_2 \Rightarrow_l \gamma\alpha_2 \Rightarrow_l^* w_2 \ ? \tag{4.6}$$

As we assume the grammar to be unambiguius, that not only means the first production was a correct decision that the given point, it was the *only* correct decision (assuming of course $\beta \neq \gamma$) and indeed the derivation from equation (4.3) is the **only** left-most derivation for w. But that's just a different way of saying that the grammar is unambiguous.

> **minimal requirement** In such a situation, "future target" w_2 must *determine* which of the rules to take!

That allows deterministic top-down parsing, but it is still impractical, as the "target" from equation (4.6) w_2 corresponds to a look-ahead of *unbounded length*! In practicem a **look-ahead of length** k must be enough resolve the "which-right-hand-side" non-determinism inspecting only fixed-length prefix of w_2 (for *all* situations as above).

> An *LL(k)-grammar* is a context-free grammar which can be parsed left-to-right and left-most strategy with a look-ahead of k.

Of course, one can always write a parser that "just makes some decision" based on looking ahead k symbols. The question is: will that allow to capture *all* words from the grammar and *only* those.

Having defined our expectations to a decent top-down parser, namely that it can make decisions with a fixed look-ahead, we should realize that this only helps if we can determined whether or not a grammar meets that requirement. So far it's more a wish rather than a criterion that we can apply to a grammar to see whether it works or not.

In the following Section 4.3, we introduce the concepts that allows us to actually **determine** whether a given grammar is LL(k) or not (it's thus decidable), The concepts are called **first-** and **follow-sets**. They will also be helpful in the another class of parsers, the bottom-up parsers to determine analogous question, is a grammar bottom-up parseable with a fixed look-ahead.

4.3 First and follow sets

The considerations leading to a useful criterion for top-down parsing without backtracking will involve the definition of the so-called "first-sets". In connection with that definition, there will be also the (related) definition of *follow-sets*.

We had a general look of what a look-ahead is, and how it helps in top-down parsing. We also saw that *left-recursion* is bad for top-down parsing (in particular, there can't be any look-ahead to help the parser). The definition discussed so far, being based on arbitrary derivations, were impractical. What is needed is a criterion, not on derivations, but on *grammars* that can be used to figure out, whether the grammar is parseable in a top-down manner with a look-ahead of, say k. Actually we will concentrate on a look-ahead of $k = 1$, which is practically often a decent thing to do.

The definitions, as mentioned, will help to figure out if a grammar is top-down parseable. Such a grammar will then be called an LL(1) grammar. One could straightforwardly generalize the definition to LL(k) (which would include generalizations of the first and follow sets), but that's not part of the pensum. Note also: the first and follow set definition will *also* be used when discussing *bottom-up* parsing later.

The *first-* and *follow-sets* is a general concept for grammars and in particular for parsing. Later we will spell out the concepts more formally (and give algorithms to calculate those sets). But let's start by stating informally what they capture.

The **first-set** of a non-terminal A is the set of terminal symbols can appear at the **start** of strings *derived from* A.
The **follow-set** of a non-terminal A is the set of terminals that can appear directly after A when a is mentioned in some *sentential form*.

Remember that a sentential form is a word *derived from* grammar's starting symbol.

There will be some fine-points to be added to this informal description, but it covers the basic idea. We may also mention here, that we will use first-sets not just for non-terminals, as in the basic definition here. We will cover an *analogous* definition for words. The actual calculation of those sets is mostly straightforward, there will only one slight complication to watch out for, which has to do with empty words ϵ *nullable* symbols (non-terminals). Note, those sets depend on the *grammar,* not the language

4.3.1 First sets

The first-sets are conceptually simpler than the follow-sets, so let's start with those, and let's start with a more formal definition.

Definition 4.3.1 (First set)**.** Given a grammar G and a non-terminal A. The *first-set* of A, written $First_G(A)$ is defined as

$$First_G(A) = \{a \mid A \Rightarrow_G^* a\alpha, \quad a \in \Sigma_T\} + \{\epsilon \mid A \Rightarrow_G^* \epsilon\} \; . \tag{4.7}$$

In the following, if the grammar G clear from the context, we write $\Rightarrow^*$ for $\Rightarrow_G^*$ and *First* for $First_G$.

Comparing the definition with the informal statement from before, we see also one finepoint not mentioned before. That's that the first set does not only contain terminal symbols, it

also carries the information whether or not the non-terminal in question can be derived into the empty word ϵ. There is a word for such non-terminals, they are called *nullable.*

Definition 4.3.2 (Nullable)**.** Given a grammar G. A non-terminal $A \in \Sigma_N$ is *nullable*, if $A \Rightarrow^* \epsilon$.

The definition here of being nullable refers to a non-terminal symbol. When concentrating on context-free grammars, as we do for parsing, that's basically the only interesting case. In principle, one can define the notion of being nullable analogously for arbitrary words from the whole alphabet $\Sigma = \Sigma_T + \Sigma_N$. The form of productions in CFGs makes it obvious, that the only words which actually may be nullable are words containing only non-terminals. Once a terminal is derived, it can never be "erased". It's equally easy to see, that a word $\alpha \in \Sigma_N^*$ is nullable iff all its non-terminal symbols are nullable. The same remarks apply to context-sensitive (but not general) grammars.

For level-0 grammars in the Chomsky-hierarchy, also words containing terminal symbols may be nullable, and nullability of a word, like most other properties in that setting, becomes undecidable.

Let's look informally at some examples. In many languages

$$\mathit{First}(\mathit{if\text{-}stmt}) = \{\text{"}\textbf{if}\text{"}\} \quad \text{and} \quad \mathit{First}(\mathit{assign\text{-}stmt}) = \{\textbf{identifier}, \text{"(")}\} \ ;$$

The **if** etc. are meant here as tokens, so assuming that a language with conditional may use the concrete lexeme `if` and that lexer produces a token with the plausbly called **if**. etc. But the distinctions between lexemes and tokens should be clear by now. A typical follow set (see later) for statements could be

$$\mathit{Follow}(\mathit{stmt}) = \{\text{";"}, \text{"}\textbf{end}\text{"}, \text{"}\textbf{else}\text{"}, \text{"}\textbf{until}\text{"}\}$$

We next generalize and reformulate the definition of the first slighty. We generalize it slightly so tha it applies not only to non-terminals, but to terminal symbols. But that's not the interesting part of new formulation..

The previous definition 4.3.1 defines the first-sets referring to words derivable from the given symbol. Being based on derivations by aritrary many steps, that does not lend itself to determine the first-set algorithmically. The following formulation 4.3.3 does not refer to derivations using a grammar, but only to the rules of the given grammar. Furthermore, it's a **recursive** definition, where the first-set of one symbol is defined in terms of the the first-set of other symbols (connected by the productions of the grammar). That recursive definition will later be the basis for an algorithms to calculate the first-set. We will do a similar recursive reformulation for the follow-sets as well.

Definition 4.3.3 (First set of a symbol)**.** Given a grammar G and grammar symbol X. The *first-set* of X, written $First(X)$, is defined as follows:

1. If $X \in \Sigma_T + \{\epsilon\}$, then $First(X)$ contains X.
2. If $X \in \Sigma_N$: For each production

$$X \to X_1 X_2 \dots X_n$$

   a) $First(X)$ contains $First(X_1) \setminus \{\epsilon\}$
   b) If, for some $i < n$, *all* $First(X_1), \dots, First(X_i)$ contain ϵ, then $First(X)$ contains $First(X_{i+1}) \setminus \{\epsilon\}$.
   c) If all $First(X_1), \dots, First(X_n)$ contain ϵ, then $First(X)$ contains $\{\epsilon\}$.

For completeness sake, let's generalize the definition a bit further, to apply also for words.

Definition 4.3.4 (First set of a word)**.** Given a grammar G and word α. The *first-set* of

$$\alpha = X_1 \dots X_n \ ,$$

written $First(\alpha)$ is satisfies the following conditions

1. $First(\alpha)$ contains $First(X_1) \setminus \{\epsilon\}$
2. for each $i = 2, \dots n$, if $First(X_k)$ contains ϵ for *all* $k = 1, \dots, i-1$, then $First(\alpha)$ contains $First(X_i) \setminus \{\epsilon\}$
3. If all $First(X_1), \dots, First(X_n)$ contain ϵ, then $First(X)$ contains $\{\epsilon\}$.

As said, the definition here is of course very close to the definition of the inductive case of the previous definition, i.e., the first set of a non-terminal.

Note that the word α may be empty, i.e., $n = 0$, In that case, the definition gives $First(\epsilon) = \{\epsilon\}$ (due to the 3rd condition in the above definition). In the definitions, the empty word ϵ plays a specific, mostly technical role. The original, non-algorithmic version of Definition 4.3.1, makes it already clear, that the first set *not precisely* corresponds to the set of terminal symbols that can appear at the beginning of a derivable word. The correct intuition is that it corresponds to that set of terminal symbols *together* with ϵ as a special case, namely when the initial symbol is nullable.

Remark 4.3.5 (The role of ϵ)**.** That may raise two questions. 1) Why does the definition makes that as special case, as opposed to just using the more "straightforward" definition without taking care of the nullable situation? 2) What role does ϵ play here?

The second question has no "real" answer, it's a choice which is being made which could be made differently. What the definition from equation (4.3.1) in fact says is: "give the set of terminal symbols in the derivable word **and** indicate whether or not the start symbol is *nullable*." The information might as well be interpreted as a *pair* consisting of a set of terminals *and* a boolean (indicating nullability). The fact that the definition of *First* as presented here uses ϵ to indicate that additional information is a particular choice of representation (probably due to historical reasons: "they always did it like that ..."). For

instance, the influential "Dragon book" [1, Section 4.4.2] uses the ϵ-based definition. The texbooks [2] (and its variants) don't use ϵ as indication for nullability.

In order that this definition works, it is important, obviously, that ϵ is *not* a terminal symbol, i.e., $\epsilon \notin \Sigma_T$ (which is generally assumed).

Having clarified 2), namely that using ϵ is a matter of conventional choice, remains question 1), why bother to include nullability-information in the definition of the first-set *at all*, why bother with the "extra information" of nullability? For that, there is a real technical reason: For the *recursive* definitions to work, we need the information whether or not a symbol or word is *nullable*, therefore it's given back as information. □

As a further point concerning the first sets. we have given 2 "definitions", Definition 4.3.1 and Definition 4.3.3 (and we generalized it to work for words). Of course they are intended to mean the same. The second version is a more recursive version, i.e., closer to a recursive algorithm. If one takes the first one as the "real" definition of that set, in principle we would be obliged to prove that both versions actually describe the same concept (resp. that the recursive definition *implements* the original definition). But we won't establish that connection.

Remark 4.3.6 (Elaborations in the recursive "definition" of *First*?)**.** The following discussion may be ignored, if wished. Even if details and theory behind it is beyond the scope of this lecture, it is worth to consider Definition 4.3.3 more closely.

The first impression may have been, it's a recursive definition of the function *First*, but we by observing that it's actually not a definition, at least it's *not* a definition of what *the* first-set of a symbol is.

As discussed later, everything gets rather simpler if we would not have to deal with nullable words and ϵ-productions. So for the current discussion, let's assume that there are no such productions and get rid of the special cases, cluttering up Definition 4.3.3. Removing the clutter gives the following simplified definition:

Definition 4.3.7 (First set of a symbol (no ϵ-productions))**.** Given a grammar G and grammar symbol X. The *First-set* of $X \neq \epsilon$, written $\mathit{First}(X)$ is defined as follows:

1. If $X \in \Sigma_T$, then $\mathit{First}(X) \supseteq \{X\}$.
2. If $X \in \Sigma_N$: For each production

$$X \to X_1 X_2 \dots X_n \ ,$$

$\mathit{First}(X) \supseteq \mathit{First}(X_1)$.

Compared to the previous condition, I did the following minor adaptation (apart from cleaning up the ϵ's): I replaced the English word "contains" with the superset relation symbol $\supseteq$. Now, with Definition 4.3.7 as a simplified version of the original definition being made slightly more explicit: in which way is that a definition at all?

For being a definition for $\mathit{First}(X)$, it seems lax. Already in (1), it "defines" that $\mathit{First}(X)$ should "at least contain X". A similar remark applies to case (2) for non-terminals. Those

two requirements are as such well-defined, but *they don't define* $First(X)$ *in a unique manner!* Definition 4.3.7 defines what the set $First(X)$ should *at least* contain!

One should thus not consider Definition 4.3.7 a recursive definition of $First(X)$ but rather

> a definition of recursive conditions on $First(X)$, which, when satisfied, ensures that $First(X)$ contains *at least* all non-terminals we are after.

What we are *really* after is the *smallest* $First(X)$ which satisfies those conditions of the definitions.

Now one may think that the problem is that the definition is just sloppy. Why did it uses the word "contains" resp. the $\supseteq$-relation, instead of requiring equality, i.e., $=$? While plausible at first sight, unfortunately, whether we use $\supseteq$ or set equality $=$ in Definition 4.3.7 does not change anything.

The core of the matter is not $=$ vs. $\supseteq$. The core of the matter is that is **circular!**

Considering that definition of $First(X)$ as a plain functional and recursive definition of a *procedure* missed the fact that grammar can, of course, contain "loops". Actually, it's a characterizing feature of reasonable context-free grammars (or even regular grammars) that they contain "loops" – that's the way they can describe infinite languages. So considering Definition 4.3.3 with $=$ instead of $\supseteq$ as the recursive definition of a function leads immediately to an **infinite regress**, the recursive function won't terminate. So that's not helpful.

Technically, such a definition can be called a recursive **constraint** (or a constraint system, if one considers the whole definition to consist of more than one constraint, namely for different terminals and for different productions).

The algorithms we will encounter later, so-called worklist algorithms, is a well-known way so solve the particular form of constraints that underly Definition 4.3.7. We leave it at that, and return to the main text. □

Algorithm for determining the first-sets

Next we present an algorithm for the first-sets, given a grammar. We don't present the algorithm right away, but focus on the simpler setting, for a grammar without ϵ-*productions*, i.e., without productions of the form $A \to \epsilon$. The ultimate algorithm won't be principally more complex, but they underlying idea of the algorithm(s) is more visible in the simplified settting, so we discuss that first.

So the goal is, given a grammar, calculate for each non-terminal the first-set. In the simplified setting, that will be a set of terminal symbol, ϵ will not be part of the result.

As a data structure, we can arrange that result as an *array,* which stores for each non-terminal that set. Very schematically, the algorithm looks is sketched in Listing 4.1. Basically, after some initialization, it's a big loop, which updates the data structure containing current estimations of the first sets. The update in the loop body is specific in that it only *increases* the current estimation of a first set. That continues until there are

no more changes to the entries in the array, i.e., no more increases. At that point, the iteration has reached the first-sets and terminates.

```
initialize(First);
while there are changes to any First[A] do
  for each production A → X_1 ... X_n do
      First[A] := First[A] ∪ First[X_1]
  end;
end
```

Listing 4.1: Schematic form of calculating the first-set (no ϵ)

We have left out the initialization of the first-set approximations. The loop of the algorithm increases the approximations until stabilization, and at the beginning we start with the smallest estimations. Or at least the smallest reasonable, taking into account information that is *immediately* available. For terminal symbols and for ϵ, the first-set is immediately clear (and it won't be increased by the loop). For non-termials, the smallest initial guess is the empty set. That's shown in 4.2.

```
for all X ∈ Σ_T ∪ {ϵ} do
    First[X] := X
end;

for all non-terminals A do
  First[A] := {}
end
```

Listing 4.2: Initialization

Currently we leave out ϵ-productions, so the first line could be simplified, leaving out ϵ. But by considering it already, tge initialization won't change also when later looking at the slightly more complex setting.

Let's illustrate how the algorithm works on a grammar without ϵ.

Example 4.3.8 (Example expression grammar). We have seen that grammar before, in the context of discussing ambiguity and the concept of precedence cascades (see Example **??** and equation (**??**)). We repeat the grammar here, but in expanded form, with one production per line.

$$\begin{aligned}
\mathit{exp} &\to \mathit{exp}\ \mathit{addop}\ \mathit{term} \\
\mathit{exp} &\to \mathit{term} \\
\mathit{addop} &\to + \\
\mathit{addop} &\to - \\
\mathit{term} &\to \mathit{term}\ \mathit{mulop}\ \mathit{factor} \\
\mathit{term} &\to \mathit{factor} \\
\mathit{mulop} &\to * \\
\mathit{factor} &\to (\ \mathit{exp}\ ) \\
\mathit{factor} &\to \mathbf{number}
\end{aligned} \tag{4.8}$$

Table 4.2 indicates how the algo runs, or at least one possible run. It should also be noted that the table from above is a schematic illustration of a particular *execution strategy* of the pseudo-code. The pseudo-code itself leaves out details of the evaluation, notably *the order* in which non-deterministic choices are done by the code. The main body of the pseudo-code is given by two nested loops. Even if details (of data structures) are not

Grammar rule	Pass 1	Pass 2	Pass 3
exp → *exp* *addop term*			
exp → *term*			First(*exp*) = { (, **number** }
addop → **+**	First(*addop*) = {**+**}		
addop → **-**	First(*addop*) = {**+**, **-**}		
term → *term* *mulop factor*			
term → *factor*		First(*term*) = { (, **number** }	
mulop → *****	First(*mulop*) = {***** }		
factor → **(** *exp* **)**	First(*factor*) = {**(** }		
factor → **number**	First(*factor*) = { (, **number** }		

Table 4.2: Run of the algorithm

given, one possible way of interpreting the code is as follows: the outer while-loop figures out which of the entries in the `First`-array have "recently" been changed, remembers that in a "collection" of non-terminals A's, and that collection is then worked off (i.e. iterated over) on the inner loop. The table is meant to indicate that the rows are done from top to bottom. That means, the inner loop treats the productions in each pass in the order as shown in equation (4.8). And the passes obviously proceed from left to right.

Doing it like that leads to the "passes" shown in the table. In other words, the two dimensions of the table represent the fact that there are two nested loops.

Tables 4.5 show the result more compactial, in particular, Table 4.5b contains the result after termination, i.e., the first-sets for each non-terminal.

	1	2	3
exp			$\{(, \mathbf{n}\}$
addop	$\{+, -\}$		
term		$\{(, \mathbf{n}\}$	
mulop	$\{*\}$		
factor	$\{(, \mathbf{n}\}$		

(a) Results per pass

	First[_]
exp	$\{(, \mathbf{n}\}$
addop	$\{+, -\}$
term	$\{(, \mathbf{n}\}$
mulop	$\{*\}$
factor	$\{(, \mathbf{n}\}$

(b) final results (at the end of pass 3, resp. 4)

Figure 4.5: Collapsing the rows & final result

The tables show 3 passes, and the result correspond to the state at the end of the 3rd pass. Technically, the algorithm cannot "know" that at the end of the 3rd pass, the result has been achieved. It has to run a 4th time, at which point it it's clear that there is no change from the 3rd round to the 4th round, which also means, that any further rounds would not give more information. The information has stabilized, at round 3 and that becomes clear at round 4, at which point, the algo terminates. □

Remark 4.3.9 (Traversal strategies and worklist algorithms)**.** Table 4.2 represents only one particular traversal througb the grammar, namely treating each production one after the other in the order as given in the grammar. The order does not influence the ultimate result, but can inflience the time it takes till stabilization. To see that it can be instructive

to apply to apply the opposite traversal strategy, going through the productions in reverse order.

Actually, it's not even needed to arrange the treatment of the productions in the way shown in Listing 4.1, by going through all the rules in one pass, and repeating the passing. One could randomly pick rules to treat until statilization.

We don't dig deeper here, who is interested can read some more at

`https://martinsteffen.github.io/compilerconstruction/worklistalgos/`

discussing the issue and so-called *worklist algorithms,* using first-set calculations as illustration. That discussion is not part of the pensum.

But such algorithms are in wider use than just for the first- and later the follow-sets. Also inside a compiler, they can play variously a role. For example, in a quite later section, we shortly look at (one example of) **data-flow** analysis. Also that is a typical example that can be addressed by a worklist algorithm. □

With ϵ-productions

Before proceeding to the follow-sets, let's shortly look at how to generalize the code from Listing 4.1 to deal with nullable symbols. See also the general recursive characterization from Definition 4.3.3. For the *initialization,* we can reuse the code from Listing 4.2, as that one covers the ϵ already; it corresponds to the case 1 for Definition 4.3.3.

The other case, in particular part 2a and 2c of the definition call for another loop, going through the right-hand side of the production under current treatment, and checking for the ϵ-status of the involved symbols. The resulting code is shown in Listing 4.3.

```
1  initialize(First);
2  while there are changes to any First[A] do
3    for each production A → X_1 ... X_n do
4      k := 1;
5      continue := true
6      while continue = true and k ≤ n do
7        First[A] := First[A] ∪ First[X_k] \ {ϵ}
8        if ϵ ∉ First[X_k] then continue := false
9        k := k + 1
10     end;
11     if   continue = true
12     then First[A] := First[A] ∪ {ϵ}
13   end;
14 end
```

Listing 4.3: First sets

4.3.2 Follow sets

Let's do the same for the follow-sets, what we did for the first-sets in . Let's start with a more formal definition of the concept.

Definition 4.3.10 (Follow set)**.** Given a grammar G with start symbol S, and a non-terminal A.
The *follow-set* of A, written $Follow_G(A)$, is

$$Follow_G(A) = \{a \mid S\,\$ \Rightarrow_G^* \alpha_1 A a \alpha_2, \quad a \in \Sigma_T + \{\,\$\,\}\} \; . \tag{4.9}$$

Compared to the informal description from page 14, apart from writing it up more formally, one detail is added. The definition does not talk about sentential forms derivable from the start symbol S, but starting from $S\,\$$. The symbol $\$$ used as a special end-marker of derivable words. It's treated as terminal.

There is a convention in grammars we often rely on, namely, the *start symbol* S is not mentioned on any right-hand side of a production. This is not a real restriction, one can always add an extra start symbol and a production $S' \to S$, if the condition is violated for S. That convention will slightly simplify the treatment here (and later), not in substance, but streamlining some corner cases.

The symbol $\$$ can be interpreted as "end-of-file" (`EOF`) token. Under the assumption, that the start symbol S does not occur on the right-hand side of any production, case, the follow set of S contains $\$$ as *only* element. Note that the follow set of other non-terminals may well contain $\$$.

Side remark 4.3.11 (Start symbol)**.** As said, it's common to assume that S does not appear on the right-hand side of any production. With S occuring only on the left-hand side, the grammar has a slightly nicer shape insofar as it makes its algorithmic treatment slightly nicer. It's basically the same reason why one sometimes assumes that, for instance, control-flow graphs have one "isolated" entry node (and/or an isolated exit node), where being isolated means, that no edge in the graph goes (back) into into the entry node; for exit nodes, the condition means, no edge goes out. In other words, while the graph can of course contain loops or cycles, the entry node is not part of any such loop. That is done likewise to (slightly) simplify the treatment of such graphs. Slightly more generally and also connected to control-flow graphs: similar conditions about the shape of loops (not just for the entry and exit nodes) have been worked out, which play a role in loop optimization and intermediate representations of a compiler, such as static single assignment forms.

We will encounter control-flow graphs in a later chapter. □

As we did for the first-sets in Definition 4.3.3, we continue by giving a **recursive characterization** of the concept, which we afterwards will turn into an algorithm.

Definition 4.3.12 (Follow set of a non-terminal)**.** Given a grammar G and non-terminal A. The *Follow-set* of A, written $Follow(A)$ is defined as follows:

1. If A is the start symbol, then $Follow(A)$ contains $\$$.
2. If there is a production $B \to \alpha A \beta$, then $Follow(A)$ contains $First(\beta) \setminus \{\boldsymbol{\epsilon}\}$.
3. If there is a production $B \to \alpha A \beta$ such that $\boldsymbol{\epsilon} \in First(\beta)$, then $Follow(A)$ contains $Follow(B)$.

The corresponding code is shown in Listing 4.4. The structure of the calculation is the same as for the first-sets. There is a initialization. The core of the algorithm is some loop adding information about the (current approximation of) the follow sets, enlarging it as long as the program loops through the iteration. The loop terminates when no new information is added and the follow-sets stabilizes. As for the first-sets, the initialization starts with minimal estimations, setting all initial sets to the empty set, with the only exception of the start symbol (assumed "isolated"), for which a follow set of {**$**} is immediately clear.

```
1  Follow[S] := {$}
2  for all non-terminals A ≠ S do
3    Follow[A] := {}
4  end
5  while there are changes to any Follow-set do
6    for each production A → X_1 ... X_n do
7      for each X_i which is a non-terminal do
8        Follow[X_i] := Follow[X_i] ∪ (First(X_{i+1} ... X_n) \ {ϵ})
9        if ϵ ∈ First(X_{i+1} X_{i+2} ... X_n)
10       then Follow[X_i] := Follow[X_i] ∪ Follow[A]
11     end
12   end
13 end
```

Listing 4.4: Follow-set algorithm

It should be noted the algorithm for the follow-sets relies on the information of the first-sets (as is also the case for the recursive characterization from Definition 4.3.12, though not for the original Definition 4.3.10), i.e., to calculate the follow sets requires to calculate the first-sets first.

In the algorithm, there is one **corner case** one has to keep in mind. It has to do with ϵ and the first-sets. At the end of the loop, there is a check whether ϵ is contained in $First(X_{i+1}X_{i+2}\dots X_n)$. In other words, it's checked whether the word $X_{i+1}X_{i+2}\dots X_n$ is **nullable.** That depends, of course, on the first-sets of the X_j's mentioned in the condition. There is, however, the corner case of $i = n$. In this case the sequence is $X_{i+1}X_{i+2}\dots X_n$ empty, i.e., it represents the empty word. That's certainly nullable, in other words $First()$ or $First(\epsilon)$ is $\{\epsilon\}$.

We mentioned that the calculation of the first-sets of a symbol when the grammar does not contain ϵ, as one can ignore the corresponding definition. *Here*, however, even if the grammar does not mention ϵ, we cannot ignore the corresponding corner-case in the algo from Listing 4.4.

Example 4.3.13 (Follow sets (of an expression grammmar)). Let's revisit the expression grammar from equation (4.8). Example 4.3.8 calculated the first-sets for that grammar. Based in that information, let's calculate the follow-sets, as well, using the saturation algorithm.

A run is illustrated in Table 4.3. The table omits productions which have terminals only on their right-hand side. The algo does not do anything in those cases anyway. The grammar does not contain nullable symbols, which means, the algo is a bit more simple. We remember, that the first-procedure used ϵ for nullable symbol. However, the first

procedure here is used not on non-terminals, but on *words*. And that word $X_{i+1} \dots X_n$ may itself be ϵ, and that is where the last clause of the algo kicks in.

Grammar rule	Pass 1	Pass 2
exp → *exp addop term*	Follow(*exp*) = {$, +, -} Follow(*addop*) = {(, **number**} Follow(*term*) = {$, +, -}	Follow(*term*) = {$, +, -, *,)}
exp → *term*		
term → *term mulop factor*	Follow(*term*) = {$, +, -, *} Follow(*mulop*) = {(, **number**} Follow(*factor*) = {$, +, -, *}	Follow(*factor*) = {$, +, -, *,)}
term → *factor*		
factor → (*exp*)	Follow(*exp*) = {$, +, -,)}	

Table 4.3: Run of the follow-set algorithm

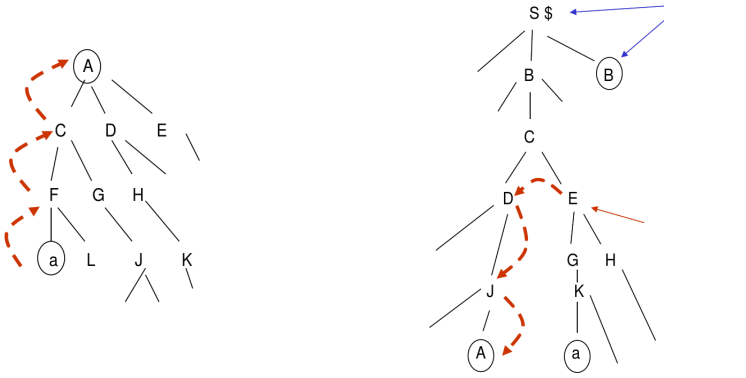
The algorithm this time contains three loops, the outermost while-loop corresponds as before to the passes, the loop nested inside that correspond to the the rows of the table. The innermost loop through the right-hand side of each rule corresponds to the lines inside the slots of the table.

Compared to the corresponding Table 4.2 for the first-sets of the same grammar, here there are only five rows, because here we are not interested in information about terminal symbols.

One should look especially at the last entry in each slot, for instance that for *term* in the first row of pass 1. The non-terminal *term* is the last symbol on the right-hand side of the production under treatment and so that corresponds to the corner-case involving ϵ, mentioned earlier. □

Let's illustrate the calculation of the first and the follow sets schematically, or at least some aspects of the calculation. See Figure 4.6. The grammar is not given explicity, but the graphs or trees in the figures are intended to represent the *grammar*, not a parse-tree or similar. For instance, in the tree of Figure 4.6a, the root A has three children C, D, and E, there must be a production $A \to C\ D\ E$, the subtree in the left-lower corner of the tree in represents a production $F \to \mathbf{a}L$, etc.

The red arrows in the figures illustrate the *information flow*, as calculated by the first-set resp. follow set algorithm. For instance, the first-set of A in the grammar of Figure 4.6a contains **a**. The first-set definition would "immediately" detect that F has **a** in its first-set, i.e., all words derivable starting from F start with an **a** (and actually with no other terminal, as F is mentioned only once in that sketch of a tree). At any rate, only *after* determining that **a** is in the first-set of F, then it can enter the first-set of C, etc. and in this way percolating upwards the tree.



(a) First-set

(b) Follow set

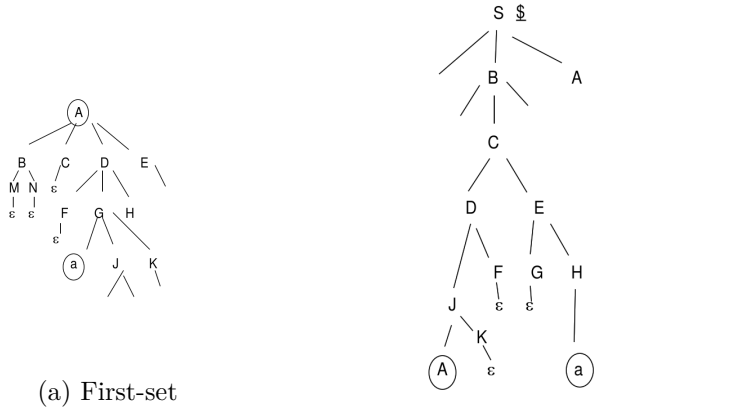
Figure 4.6: Illustration of information flow

The pictures give an impression on the flow of "information", the first-set information flows upward and the follow-set information downward, with non-terminal on the left-hand side of a production the parent-node higher-up, and the symbols on the right-hand side as children lower down. However, the picture is misleading or unrealistic in one aspect. As mentioned, the graphs are meant to represent the grammars, not parse trees. But context-free grammars always contain recursion, except for degenerated cases (for finite languages). That means, the situation is that of a graph with cycles, not that of a tree. That also means, one cannot reasonably say, that first-set information percolates *up* resp follow-set information percolates *down*. The fact that grammars contain cycles is crucial for how to algorithmically approach the problem. Without going into details, the presence of cycles is the reason that the first- and the sketched follow-set algorithms add information **until stabilization** of the collected information, and that typically requires dealing with rules multiple times. If one had an (unrealistic) grammar without cycles (i.e., a tree-problem), one could percolate the first-set information upwards the tree until one reaches the root, and then stop. There would be no need to treat productions multiple times, and one would not need stabilization as termination criterion.

Now let's look a the more complex situation with **nullable** symbols. In the tree from Figure 4.7a, B, M, N, C, and F are *nullable*. Consequently, the first-set of A contains **a**. Similarly for the grammar from Figure 4.7b. The follow-set of A contains **a** where the ϵ's and the nullable symbols in the tree are "hopped over".

4.4 Massaging grammars

We have learned the first- and follow-set as "tools" to diagnose the shape of a grammar. In particular the follow-set is connected with the notion of **look-ahead**, on which we have touched upon earlier when sketching how generally a parser works. Looking ahead can help to make decisions which step to take and to build up the parse tree, while eating through the token stream. The general picture applies to both bottom-up and top-down parsing, which implies, the first- and follow-sets play a role as diagnosis instrument for both kinds of parsings.



(a) First-set

(b) Follow set

Figure 4.7: Illustration of information flow with nullability

By diagnosis, I mean in particular:

> the first- and in particular the follow-sets can be used to check whether or not it's possible to deterministically parse a given grammar with a look-ahead of one symbol.

The whole picture could more or less straightforwardly be generalized for a longer look-ahead: top-down parsing or bottom-up parsing with a look-ahead of k would require appropriate generalizations of the first-sets and follow-sets to speak not about $k = 1$ symbol but longer words. In practice, one is mostly content with $k = 1$, which is also why we don't bother about generalizing the setting. And actually, if one understands the concept of first- and follow set with one symbold resp. the concept of one look-ahead, nothing conceptually changes when going to $k > 1$.

As said, the first- and follow sets are relevant for both top-down and bottom-up parsers. Here, however, we continue covering **top-down parsing,** which has slightly different challenges than bottom-up. Before we come actually to top-down parsing, we discuss, what are problematic patterns in grammars, i.e., patterns that top-down parser have troubles with, and we use the notions follow sets to shed light on that.

> The two troublesome pattern for top-down parsing we will discuss that way are **left-recursive** grammars and grammars with **common left factors**.

We will also dicsuss, how to massage such troublesome grammars in a way to get rid of those patterns.

Definition 4.4.1 (Left-recursion and common left-factors.)**.** A *left-recursive* production is of the form

$$A \to A\alpha \tag{4.10}$$

Two productions have a *common left factor* if they are of the form

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \qquad \text{where } \alpha \neq \boldsymbol{\epsilon} \tag{4.11}$$

For the shape of equation (4.10), one should more precisely speak of *immediate* left-recursion. Indirect left-recursion, involving more than one rule, is equally bad for top-down parsing.

At the current point in the presentation, the importance of those conditions might not yet be clear (but remember the discussion around "oracular" derivations). In general, it's that certain kind of parsing techniques require absence of left-recursion and of common left-factors. Note also that a left-linear production is a special case of a production with immediate left recursion. In particular, recursive descent parsers would not work with left-recursion. For that kind of parsers, left-recursion needs to be avoided.

Why common left-factors are undesirable should at least intuitively be clear: we see this in the example below with the two forms of conditionals. It's intuitively clear, that a parser, when encountering an **if** (and the following boolean condition and perhaps the **then** clause) cannot decide immediately which rule applies. It should also be intuitively clear that that's what a parser does: inputting a stream of tokens and trying to figure out which sequence of rules are responsible for that stream (or else reject the input). The amount of additional information, at each point of the parsing process, to *determine* which rule is responsible next is called the *look-ahead*. Of course, if the grammar is ambiguous, no unique decision may be possible (no matter the look-ahead).

On a very high level, the situation can be compared with the situation for regular languages/automata. Non-deterministic automata may be ok for *specifying a language* (they can more easily be connected to regular expressions), but they are not so useful for specifying a scanner *program*. There, deterministic automata are necessary. Here, grammars with left-recursion, grammars with common factors, or even ambiguous grammars may be ok for specifying a context-free language. For instance, ambiguity may be caused by unspecified precedences or non-associativity. Nonetheless, how to obtain a grammar representation more suitable to be more or less directly translated to a parser is an issue less clear-cut compared to regular languages. Already ambiguity of grammars is undecidable in general. If a grammar is ambiguous, there'd be no point in turning it into a practical parser. Also the question, what's an acceptable form of grammar depends on what class of parsers one is after (like a top-down parser or a bottom-up parser).

Example 4.4.2 (Left recursion and common left factors). We have seen various examples already for both phenomena left-recursion. Here for instance a typical production for expressions, which is left recursive.

$$exp \to exp + term \ .$$

A classical example for common left factors are rules for conditionals for instance like the following:

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if (}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ \textbf{end} \\ & \mid & \textbf{if (}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ \textbf{else}\ \textit{stmt}\ \textbf{end}\ . \end{array}$$

□

Next a expression grammar we have seen before from Example **??**. See equation (**??**).

Example 4.4.3 (Removing left recursion). The grammar from equation (**??**) is obviously left recursive. The grammar was used when discussing ambiguity, resp. how to make an ambiguous grammar unambigous by a concept called precedence cascade, which realizes associativity and operator precedences

The grammar can be formuated equivalently as follows:

$$\begin{array}{rcll} \textit{exp} & \rightarrow & \textit{term}\ \textit{exp}' & (4.12) \\ \textit{exp}' & \rightarrow & \textit{addop}\ \textit{term}\ \textit{exp}' \mid \boldsymbol{\epsilon} & \\ \textit{addop} & \rightarrow & + \mid - & \\ \textit{term} & \rightarrow & \textit{factor}\ \textit{term}' & \\ \textit{term}' & \rightarrow & \textit{mulop}\ \textit{factor}\ \textit{term}' \mid \boldsymbol{\epsilon} & \\ \textit{mulop} & \rightarrow & * & \\ \textit{factor} & \rightarrow & \textbf{(}\ \textit{exp}\ \textbf{)} \mid \textbf{number} & \end{array}$$

When saying that formulation is equivalent, we mean, the same set of words is accepted. So as far as the language (as sets of accepted words) is concerned the two grammars are equivalent. But far far as the parse trees are concerned they are not, because the *associativity* has changed!

Note also, the alternative formulation works now with $\boldsymbol{\epsilon}$-productions, which does not make it more readable. Otherwise, the grammar is still *unambiguous*.

The following Section 4.4.1 presents a systematic way to remove left-recursion from a grammar. The grammar from equation (4.12) is the result of that transformation applied to equation (**??**). □

4.4.1 Left-recursion removal

Next we present a transformation process to turn a context-free grammar into an (language-)equivalent one without left recursion. The price for that transformation will be that the transformed grammar uses (additional) $\boldsymbol{\epsilon}$-productions. Another disadvantage is that if the grammar embodies associativities, in particular left-associativity, which is connected to left-recursion, the transformation changes the associativity.

Let's start with the removal of left-recursion illustrated in a simplified situation. The following transformation removes immediate left-recursion on A in the presence of a second producuction involving A, which is assumed not to exhibit left recursion.

$$A \rightarrow A\alpha \mid \beta \qquad (4.13) \qquad \begin{array}{rcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \boldsymbol{\epsilon} \end{array} \qquad (4.14)$$

The transformation is easy enough, introducing a new non-terminal, which uses a right-recursive procedure instead. The transformation is illustrated in Figure 4.8.

(a) before (b) after

Figure 4.8: Left-recursion removal

Both grammars generate the same context-free language, i.e., the same of words over terminals. The language can be captured in EBNF as follows

$$A \to \beta\{\alpha\}$$

more concrete example for such a production: grammar for *expressions*

The previous situation was simplified in one aspect. It dealt with direct left-recursion, there was only *one* production suffering from direct left-recursion for the considered non-terminal, A in the illustration. The following definition covers the general case.

Definition 4.4.4 (Left-recursion removal: immediate recursion)**.** The transformation for removal of *immediate* left recursion is as follows:

$$\begin{array}{rcl} A & \to & A\alpha_1 \mid \ldots \mid A\alpha_n \\ & \mid & \beta_1 \mid \ldots \mid \beta_m \end{array} \qquad \begin{array}{rcl} A & \to & \beta_1 A' \mid \ldots \mid \beta_m A' \\ A' & \to & \alpha_1 A' \mid \ldots \mid \alpha_n A' \\ & \mid & \epsilon \end{array}$$

before after

Definition 4.4.5 (Left-recursion removal: immediate recursion)**.** The transformation for removal of *immediate* left recursion is as follows:

$$\begin{array}{rcl} A & \to & A\alpha_1 \mid \ldots \mid A\alpha_n \\ & \mid & \beta_1 \mid \ldots \mid \beta_m \end{array} \qquad \begin{array}{rcl} A & \to & \beta_1 A' \mid \ldots \mid \beta_m A' \\ A' & \to & \alpha_1 A' \mid \ldots \mid \alpha_n A' \\ & \mid & \epsilon \end{array}$$

before after

Now it's about indirect recursion. That's a bit more tricky. In particular it uses the removal of direct left recursion as subroutine. It's done by a nestest loop, going through all combinations of no-terminals. Actually, it goes only through "half" of them Assume non-terminals $A_1, \ldots, A_m$

Definition 4.4.6 (Left-recursion removal: indirect recursion)**.** Left recursion is removed by the code from Listing 4.5 (which also involves removing direct recursion from Definition 4.4.5).

```
for i := 1 to m do
  for j := 1 to i−1 do
    replace each grammar rule of the form A_i → A_j β by // i < j
    rule A_i → α_1 β | α_2 β | ... | α_k β
        where A_j → α_1 | α_2 | ... | α_k
        is the current rule(s) for A_j // current
  end
  { corresponds to i = j }
  remove, if necessary, immediate left recursion for A_i
end
```

Listing 4.5: Removal of left recursion (indirect case)

By "current rules" it's meant the rules at the current stage of the algorithm. Let's illustrate the procedure with a small example.

Example 4.4.7 (Left recursion removal). Consider the following grammar, which contains direct and indirect left recursion:

$$\begin{array}{lcl} A & \to & B\mathbf{a} \mid A\mathbf{a} \mid \mathbf{c} \\ B & \to & B\mathbf{b} \mid A\mathbf{b} \mid \mathbf{d} \end{array}$$

Assume further the two non-terminals ordered as $A = A_1$, $B = A_2$.

$$\begin{array}{lcl} A & \to & B\mathbf{a}A' \mid \mathbf{c}A' \\ A' & \to & \mathbf{a}A' \mid \epsilon \\ B & \to & B\mathbf{b} \mid A\mathbf{b} \mid \mathbf{d} \end{array} \qquad \begin{array}{lcl} A & \to & B\mathbf{a}A' \mid \mathbf{c}A' \\ A' & \to & \mathbf{a}A' \mid \epsilon \\ B & \to & B\mathbf{b} \mid B\mathbf{a}A'\mathbf{b} \mid \mathbf{c}A'\mathbf{b} \mid \mathbf{d} \end{array} \qquad \begin{array}{lcl} A & \to & B\mathbf{a}A' \mid \mathbf{c}A' \\ A' & \to & \mathbf{a}A' \mid \epsilon \\ B & \to & \mathbf{c}A'\mathbf{b}B' \mid \mathbf{d}B' \\ B' & \to & \mathbf{b}B' \mid \mathbf{a}A'\mathbf{b}B' \mid \epsilon \end{array}$$

With only two non-terminals, the algorithm for indirect recursion is used only in the second step. The outer loop over iterates over $i \in \{1, 2\}$, i.e., the only case where the inner loop is non-empty is for $i = 2$, in which case j can take the value i. Besides the inner loop for indirect recursion, there is also the case the treatment of direct recursion for the symbol A_i.

In the example, the first step treats direct recursion for A, the the indirect recursion for for the case where B makes use of A, in that A occurs on the left of a right-hand side of a production of B. In the example, there is one situation like that. Finally, in the last step, the direct recursion of B is treated. □

4.4.2 Left factor removal

As mentioned earlier, common left factors are undesirable. Let's look at a simple situation:

$$A \to \alpha\beta \mid \alpha\gamma \mid \ldots \tag{4.15}$$

Let's assume that the two shown rules are the only one with a common left factor α. The grammar can easily enough be transformed, "factoring out" the commong prefix α:

$$\begin{array}{lcl} A & \to & \alpha A' \mid \ldots \\ A' & \to & \beta \mid \gamma \end{array} \tag{4.16}$$

Another assumption in the situation is that α is the *longest* common prefix, otherwise the resulting production for A' would still have a common left factor. Let's look at some examples that could in this or similar form occur in the syntax of programming languages.

Example 4.4.8 (Sequence of statements). The grammar on the left represents non-empty sequences of statements, with semicolon as separator.

□

$$\begin{array}{lcl} \mathit{stmts} & \to & \mathit{stmt}\ \mathbf{;}\ \mathit{stmts} \\ & \mid & \mathit{stmt} \end{array} \qquad\qquad \begin{array}{lcl} \mathit{stmts} & \to & \mathit{stmt}\ \mathit{stmts}' \\ \mathit{stmts}' & \to & \mathbf{;}\ \mathit{stmts} \mid \boldsymbol{\epsilon} \end{array}$$

Example 4.4.9 (Conditionals). The following grammar covers conditionas with one arm only as well as conditional with two branches.

$$\begin{array}{lcl} \mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmts} \\ & \mid & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmts}\ \mathbf{else}\ \mathit{stmts} \end{array} \tag{4.17}$$

That can be rewritten as follows

$$\begin{array}{lcl} \mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmts}\ \mathit{else\text{-}or\text{-}empty} \\ \mathit{else\text{-}or\text{-}empty} & \to & \mathbf{else}\ \mathit{stmts} \mid \boldsymbol{\epsilon} \end{array} \tag{4.18}$$

As you may remember, that the last versions of the conditionals suffer from being ambigous; it's called the *dangling-else* problem. The next slight variation requires that branches are terminated by an **end**-marker. That's one way to address the dangling-else problem. Still, there is the common left factor, and the transformation is completely analogous to the previous one.

$$\begin{array}{lcl} \mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmts}\ \mathbf{end} \\ & \mid & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmts}\ \mathbf{else}\ \mathit{stmts}\ \mathbf{end} \end{array} \tag{4.19}$$

The one without a common left factor looks as follows:

$$\begin{array}{lcl} \mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmts}\ \mathit{else\text{-}or\text{-}end} \\ \mathit{else\text{-}or\text{-}end} & \to & \mathbf{else}\ \mathit{stmts}\ \mathbf{end} \mid \mathbf{end} \end{array} \tag{4.20}$$

□

The previous examples were straightforward enough, and in many concrete grammars the sketched step may do the job. In the general case, however, not all factorization doable in **one step**. The following artifical example illustrates that.

Example 4.4.10 (Left-factorization). As starting point, take the grammar on the left. Now there are three productions for a non-terminal with a common left-factor. Actually, some have a longer shared prefix, and some a shorter.

$$A \to \mathbf{abc}B \mid \mathbf{ab}C \mid \mathbf{a}E \qquad A \to \mathbf{ab}A' \mid \mathbf{a}E \qquad A \to \mathbf{a}A''$$
$$A' \to \mathbf{c}B \mid C \qquad A'' \to \mathbf{b}A' \mid E$$
$$A' \to \mathbf{c}B \mid C$$

Note: we choose the **longest left factor**, i.e., the longest common prefix in the first transformation step, not the maximal number of rules changed by the step. □

The observation just made about choosing the longest left factor is what's done one the algorithm from Listing 4.6.

```
while there are changes to the grammar do
  for each nonterminal A do
    let α be a prefix of max. length that is shared
                  by two or more productions for A
    if α ≠ ϵ
    then
      let A → α1 | ... | αn be all
                prod. for A and suppose that α1, ..., αk share α
                so that A → αβ1 | ... | αβk | αk+1 | ... | αn ,
                that the βj 's share no common prefix, and
                that the αk+1, ..., αn do not share α.
        replace rule A → α1 | ... | αn by the rules
        A → αA' | αk+1 | ... | αn
        A' → β1 | ... | βk
    end
  end
end
```

Listing 4.6: Left factorization

The algorithm is pretty straightforward. The only thing to keep in mind is that what is called α in the pseudo-code is the *longest* comment prefix and the β's must include *all* right-hand sides that start with that (common longest prefix) α.

4.5 LL-parsing, mostly LL(1)

After having covered the more technical definitions of the first and follow sets and transformations to remove left-recursion resp. common left factors, we go back to top-down parsing, in particular to the specific form of LL(1) parsing. Additionally, we discuss issues about abstract syntax trees vs. parse trees. Actually, we have not even yet said, what LL-actually stands for. LL stands for *left-to-right* parsing for a *left-most* derivation. Basically, it represents standard top-down parsing. In this lecture, we don't do LL(k) with $k > 1$. LL(1) is particularly easy to understand and to implement (efficiently). It is not as expressive than LR(1) (see later), but still kind of decent.

LL(1) parsing principle: Parse from 1) left-to-right (as always anyway), do a 2) **left-most** derivation and resolve the "which-right-hand-side" non-determinism by 3) looking **1 symbol ahead**.

We present to flavors of LL(1) resp. top-down parsing here: *recursive descent* and *table-based* LL(1) parser. They are different realizations of the same principles.

If one wants to be very precise: it's recursive descent with one look-ahead and without backtracking (i.e., it's predictive). It's the most common case for recursive descent parsers. Longer look-aheads are possible, but less common. Technically, even allowing back-tracking can be done using recursive descent as principle (even if that is mostly not done in practice).

To do top-down parsing with a look ahead (for grammars that are LL(1)-parseable) is straighforward. A look ahead of 1 is not much of a look-ahead anyway, it's just the current token. So, the parser does something rather unspectacular, it eats through the tokens left-to-right one by one, it reads the next token, looks at that **current token** and based on, makes a decision.

There's the special situation when there there are no more tokens. So the behavior is to read the next token if there is one, decide based on the token *or else* make a decision based on the fact that there's none left. One typically uses the special terminal **$** to mark the end (as mentioned in the context of the follow-sets

Example 4.5.1 (Factors and terms (again)). Let's revisit the expression grammar involving terms and factors, for instance in Example **??** or 4.4.3. Let's focus for now on the non-terminal *factor*, which is covered by 2 productions:

$$\mathit{factor} \to \mathbf{(}\ \mathit{exp}\ \mathbf{)} \ \mid\ \mathbf{number} \tag{4.21}$$

When parsing an input as factor, the decision that need to be done whether it's a parenthetic expression or a number. The decision is more or less *trivial* to do: the parser looks at the token: if it's **(**, then it's the first case and if it's **number**, the second. In the first case the parser will continue it's actions trying to parse the rest as expression *exp* and when that succeds, parsing the rest as **)**. Assuming that *factor* would be the start symbol, the parser would finally check if **)** is followed by the end-of-parse marker. In case the parse of factor does not start with **(** or **number** (or something else goes wrong later), an error will have to be raised. So, treating factors is pretty straight forward, but sometimes it will not be so obvious. □

The general setup for **recursive descent** parsing is as follows: The parser has access to the **current token**. Perhaps a variable, say, `tok` keeps that topen or a pointer to the current token. The parser can also *advance* that to the next token (if there's one). A general pattern to arrange the parse is

For each **non-terminal** *nonterm*, write one procedure (perhaps called `nonterm` or similar) which:

- succeeds, if starting at the current token position, the "rest" of the token stream starts with a syntactically correct word of terminals representing *nonterm*
- fail otherwise

The different procedures for the different non-terminals will call each other in a pattern of *mutual recursion,* reflecting the typically mutually recursive struture of the grammar. With that in mind, we have another indication that left-recursion will not be parseable, at least not in this way. For instance, a immediate left-recursive grammar rule is represented by a corresponding procedure that immediately calls itself, which leads to an infinite recursion, resp. the parser will fail with a stack-overflow (and indirect left recursion has the analogous problem).

Example 4.5.2 (Recursive descent). Let us have a look how the productions for factors from Example 4.5.1 can be parsed. In a C-like or Java like language

method `factor` for nonterminal *factor*

```
1     final int LPAREN=1,RPAREN=2,NUMBER=3,
2      PLUS=4,MINUS=5,TIMES=6;
```

Listing 4.7: Data types for the tokens

```
1     void factor () {
2         switch (tok) {
3         case LPAREN: eat(LPAREN);expr();eat(RPAREN);
4         case NUMBER: eat(NUMBER);
5         }
6     }
```

Listing 4.8: Recursive descent for factors

The code shows only the function or method for the non-terminal **factor**, calling the corresponding function for expression. The one for expressions is not shown, so in the shown code, there is no recursion visible. A more complete grammar would contain (indirect or direct) recursion on the productions, and consequently that would lead to a set of function using indirect and/or direct recursion.

In a functional language, we could formulate the procedure as follows

```
type token = LPAREN | RPAREN |  NUMBER
 |   PLUS | MINUS | TIMES
```

Listing 4.9: Data types for the tokens

```
let factor () =    (* function for factors *)
  match !tok with
    LPAREN -> eat(LPAREN); expr(); eat(RPAREN)
  | NUMBER -> eat(NUMBER)
  | _ -> ()            (* raise an error *)
```

Listing 4.10: Recursive descent for factors

□

Ignored for now has been that when doing a successful parse, the functions are supposed to give back the **abstract syntax tree** for the accepted non-terminal. Also the shown code snippets don't give back anything resp. the the value of unit-type in the ocaml version.

Before we address how to take care of abstract syntax trees, we first continue with top-down parsing itself. The situation is not always as immediate as in the example with the two rules for *factor*. That LL(1) parsing works, as said earlier, a unique decision must be possible based on the first non-terminal of words derivable from the current token. For the non-terminal *factor* with the two productions from equation (4.21), it's immediate from the two right-hand sides, which start with two different terminals. But right-hand sides can also start with non-terminals. That's exactly where the **first-sets** come in handy and that's what we discuss next.

But that's not all. It can be the case that the non-terminal in question cab derive to the ϵ, which has **no initial terminal.** Such non-terminals are called **nullable,** as introduced earlier. In such a situation, the parser additionally has to consult the **follow-set** to make the decision.

We start by ignoring nullable non-terminals and first cover the simplified setting of grammars without ϵ-productions. In this case, the decision can be made looking at the first-sets, only, in case that there is no overlap in those sets for the right-hand sides of a given non-terminal. If there's an overlap, the grammar cannot be LL-(1)-parsed

Lemma 4.5.3 (LL(1) (without nullable symbols))**.** *A reduced context-free grammar without nullable non-terminals is an LL(1)-grammar iff for all non-terminals A and for all pairs of productions $A \to \alpha_1$ and $A \to \alpha_2$ with $\alpha_1 \neq \alpha_2$:*

$$First_1(\alpha_1) \cap First_1(\alpha_2) = \emptyset \ .$$

Lemma 4.5.4 (LL(1))**.** *A reduced context-free grammar is an LL(1)-grammar iff for all non-terminals A and for all pairs of productions $A \to \alpha_1$ and $A \to \alpha_2$ with $\alpha_1 \neq \alpha_2$:*

$$First_1(\alpha_1 Follow_1(A)) \cap First_1(\alpha_2 Follow_1(A)) = \emptyset \ .$$

The characterization mentions that the grammar has to be *reduced*. We did not bother to formally define it. At some point earlier, we have said, grammars can be "silly", like containing productions that are never used or containing similar such defects. That characterization of LL(1) only applies to grammars which are not defective in that sense.

We have mentioned earlier forms of grammars where we'd expect problems. One was grammars with productions with common left factors. The other problem was left-recursion.

For the latter problem, we have repeatedly mentioned it: top-down parsing does not work for left-recursive grammars. The first glimpse why left-recursion is problematic was the example and discussion about *oracular derivations* (see Example 4.2.5). Now that we have a feeling of how one can make use of recursive procedures for top-down parsing, it's

also clear that left-recursion (direct or indirect) does not work: a procedure that *directly* calls itself in a situration of a production with direct left-recursion will diverge. Same for productions with indirect left recursion.

Now let's revisit these situations with the LL(1)-criterion at hand. For instance the grammar for conditionals from equation (4.17) from Example 4.4.9 with a **common left factor** of

$$\textbf{if (}\ exp\ \textbf{)}\ stmt$$

cannot be disambiguated with a look-ahead of one. Equation (4.18) from the same example is the left-factored version of that piece of syntax. It can also perhaps more clearly written in extended BNF as

$$\textit{if-stmt} \quad \rightarrow \quad \textbf{if (}\ exp\ \textbf{)}\ stmt[\textbf{else}\ stmt]$$

That could lead to a recursive-descend code as shown in Listing 4.11.

```
1  procedure ifstmt()
2    begin
3      match ("if");
4      match  ("(");
5      exp();
6      match (")");
7      stmt();
8      if    token = "else"
9      then match ("else");
10          stmt()
11     end
12   end;
```

Listing 4.11: Recursive descent for left-factored *if-stmt* from equation (4.18)

Likewise, left recursion is a no-go.

Example 4.5.5 (Factors and terms (again)). We can take again the expression grammar for factors and terms from Example **??**, i.e., the grammar from equation **??**. Now consider treatment of *exp*, i.e., the productions

$$exp \rightarrow exp\ addop\ term \ \mid \ term$$

Concerning the first-set of E, its clear that whatever is in $First(term)$, is in $First(exp)$. So according to the criterion from Lemma 4.5.3 the grammar is **not LL(1)**-parsable. And actually no amount of look-ahead would help.

Transforming the grammar by removing left-recursion may help. After the transformation, the production for *exp* is replaced by the following:

$$\begin{array}{lcl} exp & \rightarrow & term\ exp' \\ exp' & \rightarrow & addop\ term\ exp' \ \mid \ \epsilon \end{array}$$

See also equation (4.12) for the whole transformed expression grammar. The two non-terminal can be covered by the following procedures:

```
procedure exp()
begin
      term();
      exp'()
end
```

Listing 4.12: Procedure for *exp*

```
procedure exp'()
begin
  case token of
    "+": match("+");
         term();
         exp'()
    "-": match("-");
         term();
         exp'()
   end
end
```

Listing 4.13: Procedure for *exp′*

While this kind of massaging may help with making the grammar LL-parseable. But it's not unproblematic. One issue may be that the parse trees certainly gets more **messy.** See the one for the transformed grammar from Figure 4.4. Of course, we are talking about parse trees, not ASTs. At least for the AST, one would opt for a leaner presenttation, in particular one certainly does not want to have ϵ-nodes.

For comparison, here a parse-tree according to the grammar, where the left-recursion has not been removed (for the expression $1+2*(3+4)$). Trees like that certainly would look nicer. Of course, as said, that tree is for the grammar with left-recursion and that's bad for top-down parsing.

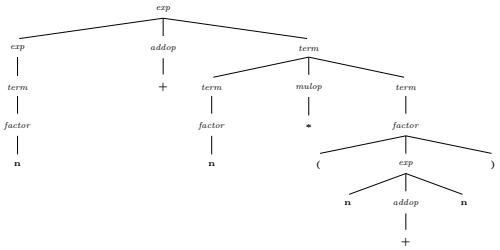


Figure 4.9: Parse tree for $1+2*(3+4)$ for the left-recursive version of the grammar

Parse trees from the "flat" formulation of the grammar, without the cascade of terms and factors, look even nicer (see Figure 4.10). But then the grammar is ambiguous, and thus likewise unsuitable for top-down parsing, resp. parsing in general.

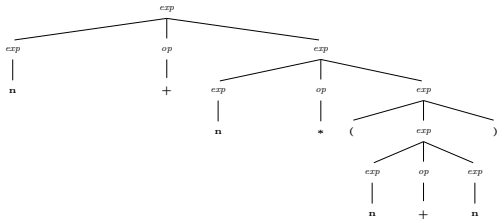
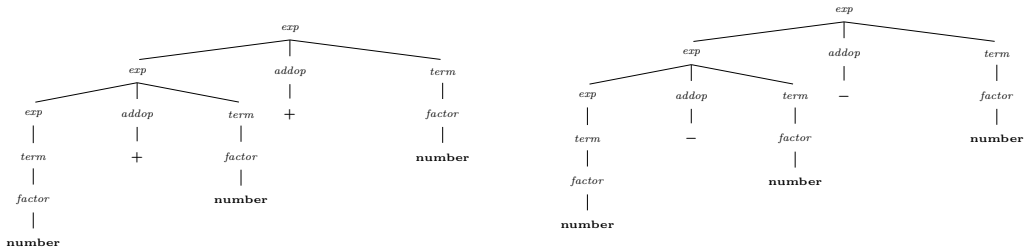


Figure 4.10: Parse tree for $1+2*(3+4)$ for the "flat" grammar

□

Nicer or less nice is one thing, but there is another more serious problem. We know already that, when parsing binary operations like the ones used for expression here, there is a connection between **associativity** and the form of recursion. For right-recursive formulations, the operators associate to the right, analogously for left associativity. That was dicussed in connection with the concept of *precedence cascade*. If a production uses left- and right-recursion, then the operation can associate to the left and the right, which makes the grammar ambigous. The precedence cascade used the expressions, factors and terms only in a left-recursive manner, which fixed the associativity as intended, namely to the left, and additionally fixed the precedences.

Cf. again the expression-terms-factor grammar from Example **??**. So the left recursive grammar parses an expression $3+4+5$ resp. $3-4-5$ as "as" $(3+4)+5$ resp. $(3-4)-5$. The corresponding trees are shown in Figure 4.11.



(a) $3+4+5$

(b) $3-4-5$

Figure 4.11: Factors and terms: left-associative parse trees

Thus, it should not come as surprise that the repair concerning left-recursion causes headache: the repair replaces left-recursion by right-recursion (using additional non-terminals and ϵ-productions. That changes the associativity from left- to right-associativity. I.e., the given expressions are now parsed "as" $3+(4+5)$ and $3-(4-5)$.

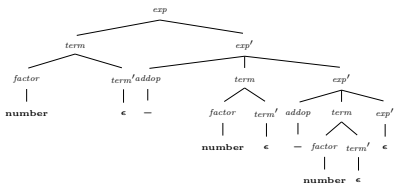


Figure 4.12: Factors and terms: right-associative parse tree for $3-4-5$

There's not much one can do about, as far as the parse-trees are concerned: right-recursion leads to a form of the parse tree that corresponds to righ-associativity. There is, however, something else one can do. One is not actually interested in the parse-tree, what the parser gives back is the **abstract syntax** tree. Thus, one needs a way to build a left-associative AST from a recursively descending parser run, that correspond to a right-associative parse tree.

4.5.1 On the design of ASTs and how to build them

So far we have focused on parse-trees. In the following section we not only discusse how to repair situations where the parse trees reflect an unwanted associativity. We also take the opportunity to say a bit about abstract syntax trees *in general*. Abstract syntax trees are not specific for top-down or recursive descent parsing, so those remarks are independent from that particular parsing style. But let's start by seeing how to repair associativity.

How to repair ill-associated parse-trees and more on ASTs

We have seen situations when the associativity of a parse tree does not match the intended one. Building an abstract syntax tree straight-forwardly reflecting more or less the parse tree thus also leads to an ill-associated tree. That can be repaired by handing over an extra argument to the recursive procedures of the top-down parser.

We illustrate the mechanism *not* for the situation when the parser returns an AST, but for **evaluating** the expression. So, returned is an integer, not a tree, and likewise the additional argument we mentioned will be an integer, not a tree.

Example 4.5.6 (Evaluating a expression with correct associativity). We use the same right-recursive expression-term-factor-grammar from before.

As illustration, we use the expression

$$3 - 4 - 5$$

which should be parsed, resp. evaluated as $(3 - 4) - 5$ *not* as $3 - (4 - 5)$. If we had the left-associative parse-tree from Figure 4.11b, evaluating the expression would be very easy. However, the parse-tree represents right-associativity, see Figure 4.12 and a straightforward, bottom-up evaluation will give the wrong result. The idea of the correct evalution is shown in Figure 4.13.

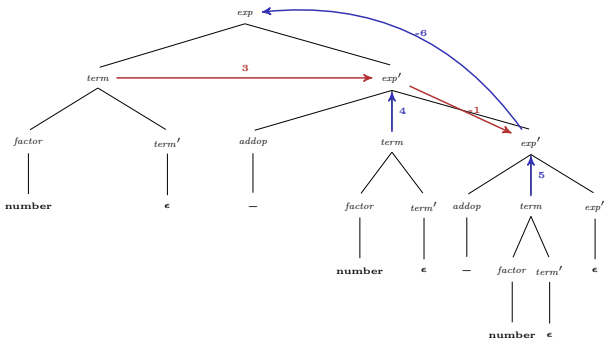


Figure 4.13: Repairing associativity (evaluation)

In the picture, the red arrows show the **extra argument** as **input** to the recursion, and the blue one, going upwards, the returned **result**. In this case, evaluating the expressions, the extra information is integers, as said.

The code for that idea is given in Listing 4.14 (at leat the part dealing with *exp′*).

```
function exp' (valsofar: int): int;
begin
  if token = '+' or token = '-'
  then
    case token of
      '+': match ('+');
           valsofar := valsofar + term;
      '-': match ('-');
           valsofar := valsofar - term;
    end case;
    return exp'(valsofar);
  else return valsofar
end;
```

Listing 4.14: Code to **evaluate** ill-associated such trees correctly

□

The example used as extra "accumulator" argument `valsofar`, representing the accumulated integer value. Instead of evaluating the expression, one could build the AST with the appropriate associativity instead: instead of `valsofar`, one had `rootoftreesofar`.

The example parses expressions and *evalutes* them while doing that. In most cases in a full-fledged parser, one does not need a numeric value as output of a successful parse-run, but an AST. But the issue of the fact, that sometimes the associativity is "the wrong way". Also the "accumulator"-pattern illustrated here in the evaluation setting could help out with AST

Some more words on ASTs, while at it. We have discussed the difference between parse trees and AST earlier and mentioned it's a design issue, and trade-offs can be made.

Before looking at ASTs, let's start by talking about general **design issues** concerning a languages, resp. a language's syntax.

It starts already with the design of the language itself, it's concrete syntax: How much of the syntax is left implicit? **Lisp** for instance, is famous/notorious in that its surface syntax is more or less an explicit notation for the ASTs.[1] It also depends on which grammar class to use: Is LL(1) good enough (or LALR(1)), or does one feel the need for something more expressive. Does one parse top-down or bottom?

Once, the language and the grammar is fixed, the parse trees are fixed, as well. The parse trees are the essence of the parsing process in a given grammar, and the abstract syntax trees have to capture the essence of the parse trees. The ASTs are typically built **on-the fly**, i.e., built while the parser parses.

> the parser "**builds**" the AST data structure while "**doing**" the parse tree.

Since the AST is the only thing relevant for later phases, it's a good idea to have *clean* structure; we mentiond that earlier. It's also possible to have the AST being basically equivalent to the CST, in which case the AST becomes straightforward. It's a possible choice, **if** the grammar for parsing is not designed "weirdly".

[1] Not that it was originally planned like this ... One can see some comments by McCarthy at `http://www-formal.stanford.edu/jmc/history/lisp/lisp.html`.

For instance, looking at the parse tree of Figure 4.12, even if we would accept that it's right-associative, with all the ϵ nodes and inner nodes representing $term'$ and exp', it's not a nice structure. So, parse trees like that, even if their associativity is acceptable, are better cleaned up as AST.

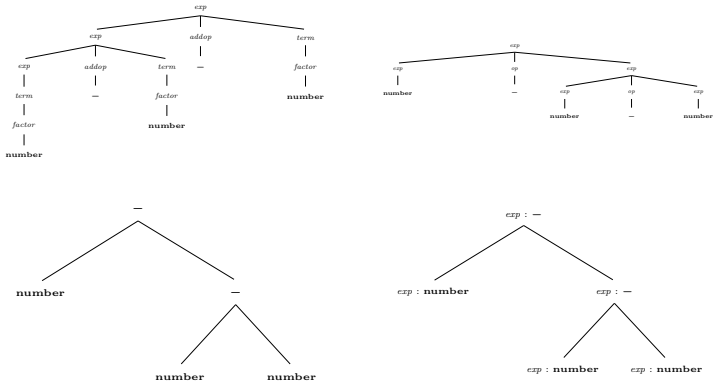


Figure 4.14: Different more or less abstract trees

But anyway, what's shows are anyway illustrative pictures how one can imagine how an AST could look like. In a concrete **implementation,** one has to choose and define concrete data types, classes etc., that realize those trees. In the following we sketch how one could do it in Java, or another class-based object-oriented language with inheritance and subtyping. It's based on using *abstract* classes for non-terminals, and concrete classes for different productions for a given non-terminal. Those classes, one for each production, *extend* the corresponding abstract class. In functional languages, one would one inductive data types instead (most functional languages support those).

Let's use expressions again, and assume, one has a "non-weird" grammar. We have seen that formulation in the previous chapter and repeat it here.

$$\begin{array}{rcl} exp & \rightarrow & exp\ \ op\ \ exp \mid (\ exp\ ) \mid \mathbf{number} \\ op & \rightarrow & + \mid - \mid * \end{array}$$

Abstract syntax trees are trees, like parse trees as well (only more abstract), so here we use the BNF notation not as specification for the parser, but as desciption of the structure of **abstract syntax trees**. Therefore it does not bother us that the grammar is highly ambiguous and not useful for parsing. To represent that as data structure, **this is how it's done:**

Recipe:

- turn each **non-terminal** to an **abstract class**.
- turn each **right-hand** side of a given non-terminal as (non-abstract) **subclass** of the class for considered non-terminal.
- chose fields & constructors of concrete classes appropriately.
- **terminal**: concrete class as well, field/constructor for token's *value*.

Following that recipe results in the following code. By choosing the classes to be public, each class would have to reside in a separate file, of course, like `Exp.java`, `BinExp.java` etc. One could do that also differently if preferred. But it's one way to routinely capture a grammar as shown, almost without thinking.

```
abstract public class Exp {
}
```

```
public class BinExp extends Exp {  // exp -> exp op exp
   public Exp left, right;
   public Op  op;
   public BinExp(Exp l, Op o, Exp r) {
       left=l; op=o; right=r;}
}
```

```
public class ParentheticExp extends Exp {  // exp -> ( op )
   public Exp exp;
   public ParentheticExp(Exp e) {exp = l;}
}
```

```
public class NumberExp extends Exp {  // exp -> NUMBER
    public  number;                   // token value
    public Number(int i) {number = i;}
}
```

```
abstract public class Op {   // non-terminal = abstract
}
```

```
public class Plus  extends Op {  // op -> "+"
}
```

```
public class Minus  extends Op {  // op -> "-"
}
```

```
public class Times extends Op {  // op -> "*"
}
```

Having a recipe as guiding line is nice, one might however choose not to follow it slavishly. For instance, the last classes, one class per operator, are perhaps pushing it too far, one could also choose to solve that differently and perhaps one get better efficiency if one would not make classes / objects out of everything, though. Here it'sdone to show that one can mechanically use the *recipe* once grammar is given, so it's a clean solution .

```
Exp e =  new BinExp(
           new NumberExp(3),
           new Minus(),
           new BinExp(new ParentheticExpr(
                new NumberExp(4),
                new Minus(),
                new NumberExp(5))))
```

Listing 4.15: AST for $3 - (4 - 5)$

For course, one could do some pragmatic deviations from the recipe, It's nice to have a guiding principle, but no need to carry it too far . . . To the very least: the `ParentheticExpr` is completely without purpose: grouping is captured by the tree structure, so that class is *not* needed. Some might prefer an implementation of the operators

$$op \to + \mid - \mid *$$

as simply integers, for instance arranged as follows

```
public class BinExp extends Exp {  // exp -> exp op exp
   public Exp left, right;
   public int  op;
   public BinExp(Exp l, int o, Exp r) {
     pos=p; left=l; oper=o; right=r;}
   public final static int PLUS=0, MINUS=1, TIMES=2;
}
```

and used as `BinExpr.PLUS` etc.

Some final words for the recipe. Space considerations for AST representations are not top priority nowadays in most cases. In my eyes, **clarity** and **cleanness** trumps quick hacks and "squeezing bits". To which extent one follows the recipe or at all:

> **Do it systematically:** A clean grammar is **the** specification of the syntax of the language and thus the parser. It is also a means of **communicating** with humans what the syntax of the language is, at least communicating with pros, like participants of a compiler course, who of course can read BNF ... A clean grammar is a very systematic and structured thing which consequently *can* and *should* be **systematically** and **cleanly** represented in an AST, including judicious and systematic choice of names and conventions (the non-terminal *exp* is represented by class `Exp`, the non-terminal *stmt* by class `Stmt`, etc.)

Building ASTs

Let's have a look how to produce "something" during recursive descent parsing. So far we look at recursive-descent, i.e., a top-down (parse-)tree traversal via recursive procedures.[2] The possible outcome was termination or failure. Now: instead of returning "nothing" (return type `void` or similar), we want to return some meaningful, and build that up during traversal.

Example 4.5.7 (Evaluating an *exp* during RD parsing). For illustration, let's sketch a procedure for **evaluating** expression. So after a successful parse, a integer value is returned.

```
function exp() : int;
var temp: int
begin
  temp := term ();        { recursive call }
  while token = "+" or token = "-"
    case token  of
      "+": match ("+");
           temp := temp + term();
      "-": match ("-")
           temp := temp - term();
    end
  end
  return temp;
end
```

Listing 4.16: Evaluating expressions

[2] Modulo the fact that the tree being traversed is "conceptual" and not the input of the traversal procedure; instead, the traversal is "steered" by stream of tokens.

□

Conceptually, building of an AST is not much harder

```
function exp() : syntaxTree;
var temp, newtemp: syntaxTree
begin
  temp := term ();        { recursive call }
  while token = "+" or token = "-"
    case token  of
      "+": match ("+");
           newtemp := makeOpNode("+");
           leftChild(newtemp)  := temp;
           rightChild(newtemp) := term();
           temp := newtemp;
      "-": match ("-")
           newtemp := makeOpNode("-");
           leftChild(newtemp)  := temp;
           rightChild(newtemp) := term();
           temp := newtemp;
    end
  end
  return temp;
end
```

Listing 4.17: Building ASTs for expression

note: the use of `temp` and the `while` loop

$$\textit{factor} \rightarrow \textbf{(}\ \textit{exp}\ \textbf{)}\ \mid\ \textbf{number}$$

```
function factor() : syntaxTree;
var fact: syntaxTree
begin
  case token  of
    "(": match ("(");
         fact := exp();
         match (")");
    number:
        match (number)
        fact := makeNumberNode(number);
     else : error ...    // fall through
  end
  return fact;
end
```

Listing 4.18: Building and AST (factor)

$$\textit{if-stmt} \rightarrow \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ [\textbf{else}\ \textit{stmt}]$$

```
function ifStmt() : syntaxTree;
var temp: syntaxTree
begin
  match ("if");
  match ("(");
  temp := makeStmtNode("if")
  testChild(temp) := exp();
  match (")");
  thenChild(temp) := stmt();
  if   token = "else"
```

```
11    then match "else";
12         elseChild(temp) := stmt();
13    else elseChild(temp) := nil;
14    end
15    return temp;
16  end
```

Listing 4.19: Building ASTs (conditionals)

In summary: the recursive descent parser do have one procedure/function/method per each specific non-terminal. It decides on alternatives, looking only at the current token. Upon entry of function `A` for non-terminal A, the **first** terminal symbol for A in `token`. Upon exit of the procedure, the parser is at the first terminal symbol *after* the unit derived from A in `token`. `match("a")` : checks for `"a"` in `token` *and eats* the token (if matched).

The fact that the parsing proceeds based on the first terminal in each procedure corresponds to the fact that LL(1)-parseability. We have a closer look at conditions when LL(1) parsing is possible in the following.

4.5.2 LL(1) parsing principle and table-based parsing

For the rest of the top-down parsing section, we look at a "variation", not as far as the principle is concerned, but as far as the implementation is concerned. Instead of making a recursive solution, one condenses the relevant information in tabular form. This data structure is called an **LL(1) table**. That table is easily constructed making use of the *First*- and *Follow*-sets, and instead of mutually recursive calls, the algo is iterative, manipulating an explicit stack. As a look forward: also the bottom-up parsers will make use of a table (which then will be an LR-table or one of its variants, not an LL-table).

Now, instead using recursion, as before, we use now an *explicit stack* and the decision-making based on the next terminal symbol is collated into the so-called **LL(1) parsing table**. Remember also the characterization of LL(1)-parseable grammars from Lemma 4.5.3 (for grammars without ϵ-productions).

The LL(1) parsing table is a finite data structure M, for instance a two-dimensional array

$$M : \Sigma_N \times \Sigma_T \to ((\Sigma_N \times \Sigma^*) + \texttt{error})$$

For looking up entries, we use the notation $M[A, a] = w$. In the following, we assume pure BNF, not any of the extended forms.

Side remark 4.5.8. Sometimes, depending on the book, the entry in the parse table does not contain a full rule as here, needed is only the *right-hand-side*. In that case the table is of type $\Sigma_N \times \Sigma_T \to (\Sigma^* + \texttt{error})$. □

Construction of the parsing table

We present two versions of the same recipe. The second one is based on the concept of the first- and follows sets.

Earlier, we have have seen a characterization of when a grammar is LL(1) parsable (see Lemma 4.5.3 for the restricted setting of a grammar without ϵ). One can see the recipe here, how to make a LL(1) table as another, but equivalent way of figuring whether a grammar is LL(1) or not, namely: build a table following the recipe, and if that table contains **double-entries,** then it's not LL(1) parseable. A recursive decent parser, using that table, would not know what to do. Such a double entry is called a conflict, here an **LL(1)-conflict**.

Remember that we are restricting us to *predictive* parsing. One can have a recursive descent parser using *backtracking* that would explore alternatives, but that's mostly avoided in parsing practice.

1. If $A \to \alpha \in P$ and $\alpha \Rightarrow^* \mathbf{a}\beta$, then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$
2. If $A \to \alpha \in P$ and $\alpha \Rightarrow^* \boldsymbol{\epsilon}$ and $S\,\mathbf{\$} \Rightarrow^* \beta A \mathbf{a} \gamma$ (where $\mathbf{a}$ is a token (=non-terminal) *or* $\mathbf{\$}$), then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$

Table 4.4: Recipe for LL(1) parsing table

Assume $A \to \alpha \in P$.

1. If $\mathbf{a} \in \mathit{First}(\alpha)$, then add $A \to \alpha$ to $M[A, \mathbf{a}]$.
2. If α is *nullable* and $\mathbf{a} \in \mathit{Follow}(A)$, then add $A \to \alpha$ to $M[A, \mathbf{a}]$.

Table 4.5: Recipe for LL(1) parsing table

The two recipes are *equivalent*. One can use the recipes to fill out LL(1) table, we will do that in the following. In case a slot in such a table means that the grammar is not LL(1)-parseable, i.e., the LL(1) parsing principle is violated. One may compare that also to Lemma 4.5.3.

Example 4.5.9 (If-statements). Consider the following grammar for conditionals:

$$\begin{array}{rcl} \mathit{stmt} & \to & \mathit{if\text{-}stmt} \mid \mathbf{other} \\ \mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt}\ \mathit{else{-}part} \\ \mathit{else{-}part} & \to & \mathbf{else}\ \mathit{stmt} \mid \epsilon \\ \mathit{exp} & \to & \mathbf{0} \mid \mathbf{1} \end{array}$$

The grammars is left-factored and not left-recursive. See also the corresponding grammars from Example 4.4.9.

The table lists the first and follow set for all non-terminals (as was the basic definition for those concepts). In the recipe, though, we actually need the first-set of *words*, namely for the right-hand sides of the productions (for the follow-set, the definition for non-terminals is good enough). Therefore, one might, before filling out the LL(1)-table also list the

	First	*Follow*
stmt	**other**, **if**	**$**, **else**
if-stmt	**if**	**$**, **else**
else−part	**else**, ϵ	**$**, **else**
exp	**0**, **1**	**)**

Table 4.6: First- and follow sets

first set of all right-hand sides of the grammar. On the other hand, it's not a big step, especially in this grammar.

With this information, we can contruct the LL(1) parsing table.

M[N, T]	`if`	`other`	`else`	0	1	$
statement	*statement* → *if-stmt*	*statement* → `other`				
if-stmt	*if-stmt* → `if` `(` *exp* `)` *statement* *else-part*					
else-part			*else-part* → `else` *statement* *else-part* → ε			*else-part* → ε
exp				*exp* → `0`	*exp* → `1`	

Table 4.7: LL(1) parsing table

The highlighted slot contains two productions. Thus it's not a proper an LL(1) table, and it's not an LL(1) grammar. Note therefore, temoving left-recursion and left-factoring did not help! □

Side remark 4.5.10. Saying that it's "not-an-LL(1)-table" is perhaps a bit nitpicking. The shape *is* according to the required format. It's only that in one slot, there is more than one rule, actually tow. That's a *conflict* and makes it at least not a legal LL(1) table. So, if in an exam question in a written exam, the task is "build the LL(1)-table for the following grammar ..... Is the grammar LL(1)?". Then one is supposed to fill up a table like that, and then point out, if there is a double entry, which is the symptom that the grammar is not LL(1), that this is the case. □

Similar remarks apply later for LR-parsers and corresponding tables. Actually, for LR-parsers, tools like `yacc` build up a table (not an LL-table, but an LR-table) and, in case of double entries, making a choice which one to include. The user, in those cases, will receive a warning about the grammar containing a corresponding *conflict*. So the user should be aware that the grammar is problematic.

Conflicts are typically to be avoided, though upon analyzing it carefully, there may be cases, were one can "live with it", that the parser makes a particular choice and ignore

another. What kind of situations might that be? Actually, the one here in the example is one. The given grammar "suffers" from the ambiguity called **dangling-else problem.** Anyway, the conflict in the table puts the finger onto that problem: when trying to parse an else-part and seeing the `else`-keyword next, the top-down parser would not know, if the else belongs to the last "dangling" conditional or to some older one (if that existed). Typically, the parser would choose the *first* alternative, i.e., the first production for the else-part. If one is sure of the parser's behavior (namely always choosing the first alternative, in case of a conflict) and if one convinces oneself that this is the intended behavior of a dangling-else (in that it should belong to the last open conditional), then one may "live with it". But it's a bit brittle.

With an LL(1)-table at hand, it's straighforward to formulate an algorithm based on that, see Listing 4.20.

```
push the start symbol of the parsing stack;
while the top of the parsing stack ≠ $
  and the next input ≠ $
  if the top of the parsing stack is terminal a
     and the next input token = a
  then
     pop the parsing stack;
     advance the input; // ``match'' ``eat''
  else if   the top the parsing is non-terminal A
        and the next input token is a terminal or $
        and parsing table M[A, a] contains
            production A → X1X2 ... Xn
        then (* generate *)
            pop the parsing stack
            for i := n to 1 do
            push Xi onto the stack;
       else error
  if   the top of the stack = $
       and the next input token is $
  then accept
  else error;
end
```

Listing 4.20: LL(1) table-based algorithm

The most interesting steps are of course those dealing with the dangling else situation. Those are the line with the non-terminal *else−part* at the top of the stack and the **else** as next input, (in the picture, abbreviated as L and **e**). That's where the LL(1) table is ambiguous. In principle, with *else−part* on top of the stack , the parser table allows always to make the decision that the "current statement" resp "current conditional" is done. In the derivation, we see the choice which is done: the reduction picks *else−part* → **else** *stmt* and not the alternative *else−part* → ϵ. That matches the slot in the table, under the assumption that the algo takes the first one listed, and ignores later alternatives ("first match"). As far as the dangling else situation is concerned, this leads to the intended behavior: an else part associates to the last open if-construct.

Let's do another example; more can be found in the exercises and earlier written exams.

Parsing stack	Input	Action
$ S	i(0)i(1)o e o$	$S \rightarrow I$
$ I	i(0)i(1)o e o$	$I \rightarrow$ i (E) S L
$ L S) E (i	i(0)i(1)o e o$	match
$ L S) E (	(0)i(1)o e o$	match
$ L S) E	0)i(1)o e o$	$E \rightarrow$ 0
$ L S) 0	0)i(1)o e o$	match
$ L S)	)i(1)o e o$	match
$ L S	i(1)o e o$	$S \rightarrow I$
$ L I	i(1)o e o$	$I \rightarrow$ i (E) S L
$ L L S) E (i	i(1)o e o$	match
$ L L S) E (	(1)o e o$	match
$ L L S) E	1)o e o$	$E \rightarrow$ 1
$ L L S) 1	1)o e o$	match
$ L L S)	)o e o$	match
$ L L S	o e o$	$S \rightarrow$ o
$ L L o	o e o$	match
$ L L	e o$	$L \rightarrow$ e S
$ L S e	e o$	match
$ L S	o$	$S \rightarrow$ o
$ L o	o$	match
$ L	$	$L \rightarrow \varepsilon$
$	$	accept

Table 4.8: Illustration of a run of the algo

Example 4.5.11 (Expressions). The example is the expression grammar, namely the version where the precedences are ok and left recursions has been replaced by right-recursion, i.e., the frammar from equation (4.12) in Example 4.4.3.

The first- and follow sets are summarized in the following table. □

	First	*Follow*
exp	(, **number**	**$**,)
exp'	+, −, ϵ	**$**,)
addop	+, −	(, **number**
term	(, **number**	**$**,), +, −
$term'$	∗, ϵ	**$**,), +, −
mulop	∗	(, **number**
factor	(, **number**	**$**,), +, −, ∗

Table 4.9: First- and follow-sets

4.6 Error handling

The error handling section is not part of the pensum (it never was), insofar it will not be asked in the exam, written or otherwise. That does not mean that, that we don't want some halfway decent error handling for the compiler in the oblig. The slides are not presented in detail in class. Parsers (and lexers) are built on some robust, established and well-understood theoretical foundations. That's less the case for how to deal with errors, where it's more of an art, and more pragmatics enter the pictures. It does not mean it's

M[N, T]	(	number	)	+	-	*	$
exp	*exp* → *term exp'*	*exp* → *term exp'*					
exp'			*exp'* → ε	*exp'* → *addop term exp'*	*exp'* → *addop term exp'*		*exp'* → ε
addop				*addop* → +	*addop* → -		
term	*term* → *factor term'*	*term* → *factor term'*					
term'			*term'* → ε	*term'* → ε	*term'* → ε	*term'* → *mulop factor term'*	*term'* → ε
mulop						*mulop* → *	
factor	*factor* → (*exp*)	*factor* → **number**					

Table 4.10: LL(1) parse table

unimportant, it's just that the topic is less conceptually clarified. So, while certainly there is research on error handling, in practice it's mostly done "by common sense". Parsers (and compilers) can certainly be tested systematically, finding out if the parser detects all syntactically erroneous situations. Whether the corresponding feedback is useful for debugging, that is a question of whether humans can make sense out of the feedback. Different parser technologies (bottom-up vs. top-down for instance) may have different challenges to provide decent feedback. One core challenge may be the disconnect between the technicalities of the internal workings of the parser (which the programmer may not be aware of) and the source-level representation. A parser runs into trouble, like encountering an unexpected symbol, when currently looking at a field in the LL- or LR-table. That constitutes some "syntactic error" and should be reported, but it's not even clear what the "real cause" of an error is. Error localization as such cannot be formally solved, since one cannot properly define was the source of an error is in general. So, we focus here more on general "advice".

At the least: do an understandable error message, i.e., give an indication of line / character or region responsible for the error in the source file. Potentially, *stop* the parsing. Some compilers do **error recovery.** Also there, give an understandable error message (as minimum), continue reading input , until it's plausible to resume parsing. That allows to find more errors in one compilation attempt. Of course, when finding at least one error, there's no code to generate. It's fair to say, that resuming in a meaningful way after a syntax error is not easy.

Concerning **error messages**: report errors as early as possible, if possible at the first point where the program cannot be extended to a correct program. If doing error concurrency, try to avoid error messages that only occur because of an already reported error! Make sure that, after an error, one doesn't end up in a infinite loop without reading any input symbols.

It is not alway clear what a good error message is, resp. find a way that clearest for the programmer under all circumstances. As illustration, assume: that the recursive parse function `factor()` chooses the alternative (*exp*) but that it, when control returns from method `exp()`, does not find a parenthesis). That's of course a syntax error, and the parser could report

```
right paranthesis missing
```

That sounds clear enough, but it can also be confusing. What if the program text is

```
( a + b c )
```

often be confusing, e.g. if what the program text is:

Here the `exp()` parse function will terminate after `( a + b`, as `c` cannot extend the expression. But the syntax error is not repaired by adding a right-parenthesis. A better message could be `error in expression or right paranthesis missing`.

4.7 Bottom-up parsing

4.7.1 Introduction

This quite big section covers the second general class of parsers, the bottom-up ones. As indicated in Figure 4.2, with the same amount of look-ahead, bottom-up parsers are more powerful than their top-down counter-parts. Bottom-parsers or LR-parsers are probably the most common and default parser technology, realized by tools like yacc (and CUP ...). While top-down parsing and recursive descent-parsers are conceptually straightforward, LR-parsing is a more subtle. Another name for that kind of parsing is **shift-reduce** parsing

Let's start with what LR abbreviates: The "L" stands, as in LL, for left-to-right and

> the "R" in LR stands for *right-most* derivation.

We cover different LR-parsers, starting with LR(0), i.e. shift-reduce parsing without look-ahead. All versions are based on the same principles and basically the same construction, but they differ in the amount of look-ahead, resp. the way they deal with the available information.

LR-parsing and its subclasses

The LR(0) and the LR(1) are the pure forms: no look-ahead and a look-ahead of one. The LR(0) class is quite weak and thus not very useful (but not as ridiculously weak as LL(0)). Still we cover it because the principles of shift-reduce parsing can be understood already in this situation.

Besides the two "pure" forms LR(0) and LR(1), we discuss two variations, SLR and LALR(1). They are positioned in their expressiveness between LR(0) and LR(1), and in particular LALR(1) is the method of choice, underlying yacc and friends.

Table 4.11 shows an overview over the techniques we cover. The information about LR(0) seems a bit contradictory: if LR(0) is so weak that it works only for unreasonably simple languages, why is it said that LR(0) automata for *standard* languages have 300 states or so; standard languages don't use LR(0)? The answer is, the other more expressive parsers

LR(0)	• only for very simple grammars • approx. 300 states for standard programming languages • only as warm-up for SLR(1) and LALR(1)
SLR(1)	• expressive enough for most grammars for standard PLs • same number of states as LR(0)
LALR(1)	• slightly more expressive than SLR(1) • same number of states as LR(0) • we look at ideas behind that method, as well
LR(1)	• covers all grammars, which can in principle be top-down parsed by looking at the next token

Table 4.11: LR or shift-reduce parsers

(SLR(1) and LALR(1)) use the *same* number of states, that's why one can estimate the number of states, even if standard languages don't have an LR(0) parser; they may have an LALR(1)-parser, which has, it its core, LR(0)-states.

4.7.2 Principles of bottom-up resp. shift-reduce parsing

Top-down parsers like LL(1) can straightforwardly be hand-coded. LR parsing is more subtle and typically for those, one relies on *tool-support*. Typical tools are the parser generators like yacc and friends (bison, CUP, etc.)

That style of parsing is based on a *parsing table* and an explicit *stack*. Thankfully, unlike in LL-parsing, *left-recursion* is no longer problematic.

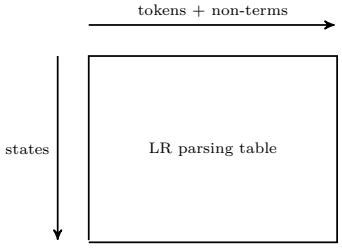


Figure 4.15: LR parsing table (schematic)

Also for top-down parsers, we looked at table-based realizations. There we were talking about LL(1)-tables. Those were a different angle on the idea of recursively descending the parse-tree "following" the parse-tree from the root to the leaves. In some general way, the tables here will play a similar role: they are consulted by the parser, determining the

reaction, which can be to read an input and proceed, pushing the input to the stack, or removing symbols from the stack. The format of the LR-tables is different, though. We will see the exact format later, but already on this slide, we see that the lines corresponds to states and the columns to terminals and non-terminals (i.e., tokens).

Indeed, it probably went unnoticed: very abstractly, we mentioned that context-free grammars can be parsed by so-called push-down automata or stack-automata, i.e., finite-state automata with an aditional stack. The illustrative pictures showed that those automata read input can manipulate the stack and move between their states until reaction. This section will describe in detail how the LR-table is filled.

Example 4.7.1 (Example grammar). Let's take an artificial grammar for illustration.

$$\begin{array}{lcl} S' & \to & S \\ S & \to & AB\mathbf{t_7} \mid \ldots \\ A & \to & \mathbf{t_4}\mathbf{t_5} \mid \mathbf{t_1}B \mid \ldots \\ B & \to & \mathbf{t_2}\mathbf{t_3} \mid A\mathbf{t_6} \mid \ldots \end{array} \tag{4.22}$$

Assume the grammar unambigous and let's take as input the word

$$\mathbf{t_1}\mathbf{t_2}\ldots\mathbf{t_7}$$

Generally, we assume for bottom-up parsing that the start symbol *never* occurs on the right-hand side of a production. If that happens we add another "extra" start-symbol, (here S'). The fact that the start symbol *never* occurs on the right-hand side of a production will later be relied upon when constructing a DFA for "scanning" the stack, to control the reactions of the stack machine. This restriction leads to a unique, well-defined initial state. All goes just smoother (and the construction of the LR-automaton is slightly more straighforward) if one obeys that convention.

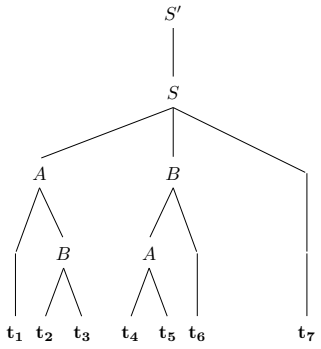


Figure 4.16: Parse tree for $\mathbf{t_1}\ldots\mathbf{t_7}$

□

Let's start by recalling the definitions of derivations and of sentential forms. A word from Σ^* derivable from the start-symbol is called a sententional form. Consequently, if derivable with a right-most derivation, it's called a **right-sentential form:**[3]

$$S \Rightarrow_r^* \alpha \tag{4.23}$$

[3] We stated that it's irrelevant whether one uses a left-most or right-most derivation, or some other (see Lemma 4.2.4. But that was for words of terminals, here we are talking about sentential forms.

It may sound puzzling at first, the parser eats through the input from left-to-right, that's a given. That means, when building up the parse tree, things happens on the left of the tree before they happens on the right, so how can that correspond to a right-most derivation? The answer is simple. The notion of *derivation* (left-most, right-most, or otherwise) is top-down: replacing non-terminals (parent nodes) by right-hand sides (children-nodes). But, when doing a left-to-right parse, the bottom-up parser (implicitly) builds tree the other way, starting from the leaves. It progresses from the left and the lower parts of the parse tree to the parts on the right and the ones higher up, until reaching the start-symbol at the root, in a successful parse. In other words, when doing the parse, the parser (implicitly) builds a **right-most** derivation **in reverse**, as the parse progresses bottom-up.

Let's illustrate that on some small example, one of the expression grammars once again

Example 4.7.2 (Building the parse tree). Let's revisit one version of the expression grammar, the one with terms and factors. A parse tree of a simple expression like $1 + 2$ is shown in Figure 4.17.

The slides show in a series of overlays, how the parse-tree is growing, here we only show the completed parse-tree. The parser processes the input **number** $*$ **number**, which are the leaves of the parse tree, and it treats the terminal leaves from left to right, of course. Doing so, it builds up the parse-tree bottom up, until until having reached the start symbol as root of the tree. That stepwise process of transforming the input into the start-symbols is called **reduction.** That's the reverse direction compared to how one can use grammars to generate or **derive** words and which corresponds to the direction of how top-down parsers work.

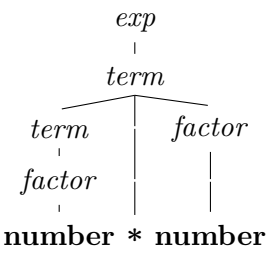


Figure 4.17: Parse tree

See Figure 4.18 for the processes of reduction resp. derivation. The underlined parts

$$
\begin{array}{lcl}
\underline{\mathbf{n}} * \mathbf{n} & \hookrightarrow & \underline{\mathit{factor}} * \mathbf{n} \\
 & \hookrightarrow & \mathit{term} * \underline{\mathbf{n}} \\
 & \hookrightarrow & \underline{\mathit{term} * \mathit{factor}} \\
 & \hookrightarrow & \underline{\mathit{term}} \\
 & \hookrightarrow & \mathit{exp}
\end{array}
$$

(a) Reduction

$$
\begin{array}{lcl}
\mathbf{n} * \mathbf{n} & \Leftarrow_r & \underline{\mathit{factor}} * \mathbf{n} \\
 & \Leftarrow_r & \underline{\mathit{term}} * \mathbf{n} \\
 & \Leftarrow_r & \mathit{term} * \underline{\mathit{factor}} \\
 & \Leftarrow_r & \underline{\mathit{term}} \\
 & \Leftarrow_r & \underline{\mathit{exp}}
\end{array}
$$

(b) Right derivation

Figure 4.18

are different in the reduction and the derivation, but they represent in both cases the "part being replaced" in the corresponding step. The derivation replaces in each step the right-most non-terminal, i.e., it's a right-most derivation. Consequently are all intermediate words *right-sentential forms* (of course only in case of a successful parse). For the reduction, the underlined part is connected with the notion of **handle**. □

So, the example illustrated:

reduction in reverse = right-most derivation

Definition 4.7.3 (Handle)**.** Assume $S \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha\beta w$. A production $A \to \beta$ at position k following α is a *handle of* $\alpha\beta w$. We write $\langle A \to \beta, k\rangle$ for such a handle.

Remember the schematic picture of a parser machine, from Figure 4.3. Note, the word w corresponds to the future input still to be parsed, so the symbols to the right-hand side of a handle contain only terminals. The part $\alpha\beta$ will correspond to the **stack content**, where β is the part of the stack touched by reduction step. Note also that the "handle"-part β can be *empty*, i.e. β can be ϵ. As said, right-most derivation steps $\Rightarrow_r$ correspond to reduction steps *in reverse* (for which we here write $\hookrightarrow$). Those steps corresponds also to one of two possible kinds of steps of of the parser stack machine. Remember that LR-parsing is also called *shift-reduce* parsing, and $\hookrightarrow$ corresponds, of course, to a **reduce**-step. That form of steps replaces β on the stack by A[4] In the parse-tree, A is a parent node to the children that correspond to β. This, the reduce step is the code **bottom-up** movement of the parser.

What then about the other kinds of steps of a bottom-up parser, the shift-steps? They eat the next input letter, a terminal, and **push** it on the stack. A reduce step using a grammar production $A \to \beta$ is possible if the current stack contains β on top. Of course, there may be different rules with the same right-hand side that match in this way, and it may also be that the is another rule $A' \to \beta'\beta$. If the top of the stack contains $\alpha'\beta$, then both grammar rules can be used for a reduce step. If that occurs it's called **reduce-reduce** conflict. A shift-step may seem always possible, as long as there is still input. A situation where a shift and a reduce is possible at the same time will be called **shift-reduce**-conflict.

The discussion about conflicts so far is a bit simplified and neglected that the parser machine has also *states*; it's a stack machine, i.e., an automaton with finitely many states *plus* a stack. If a reduce steps is possible does *not* just depend on the top of the stack, but also on which state the machine is in. Same for shift-steps. Otherwise, as hinted at, a shift step would always be an option and any possible reduce step would be in conflict with the unconditionally possible shift step.

The trick will be: how to construct the states of the parser machine!

[4] Saying that the step replaces stack content sounds a bit silly if β is ϵ, . In that case, A appears or is pushed on the stack, without actually replacing any symbol.

The construction will be for later, we continue for the time being generally describing the behavior of the parser machine, without concretely showing the individual states or how they can be constructed.

Example 4.7.4 (Run of a bottom-up parser). Let's revisit the artificial grammar from Example 4.7.1. Table 4.12 contains schematically the steps of a shift-reduce parser for the parse-tree of Figure 4.16.

The table contains two columns, the one of the left represents the stack-content, the one on the right the remaining input. We assume **$** as the end of the input, i.e., the end of the token stream. The same symbol is also used for representing the stack content, representing the "bottom" of the stack.

$\mathbf{\$}$	$\mathbf{t}_1\mathbf{t}_2\mathbf{t}_3\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,\mathbf{t}_1$	$\mathbf{t}_2\mathbf{t}_3\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,\mathbf{t}_1\mathbf{t}_2$	$\mathbf{t}_3\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,\mathbf{t}_1\mathbf{t}_2\mathbf{t}_3$	$\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,\mathbf{t}_1 B$	$\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,A$	$\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,A\mathbf{t}_4$	$\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,A\mathbf{t}_4\mathbf{t}_5$	$\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,AA$	$\mathbf{t}_6\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,AA\mathbf{t}_6$	$\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,AB$	$\mathbf{t}_7\,\mathbf{\$}$
$\mathbf{\$}\,AB\mathbf{t}_7$	$\mathbf{\$}$
$\mathbf{\$}\,S$	$\mathbf{\$}$
$\mathbf{\$}\,S'$	$\mathbf{\$}$

Table 4.12: Schematic run (reduction from top to bottom)

The parser starts with an empty stack, i.e., the stack consists of only **$**, and the full input $\mathbf{t}_1 \ldots \mathbf{t}_7$, terminated by **$**, still ahead. In each step, either the first symbol of the input pushed to the stack, shortening the remaining input by one: the token is *shifted* from the input to the stack. The *reduce* steps don't shorten the remaining input, but they replace the top of the stack according to one production. For instance, in the 3rd step, the stack content is $\mathbf{\$}\,\mathbf{t}_1\mathbf{t}_2\mathbf{t}_3$. Applying production $B \to \mathbf{t}_2\mathbf{t}_3$ results in the stack content of $\mathbf{\$}\,\mathbf{t}_1 B$ after that reduce-step. □

Apart from the fact that it ignored the **states** of the parser machine, the example should have given a good general impression what shifting and reducing means, and how the **configuration** of such a parser evolves during a parse. To summarize the observations: A configuration consists of 3 ingredients.

1) A **stack,** containing terminals and non-terminals (and **$** at the bottom. It represents what has been read already but not yet fully "processed". 2) the word of terminals **not yet read**. And 3), the **state** of the machine (not shown in the example).

Remark 4.7.5 (Rest of the input and position of the reading head)**.** The terminals not yet read can more generally interpreted as position of the reading head on the input. The parser works strictly one-directional, **left-to-right**. That's the "L" in LR. A more general machine model might be able to move the head in both directions, visiting again symbols already read. For parsing (and standard stack-automata and in particular for shift-reduce parsing), that's not done (and "rewinding" on the input is not seen useful for parsing (nor lexing, of course). In other words, once read, input is considered consumed and processed, and relevant for the configuration is only the remaining input. □

As mentioned, the state of the parse was not illustrated in the previous example. Indeed, we continue to leave that out also the following schematic illustration. That the machine is in particular states during the run is central to make correct decisions wether to do a shift step or a reduce step, resp. which reduce step to take. The states will also be part of the **parser table** (remember Figure 4.15)

Since for the moment, we explain aspects of LR-parsing *without* referring to the state of the parser-engine, we assume in the following, that the parse tree is somehow already known and show then runs where the machines simply does the right decisions that fit to the intended parse-tree. We did a similar assumption earlier, discussion **oracular** derivations. Obviously the trick ultimately will be: how do achieve the same *without that tree already given*, just parsing left-to-right. So far, we also won't mention of look-ahead, i.e., the decision making also is helped by looking at the next token(s), also that will play a role in the concrete construction.

To sum up the two kinds of steps of an LR-parser:

Shift: move the next input symbol (terminal) over to the top of the stack ("push").

Reduce: , more concretely, reduce using currently appropriate grammar producion $A \to \beta$: remove the symbols of the **left-most** subtree (corresponding to β from the stack and **replace** it by the non-terminal at the root of the subtree (corresponding to A)

The replacement of the top of the stack β by A, moving up the tree, can also be interpreted as a combination of "pop + push": popping β followed by pushing A.

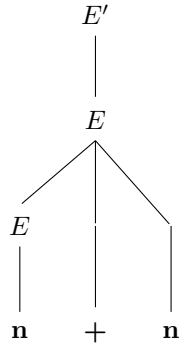
The remark that it's "easy to do" refers to what is illustrated next: the question namely how the parser makes its decisions: Should it do a shift or a reduce and if the latter, reduce with what rule. If one assumes the "target" parse-tree as already given (as we currently do in our presentation, then the parse tree embodies those decisions: it's the result of those decisions. In particular, a parent-childen situation, where the parser progresses upwards, corresponds to a *reduce* step.

Of course, the tree is *not given* a priori, it's the parser's task to build the tree (at least implicitly) by making those decisions about what the next step is (shift or reduce).

Example 4.7.6 (LR parse for "+", given the tree). Consider the following grammar:

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array} \tag{4.24}$$

Note that the very simply expression grammer is left-recursive and that there are no ambiguity issues with associativity and precedence: there is only addition as operator and the grammar enforces left-associativity.



(a) Parse tree

	parse stack	input	action
1	$	n + n $	shift
2	$ n	+ n $	red:. $E \to \mathbf{n}$
3	$ E	+ n $	shift
4	$ E +	n $	shift
5	$ E + n	$	red. $E \to E + \mathbf{n}$
6	$ E	$	red.: $E' \to E$
7	$ E'	$	accept

(b) run resp. reduction

Figure 4.19

Figure 4.19 shows a parse tree for a simple expression like $1 + 2$ and a corresponding run of a shift-reduce parser. The run in Figure 4.19b in reverse, i.e., from the last to the first line, corresponds to the derivation:

$$\underline{E'} \Rightarrow \underline{E} \Rightarrow \underline{E} + \mathbf{n} \Rightarrow \mathbf{n} + \mathbf{n} \ . \tag{4.25}$$

The grammar is so simple that sentential forms contain only one non-terminal or none. So the derivation from equation (4.25) is a right-most derivation, it just not so visible, as it's also a left-most derivation, resp. the only possible derivation anyway.[5]

The message of the example is, however, not the connection between reduction and derivations, but to shed light on how the machine can make decisions assuming that the tree is already given.

For that, one should compare the situation in stage 3 and state 6. In both situations, the machine has **the same stack content**, containing only the end-marker and E on top of the stack. At stage 3, the machine does a shift, whereas in stage 6, it does a reduce. In the tree, the two situations correspond to the to E-nodes in the tree. In the first case, the E cannot immediately be treated in isolation. After shifting the two terminals to the stack, the stack-content is $E + \mathbf{n}$. That corresponds to the right-hand side of production 2, and

[5] The grammar is an example of a *left-linear* grammar. Left-linear context free grammars can only express regular languages. Same for right-linear ones.

at that point, a reduction step can be done that corresponds to the tree structure. In the situation corresponding to line 6, with the other E-node, doing the reduction according to the first production.

The stack content (representing the "past" of the parse, i.e., the already processed input) is the identical in both cases. We will see that in more detail later how it concretely works but a few general remarks about the stack content. As said, it represents the past of the current parser run, i.e., what it has handled so far. It will play the role of the memory of the parser. Actually, it's not a perfect memory.

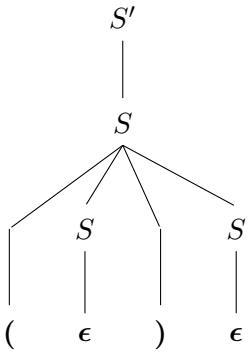
The parser machine is necessarily *in the same state* in both stages, which means, it cannot be the state that makes the difference. What then? In the example, the form of the parse tree shows what the parser should do. But of course the tree is not available. Instead (and not surprisingly), if the past input alone cannot be used to make the distinction, one also takes the "future" input into accounnt. Maybe not all of it, but part of it. That's of course the concept of **look-ahead**, we have encountered also for top-down parsing. Look-head will *not* yet be done for LR(0), as that form is without look-ahead. □

Let's have a look at another example, this one with ϵ-transitions.

Example 4.7.7 (Example with ϵ-transitions: parentheses). Let's have a look at the following grammar for parentheses:

$$\begin{array}{lcl} S' & \to & S \\ S & \to & (\,S\,)\,S \;\mid\; \epsilon \end{array} \tag{4.26}$$

As a side remark: unlike the previous grammar, the grammar here has productions with *two* non-terminals on the right. Consequently, there is now a difference between left-most and right-most derivations (and mixed ones).



(a) parse tree

	parse stack	input	action
1	\$	( ) \$	shift
2	\$ (	) \$	reduce $S \to \epsilon$
3	\$ (S	) \$	shift
4	\$ (S)	\$	reduce $S \to \epsilon$
5	\$ (S) S	\$	reduce $S \to (\,S\,)\,S$
6	\$ S	\$	reduce $S' \to S$
7	\$ S'	\$	accept

(b) Run resp. reduction

Figure 4.20: parentheses

Note in particular the 2 reduce steps for the ϵ production. Those steps simply add or push the non-terminal S to the stack. The right-most derivation and right-sentential forms that correspond to the reduce-steps in reverse are:

$$\underline{S'} \Rightarrow_r \underline{S} \Rightarrow_r (\,S\,)\,\underline{S} \Rightarrow_r (\,\underline{S}\,) \Rightarrow_r (\;) \;. \tag{4.27}$$

□

After the examples, let's have a further look into the form of the reductions and the corresponding derivations and the configurations of the stack machines. Sentential forms, as introduced earlier are words from Σ^* derivable from start-symbol. For LR-parsing, we are in particular dealing with *right-sentential* forms, i.e., those derivable by a right-most derivation:

$$S \Rightarrow_r^* \alpha$$

Let's revisit the addition Example 4.7.6 again.

Example 4.7.8 (Right-sentential forms and parser configurations). Figure 4.19b showed the run and the configurations of shift-reduce parser. For clarity, we show here the right-most derivation (on the left) and the correspond reduction sequence (on the right).

$$\underline{E'} \Rightarrow_r \underline{E} \Rightarrow_r \underline{E + \mathbf{n}} \Rightarrow_r \mathbf{n} + \mathbf{n} \quad (4.28) \qquad \underline{\mathbf{n}} + \mathbf{n} \hookrightarrow \underline{E + \mathbf{n}} \hookrightarrow \underline{E} \hookrightarrow E' \quad (4.29)$$

All the words are right-sentential forms. Looking at configurations from Figure 4.19b, we see that the forms show up in the configurations as part of the run, but **split** between *stack* on the left and *input* on the right.

The derivation from (4.28) (which is repeproduced here from equation (4.25) "contains" only the reduce-steps from the run of Figure 4.19b, the shift-steps (in reverse) are not part of the derivation. Also, only the reduce-step correspond to growing the parse-tree.

For the discussion here, let's introduce some ad-hoc notation to illustrate the separation between the parse stack on the left and the future input on the right. With **.** for this notation and $\sim_s$ to indicate a shift step that does not grow the parse tree, we can write the derivation sequence from equation (4.28) more detailed as follows:

$$\underline{E'}\mathbf{.} \Rightarrow_r \underline{E}\mathbf{.} \Rightarrow_r \underline{E} + \mathbf{n}. \sim_s \underline{E} + \mathbf{.n} \sim_s \underline{E}\mathbf{.} + \mathbf{n} \Rightarrow_r \mathbf{n}\mathbf{.} + \mathbf{n} \sim_s \mathbf{.n} + \mathbf{n} \quad (4.30)$$

□

Figure 4.21 shows conceptually the status of a parser, halfway through the parse. We still assume that the parse tree is given, it's shown in the picture. So the left-lower part of the overall parse tree has already been processed by the parser, i.e., the first tokens have been eaten already by a number of shift steps. But processing those triggered already some reduce-steps as well, which means ...

So far we clarified the general working of a shift-reduce parser, its stack and how it grows the parse tree in a bottom-up fashion, but omitted the role of the **states.** States are important for the decision-making of the maching, i.e., to decide whether to shift, or to reduce (and perhaps reduce according to which producion, of more than one is possible. The decision-making was brushed under the rug by assuming in the examples that the parse tree is somehow known, and the parser just the appropriate step to obtain that tree.

Without having that tree at hand: which criteria can the parser base its decision on for the next step?

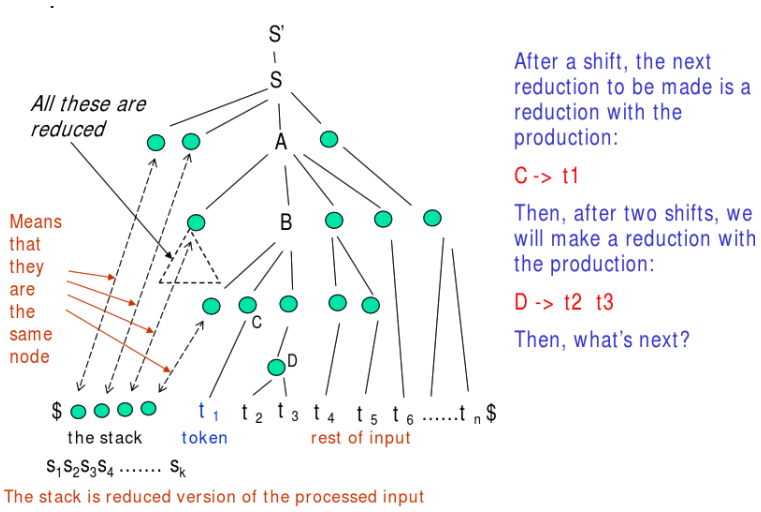


Figure 4.21: A typical situation during LR-parsing

1. the top of the **stack** (handle)
2. the **look-ahead** on the input
3. the **state**

As far as the look-ahead is concerned: the first construction we concretely present will be that for LR(0) parsers in Section 4.7.3. Those won't even have a look-ahead. But what are the states of an LR-parser?

General idea (change the slide):
Construct an NFA (and ultimately DFA) which works on the **stack** (not the input). The alphabet consists of terminals and non-terminals $\Sigma_T \cup \Sigma_N$. The language

$$Stacks(G) = \{\alpha \mid \begin{array}{l}\alpha \text{ may occur on the stack during} \\ \text{LR-parsing of a sentence in } \mathcal{L}(G)\end{array}\}$$

is **regular**!

Note that this is (or at least seems) a *restriction* of what one can do with a stack-machine (or push-down automaton) can do. As mentioned, exploiting the full-power of context-free grammars is impractical, already for the fact that one does not want ambiguity (and non-determinism and backtracking). One further general restriction is that one wants a bounded look-head, maybe a look-ahead of one. The restriction here is a kind of strange one, insofar it does not all the stack content to be of arbitrary shape, but all allowed stack contents (for one grammar) must be regular.

On the other hand, the restriction is also kind of natural. Any push-down automaton consists of a stack and a finite-state automaton. It's a natural general restriction, that the automaton is *deterministic*: given a particular input *determines* the state the machine is in. Realizing that the stack-content is an "abstract representation" of the past, it's

natural that the finite-state automaton is also *deterministic wrt. that abstract past*. Or to say it differently: the parser machine has in some way an unbounded memory, the stack. The memory is insfor restricted, in that it can be used not via random access, but only via a stack discipline with push and pop (that inhererent to the notion of context-free grammars). Having an infinite memory is fine, one can in principle remember everything (using only push and without ever forgetting anything by using pop). But the machine has to make also *decisions* based on the past. So for that decision-making part, it cannot make infinitely many different decisions, based on ininitely many pasts. Relevant are are only finitely many different pasts. This is the abstraction built into the stack-memory: doing a push followed by a pop does not change the stack. So both situations have the same stack content, so a past with a history of push and pop is treated the same as if nothing had happend at all. So, it's natural to connect the state of the machine on which the decision is made on the stack content.

4.7.3 LR(0) parsing as easy pre-stage

Now, let's get real and show a parser construction including the states for real, the one for LR(0). In practice, those parsers are *too simple* and not expressive enough for realistic language, but construction is an easy step towards the variants wich take more look-ahead into account, like LR(1), SLR(1) etc. Actually, already LR(1) is good enough in practice, so that LR(k) is not used for $k > 1$. Indeed the most well-known class of parser generators, yacc and friends, is not even LR(1), but the weaker class LALR(1).

Even if not expressive enough, the contruction of LR(0) parsers underlies directly or at least conceptually the more complex parser constructions to come. In particular: for LR(0) parsing, the core of the construction is the so-called **LR(0)-DFA,** based on **LR(0)-items.** This construction is directly also used for SLR-parsing. For LR(1) and LALR(1), the construction of the corresponding DFA is not identical, but analogous to the construction of LR(0)-DFA.

An LR(0)-item is a production with specific "parser position" marker **.** in its right-hand side. The "**.**" is not part of the production.

Definition 4.7.9 (LR(0) item)**.** An **LR(0) item** for a production $A \to \beta\gamma$ is of the form

$$A \to \beta.\gamma$$

Items with dot at the beginning are called **initial** items, those with with the dot at the end are called **complete** items. □

Being part of a right-hand side of a production, γ and γ are words containing terminals and non-terminals and can be empty words, including the case, where both are empty (for an ϵ-production).

Let's use some of the previous grammars for illustration. They should make the concept of items clear enough. The only point to keep in mind is the treatment of the ϵ symbol.

Example 4.7.10 (Items). For the grammars from Example 4.7.6 (parentheses) resp. from Example 4.7.7 (addition), the lists of items look as follows:

$$
\begin{array}{lcl}
E' & \to & \mathbf{.}E \\
E' & \to & E\mathbf{.} \\
E & \to & \mathbf{.}E + \mathbf{number} \\
E & \to & E\mathbf{.} + \mathbf{number} \\
E & \to & E + \mathbf{.number} \\
E & \to & E + \mathbf{number.} \\
E & \to & \mathbf{.number} \\
E & \to & \mathbf{number.}
\end{array}
\qquad
\begin{array}{lcl}
S' & \to & \mathbf{.}S \\
S' & \to & S\mathbf{.} \\
S & \to & \mathbf{.}(\,S\,)\,S \\
S & \to & (\,\mathbf{.}S\,)\,S \\
S & \to & (\,S\mathbf{.}\,)\,S \\
S & \to & (\,S\,)\,\mathbf{.}S \\
S & \to & (\,S\,)\,S\mathbf{.} \\
S & \to & \mathbf{.}
\end{array}
$$

Both grammars lead (coincidentally) to altogether 8 items. Note that the production $S \to \epsilon$ from the second example gives $S \to \mathbf{.}$ as item, and not two items $S \to \epsilon\mathbf{.}$ and $S \to \mathbf{.}\epsilon$). As a side remark for later: it will turn out, both grammars are *not LR(0)*. □

The LR(0)-DFA now uses items as states, resp. sets of items as states. As said, the DFA does not operates on the input, **but on the stack**. As alphabet therefore it uses terminal as well as non-terminal symbols.

To construct the DFA, one can go two routes, either one does a NFA, which is afterwards made deterministic, or one constructs the DFA directly. We show both ways.

LR(0)-NFA

Let's do the NFA-construction first, i.e., we construct an automaton whose states are single *LR(0)*-items; the states of the corresponding DFA will be sets of items. So, states are of the form

$$A \to \beta\mathbf{.}\gamma$$

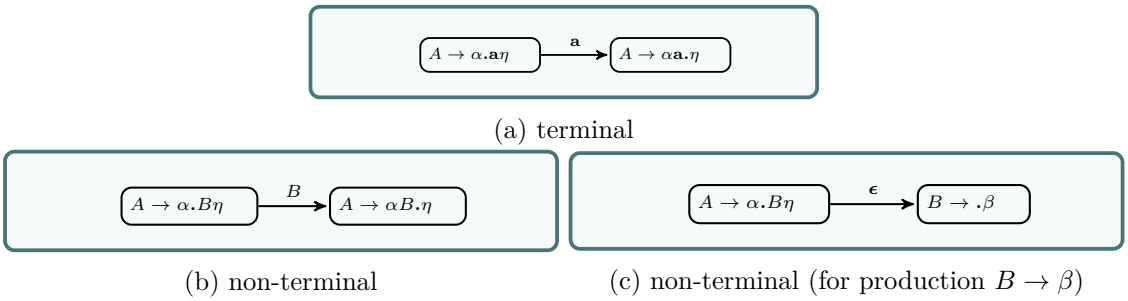
where $A \to \beta\gamma$ is a production of the given grammar. The marker **.** is meant to represent split between the current stack content and the expected future input. More precisely, the β is supposed to be the current top of the stack, and γ is to be treated next. That can be compared with the derivation from equation (4.30), where we used a dot to make that split visible.[6]

Note that both β and γ can contain terminals and *non-terminals*. The latter point require some explanation, since of course the input consists of terminal symbols, the tokens from the scanner. We come back to this point after introducing the transitions of the NFA, and we see how it works also in the examples later.

With the states of the NFA fixed, next we define its transitions. The **transitions** between the "item-states" are shown in Figure 4.22.

The distinction is based on the position of the **.**-marker, i.e., whether it's in front of a terminal symbol or a non-terminal. Let's start with transitions for **terminals** from

[6] There, the notation was meant informally, it's not a derivation consisting of items, though it's connected.



(a) terminal

(b) non-terminal

(c) non-terminal (for production $B \to \beta$)

Figure 4.22: Transitions of the LR(0)-NFA

Figure 4.22a. The marker is right in fron of a terminal symbol. Remenber that the marker is intended to represent the position of the parser-machine. So in the start-state of that transition, the reading had is in front of a terminal symbol, which is consumed and it the state afterwards, the symbol **a** is on the left of the **.**-marker, i.e., the transition corresponds to **shift-**step:

> A parser in state $A \to \alpha.\mathbf{a}\eta$ has α on the top-most part of it's stack. It can do a shift-step, pushing **a** to its stack, and move to state $A \to \alpha\mathbf{a}.\eta$

The non-terminal A, mentioned in the states, is not (yet) on the stack. Doing steps, at least those corresponding to the transitions from Figure 4.22a and 4.22b, moves the **.** step by step further to the left, until it reaches the end, i.e., until the machine is in the state $A \to \alpha\mathbf{a}\eta.$ with the marker at the end. Btw: that's a *complete* item. At that point, the whole right-hand side of the considered production for A has been pushed on top of the stack. In other words, at that point, a **reduce**-step is possible for production $A \to \alpha\mathbf{a}\eta$. That step replaces on the stack the production's right-hand side $\alpha\mathbf{a}\eta$ by its left-hand side A.

That brings us the the treatment of non-terminals, covered by Figures 4.22b and 4.22c. We have also to answer the slight puzzle what it means that the **.**-marker in an item stands in front of a non-terminal. In first approximation, we mentioned, that intuitively means the parser head right in front of the symbol mentioned after the marker. That's a proper intuitition for terminals, and then the transition corresponds to a shift step.

Of course, the intuition is slightly misleading in that on the actual input, the token stream, the are only terminals. We should not forget: the NFA does **not actually operate on the input**, like a scanner-automaton would to! It describes the allowed stack-contents and the stack-content, (with the help of the NFA, resp. ultimately the DFA), determines the state of the parser. So in case of a shift-step, the parser puts the symbol to the stack, and the NFA (or DFA) then moves to a (potentially) different state reflecting the change in stack content. For the shift-steps, that actually make the NFA behave like scanner would do, because in a FSA-state with, say outgoing transitions labelled ***b*** and ***c***, the parser would accept either of the two terminals as legal one-step continuation of the parse, doing a shift-step step, but when presented with a different letter ***d***, it would reject it and stop the parse.

What about the non-terminals? That might be clearer when looking at the ϵ transition in Figure 4.22c and the role it plays in connection with *reduce*-moves of the parser.

The step can be done when the **.** stands in front of a non-terminal say B. Let's make it a bit more concrete than the general transitions for non-terminals in Figures 4.22b and 4.22c, and assume a item state of the form

$$A \to \alpha \mathbf{.} B\, \boldsymbol{a} \tag{4.31}$$

i.e., the non-terminal in question is immediately followed by one terminal $\boldsymbol{a}$. Being in this item-state is currently means for the NFA: on the lower part of the stack there is α, and on top of the stack, when continuing "scanning" the stack successfully following the transitions, it would do the following steps:

$$A \to \alpha \mathbf{.} B\, \boldsymbol{a} \quad \xrightarrow{B} \quad A \to \alpha B \mathbf{.}\, \boldsymbol{a} \quad \xrightarrow{\boldsymbol{a}} \quad A \to \alpha B\, \boldsymbol{a} \mathbf{.} \tag{4.32}$$

The B-step in the middle has no effect on the real input, it's just the internal book-keeping over the stack content, but the subsequent $\boldsymbol{a}$-transition corresponds to shift step. I.e. the behavior of equation (4.32) states that the `next token` on the input is $\boldsymbol{a}$. That's not the only possible behavior starting from the state of equation (4.31). Alternatively it could start with a $\boldsymbol{\epsilon}$-transition according to Figure 4.22c:

$$A \to \alpha \mathbf{.} B\, \boldsymbol{a} \quad \xrightarrow{\epsilon} \quad B \to \mathbf{.} \beta \tag{4.33}$$

Concerning the stack that means, the top of the stack α is (assumed to be) replaced by β. That of course describes a **reduce-step** of the parser-machine.Being in the item-state (4.31) means α is actually on the stack, with the **.** after α. In the **initial state** $B \to \mathbf{.}\beta$ afterward, the **.** is in frontI say "assumed"

ms Error: reduce

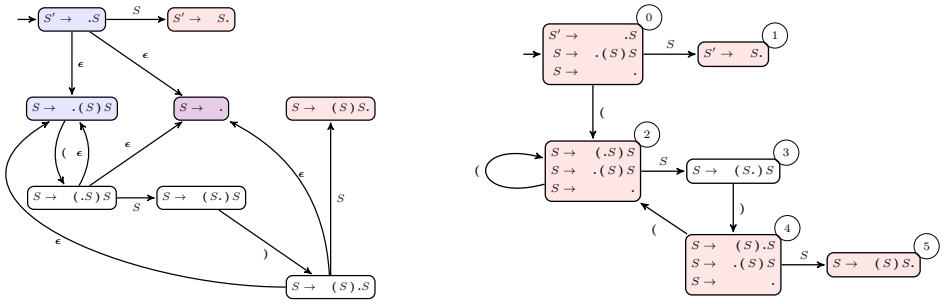
ms Error: Is it really a reduce step? I thought reduce step is about complete items.

and moving to the item $\mathbf{.}B$ places the **.** in front of the right-hand sides B of any production for the non-terminal B. Note that item-states $A \to \alpha \mathbf{.} B\eta$ have at least kinds of outgoing transitions, one according Figure 4.22b and an ϵ-transition according to Figure 4.22c, according to each production of B. (4.31)

Example 4.7.11 (LR(0)-NFA for parentheses grammar). Let's do the construction for the parethesis from Example 4.7.7. The corresponding LR(0)-NFA is shown in Figure 4.23a. We also show the DFA in Figure 4.23b, though we are currently just discussing the NFA.

In the figure, we use colors for illustration, only, they are not officially part of the construction. We use "reddish" for complete items, "blueish" for initial items, and "violet'ish" for states that are both initial and complete.

Furthermore, you may notice: there is one initial item state per production of the grammar. The initial items is where the $\boldsymbol{\epsilon}$-transisitions go into (as defined in Figure 4.22c), but *with exception* of the initial state (with S'-production). And there are not no outgoing edges from the complete items. Note the uniformity of the $\boldsymbol{\epsilon}$-transitions in the following sense.



(a) NFA: parentheses

(b) DFA: parentheses

Figure 4.23: Parentheses

For each production with a given non-terminal (for instance S in the given example), there is one ingoing ϵ-transition from each state/item where the **.** is in front of said non-terminal.

□

Side remark 4.7.12 (Nitpicking)**.** The nitpicking mind may comment: the automaton from Figure 4.23a has no accepting states, the automaton accepts not words, so its language may equivalently represented by the empty automaton.

Well, the machine that accepts input is not the NFA (or DFA), but the parse machine, the stack automaton. The NFA here is a vehicle to construct and define the states of the stack automaton, "working" on the stack content. It's best to imagine that *all* states in the NFA (or DFA later) are *accepting*. Often that is left out, as more important in the overall pictue are the concepts of initial-items and complete-item states, and the stack-parser maching will have a specific acceptance condition. □

To look forward, and concerning the role of the ϵ-transitions. Those are allowed for *non-determistic* automata, but not for DFAs. The underlying construction (discussed later) is building the ϵ-closure, in this case the close of $A' \to A$. If one does that directly, one obtains directly a DFA (as opposed to first do an NFA to make deterministic in a second phase).

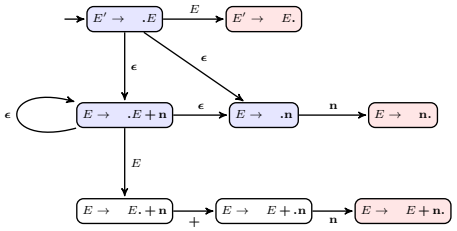
We made our lives *easier*: assume one *extra* start symbol say S' (augmented grammar)

> The **initial** state of the NFA (and the parser machine) is the initial item $S' \to .S$ as (only) of the

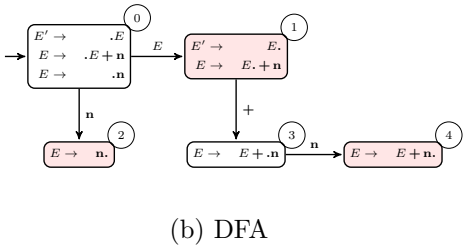
The acceptance condition of the *overall* machine: a bit more complex

> The input must be empty, stack must be empty except the (new) start symbol **$**, the NFA has a word to say about acceptance, but *not* in form of being in an accepting state[a], but and accepting **action**, see later.
>
> [a] as mentioned one may see all NFA-states as accepting.

Example 4.7.13 (LR(0)-NFA for the addition grammar). Let's show the construction also for the addition grammar from Example 4.7.6. Figure 4.24a shows the corresponding LR(0)-NFA (and for comparizon, we show als a deterministic machine in Figure 4.24b. □



(a) NFA



(b) DFA

Figure 4.24: Addition

The parser stack machine will not work with non-deterministic automaton, but with a deterministic version. In that case, the stack content (if acceptable to the automaton) *determines* the state of the automaton (and the parser machine). In the examples with the LR(0)-NFAs, we also showed the corresponding DFAs.[7]. We have covered the way to turn an NFA to a DFA earlier in the context of scanning, the so called subset-construction. For the deterministic version, states then contain *sets* of items, and an important concept was also the so-called ϵ-closure.

Next, we show a *direct* construction of the LR(0)-DFA, (without the detour over NFAs).

Direct construction of an LR(0)-DFA

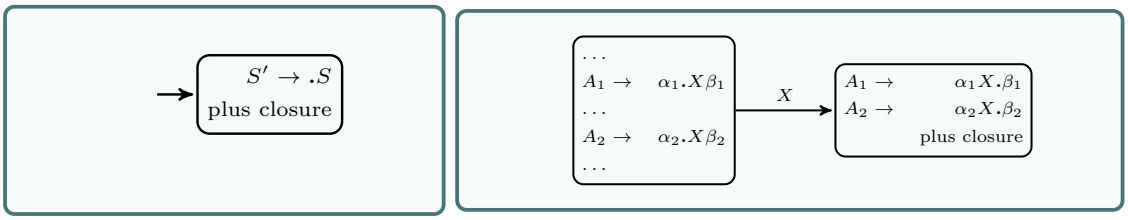
The direct construction of a deterministic LR(0) automaton is actually not much harder that the construction of the NFA. For the constructed NFAs, there are no situations like $q_0 \xrightarrow{\boldsymbol{a}} q_1$ and $q_1 \xrightarrow{\boldsymbol{a}} q_2$ with $q_1 \neq q_2$, actually not even $q_1 \xrightarrow{\boldsymbol{a_1}}$ and $q_1 \xrightarrow{\boldsymbol{a_2}}$ (with $\boldsymbol{a_1} \neq \boldsymbol{a_2}$, and the same for transitions involving non-terminals. The *only* symptom of non-determinism are the **ϵ-transitions,** and so the key for the direct construction is build-in the ϵ-closure directly

> ϵ-closure: if $A \to \alpha.B\gamma$ is an item in a state, and the grammar has productions $B \to \beta_1 \mid \beta_2 \ldots$. Then **add** items $B \to .\beta_1$, $B \to .\beta_2 \ldots$ to the state. Continue that process, **until saturation**.

The initial state and the transitions are shown in Figures 4.25a and 4.25b. The symbol X represents a terminal or a non-terminal. Both are treated uniformely.

For the transition in Figure 4.25b, the picture is meant to indicate that *all* items of the form $A \to \alpha.X\beta$ must be included in the post-state and all others (indicated by "…") in the pre-state, are not included in the post-state (unless they happen to be added by in the closure).

[7] Technically, we don't require here a *total* transition function, we left out the error state.



(a) initial state (b) transitions

Figure 4.25: LR(0)-DFA

One can re-check the previous examples first doing the NFA, then the DFA), in particular Figures 4.23b and 4.24b: the outcome is the same.

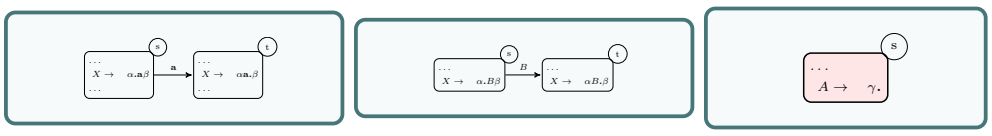
The LR(0) parser stack automaton

No we have seen the bottom-up parse tree generation, the general working of shift- and reduce-steps, we have seen the stack and the input as part of the parser's configuration, and finally the construction of the LR(0)-DFA.

We need then to interpret the "set-of-item-states" in the light of the stack content and define **reaction** in terms of transitions in the *stack automaton*, stack manipulations and shift and reduce step, it's acceptance condition. Since we are doing parsing without look-head in LR(0), the content input is not taken into accound when deciding the reaction. If the decision is to do a shift step, the next letter is shifted to the stack independent of what letter it is. However, different letters may lead to different successor states.

Let's start with the LR(0)-automaton. As said, it "operators" on the stack, which contains words over Σ^*. The DFA operates, deterministically on such words, when feeding the stack content into the automaton, starting with the oldest symbol (not in a LIFO manner) and starting with the DFA's initial state. In other words, the stack content **determines** state of the DFA. Actually: each prefix of the stack also determines uniquely a state and we call **top state**, the state after scanning the complete stack content. That's the state which at the same time is taken as state current **state of the LR(0) parser machine** and is crucial when determining *reaction* of that machine.

Figure 4.26 shows three different situations allowing three different reactions. In all three figures, the "source" state labelle s, which also corresponds to what we called *top-state*, i.e. the state of the DFA after parsing the current stack.



(a) shift (b) "goto" (c) Reduce

Figure 4.26: Parser machine transitions

The state of Figure 4.45a contains an item $X \to \alpha.\boldsymbol{a}\,\beta$ and that's a state allowing a **shift.** After eating $\boldsymbol{a}$ and pushing it to the stack, the machine is in (top-)-state t. State

transitions for non-terminals work similar. The source-state in Figure 4.26b contains an item $X \to \alpha.B\beta$. In that step, the input is left unchanged. Those transitions are also called **goto**-steps, and they will in particular listed as goto-steps in the LR-parse tables later.

Figure 4.26c shows an item that contains a **complete**-item. That is characteristic for **reduce**-steps. There is no transition shown in the figure, since reduce steps are not reflected as transitions in the LR(0)-automaton. It's of course a transition in the parser-machine, namely a transition that replaces the right hand side of a production by the non-terminal on the production's left-hand side.

So in the situation of Figure 4.26c, with a **top-state** as shown, the stack contains a full right-hand side ("handle") γ for the non-terminal A on the stack. A reduce step using that production replaces A by A on the stack. With A on top of the stack instead of γ, there will in general a **new top state**!

So, sommarize: if a reduce happens, the parser engine *changes state*![8] This state change, however, is **not** represented by a transition in the DFA (or NFA for that matter). In particular, the reduce step is *not* represented ly outgoing arrows of completed items, though having reached a state containing a completed item indicates that a reduce step can be done.

Example 4.7.14 (Example: LR parsing for addition). Let's revisit Example 4.7.6, which illustrated shift-reduce parsing assuming the parse-tree given. This was done for the addition-expressions from the grammar from equation (4.24). The parser run was shown in Figure 4.19b.

In the meantime, we have constructed the LR(0)-DFA (see Figure 4.24b), so we can "re-run" the example with the states from that DFA.

Previously, at each point the decision was clear, with the tree at hand. Of course, the LR(0)-parser does not have the tree available, the only thing it has is "the past" which is represented (partially) by the stack content. As discussed for Figure 4.19b, interesting in the run are stage 3 and state 6, which have the **same stack content**. That also means, the parser is in **the same state** of its LR(0)-DFA. In the automaton from Figure 4.24b that's state 1.

The state 1 is important, as it illustrates a **shift/reduce conflict.** Remember: reduce-steps are *not* represented in the LR(0)-automaton via *transitions*. They are only implicitly represented by *complete items*. A shift-reduce conflict is not characterized by 2 outgoing edges. A LR(0)-shift-reduce conflict is an **outgoing edge from a state containing a complete item**. □

We know that a parser can make decisions based on a look-head, but that is not yet done: the LR(0)-DFA, in state 1 especially, can do a reduce step or a shift step, which constitutes the conflict. In other words: the addition grammar is *not LR(0).*

Later, we will see under which circumstances, looking at the "next symbol" can help to make the decision. That leads to SLR parsing or later to LR(1)/LALR(1). In the

[8] In general, there is a change. There may be situations, where the the parse moves from one state to the same state again.

particular situation of state 1 in the example, the next possible symbol would be + or else **$**.

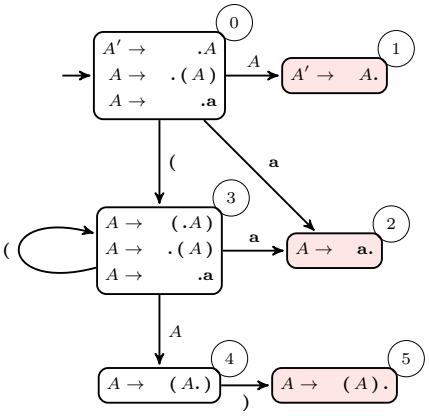
The point being made when lookig at that 1 is the following: the state is a complete state (a state containing a complete item). Besides that, there is an outgoing edge. That means, in that state, there are two reactions possible: a shift (following the edge) and a reduce, as indicated by the complete item. That indicates a conflict-situation, especially if we don't make use of look-aheads, as we do currently, when discussing LR(0). The conflict-situation is called, not surprisingly, a "shift-reduce-conflict", more precisely an LR(0)-shift/reduce conflict. The qualification LR(0) is necessary, as sometimes, a more close look at the situation and taking a look-ahead into account may defuse the conflict. Those more fine-grainend considerations will lead to extensions of the plain LR(0)-parsing (like SLR(0), or LR(1) and LALR(1)).

Let's have a look at another example, one involving parentheses. We have earlier, in Example 4.7.7 looked at a grammar for parentheses. The next example captures a more simplistic use of parentheses.

Example 4.7.15 (Simple parentheses). Consider the following grammar:

$$A \;\rightarrow\; (\,A\,) \mid \mathbf{a} \tag{4.34}$$

Figure 4.27a shows the LR(0)-DFA for the grammar, and Figure 4.27b possible reactions for each each (top)-state.



(a) LR(0)-DFA

state	possible action
0	only shift
1	only red: $(A' \rightarrow A)$
2	only red: $(A \rightarrow \mathbf{a})$
3	only shift
4	only shift
5	only red $(A \rightarrow (\,A\,))$

(b) Run

Figure 4.27: Simple parentheses

For completeness sake, Figure 4.28 shows also the NFA for the example.

□

As for scanning automata, the reactions of the machine can also be represented in tabular form. Here, the table is called **LR(0) parsing table**. There later will be variant for SLR(1), LALR(1), and LR(1) (see later), which are slightly different and incorporate more information in connection with the look-ahead of course. Let's simply look at how the tables look for the examples just presented.

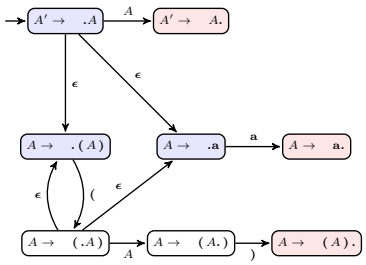


Figure 4.28: NFA for simple parentheses (bonus)

Example 4.7.16 (LR(0)-table for simple parentheses). The syntax for the simple parentheses have been defined in Example 4.7.15. The parse table from Table 4.13 is nothing else than the LR(0)-DFA from Figure 4.27a in tabular form.

state	action	rule	input			goto
			(	**a**	)	A
0	shift		3	2		1
1	reduce	$A' \to A$				
2	reduce	$A \to \mathbf{a}$				
3	shift		3	2		4
4	shift				5	
5	reduce	$A \to (\, A \,)$				

Table 4.13: LR(0) parsing table

The shift-steps are listing under the "input"-columns. The reaction on non-terminal(s), here only A, is listed in the "goto"-column(s), here only one. Note there is no goto-transition for the extra starting state A'. It's a consequence that we agreed that the start state should never occur on the right-hand side of a production.

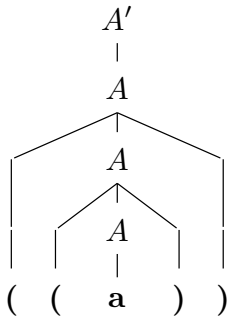
The table contains a definite reaction per state. It's either a shift-transition, or else a reduce reaction and in the latter case, there is exactly one production to be used in the reduce-step. So that makes the table a legit LR(0)-table, in other words, the **simple parentheses grammar is LR(0) parseable.**

As before, the stack shown on the left, the remaining input on the right. For the stack, also the state of the machine is indicated as sub-script after the corresponding prefix of the stack. In particular, the **top-state**, i.e., current state of the parser machine shows on the right-most end of the stack. Initially, the stack is empty, and consequently the top-state is the DFA's initial state, here numbered 0. Of course, the left-most column is just line numbers ("stage" of the computation), it's not the *state*. Note also the **accept-action**. It corresponds to doing a reduce wrt. $A' \to A$ and reaching **empty stack**. apart from **\$** and A' (after that accept-reduction step).

As discussed earlier, the reduction "contains" the parse-tree, it builds it bottom up and reduction in reverse corresponds to a *right-most derivation* (which is "top-down"). The corresponding parse tree is shown in Figure 4.29b. □

stage	parsing stack	input	action
1	$\$_0$	((**a**)) $	shift
2	$\$_0(_3$	(**a**)) $	shift
3	$\$_0(_3(_3$	**a**)) $	shift
4	$\$_0(_3(_3\mathbf{a}_2$	)) $	reduce $A \to \mathbf{a}$
5	$\$_0(_3(_3A_4$	)) $	shift
6	$\$_0(_3(_3A_4)_5$	) $	reduce $A \to (\, A\,)$
7	$\$_0(_3A_4$	) $	shift
8	$\$_0(_3A_4)_5$	$	reduce $A \to (\, A\,)$
9	$\$_0A_1$	$	accept

(a) Run



(b) Parse tree

Figure 4.29: Simple parentheses

The LR-table shown contains **empty** slots. Another way to see it is that for the deterministic automata, like the one from Figure 4.27a is not complete, i.e., it has not explicit error state resp. we did not bother to show it. The DFA is constructed to define the legal stack content. Thus, hitting an empty slot means stumbling upon a **parse error.**

> As an important general invariant for LR-parsing: never put something "illegal" onto the stack.

Let's illustrate that, using again the same simple parenthesis grammar.

Example 4.7.17 (Simple parentheses: parsing of erroneous input). Let's pick up on the simple parenthesis syntax again from Example 4.7.15. Figure 4.30 shows two erronous situations. □

stage	parsing stack	input	action
1	$\$_0$	((**a**) $	shift
2	$\$_0(_3$	(**a**) $	shift
3	$\$_0(_3(_3$	**a**) $	shift
4	$\$_0(_3(_3\mathbf{a}_2$	) $	reduce $A \to \mathbf{a}$
5	$\$_0(_3(_3A_4$	) $	shift
6	$\$_0(_3(_3A_4)_5$	$	reduce $A \to (\, A\,)$
7	$\$_0(_3A_4$	$	????

(a) Unbalanced ((***a***)

stage	parsing stack	input	action
1	$\$_0$	( ) $	shift
2	$\$_0(_3$	) $	?????

(b) Unexpected input

Figure 4.30: Parse errors

Figure **??** summarizes the steps of an LR(0)-parser machine.

Let's conclude the section about LR(0)-parsing with two more examples.

Example 4.7.18 (DFA (non-simple) parentheses again: LR(0)?). Let's revistit the non-simple parentheses from Example 4.7.7 and the grammar from equation (4.26). See in

let s be the current state, on top of the parse stack

1. s contains $A \to \alpha.X\beta$, where X is a *terminal*
   - shift X from input to top of stack. The new *state* pushed on the stack: state t where $s \xrightarrow{X} t$
   - else: if s does not have such a transition: *error*
2. s contains a **complete** item (say $A \to \gamma.$): **reduce** by rule $A \to \gamma$:
   - A reduction by $S' \to S$: **accept**, if input is empty; else **error**:
   - else:

     pop: remove γ (including "its" states from the stack)

     back up: assume to be in state u which is *now* head state

     push: push A to the stack, new head state t where $u \xrightarrow{A} t$ (in the DFA)

Figure 4.31: Steps of the LR(0) parser machine

particular the LR(0)-DFA from Figure 4.23b. In this DFA, the states 0, 2, and 4 contain complete items and additionally have outgoing edges labelled by a non-terminal (concretely by **(**). That means, when being in one of those states, the machine can do a reduce step or a shift-step with **)**. In other words, there is a shift-reduce conflict in those three states and the grammar is not LR(0).

□

Example 4.7.19 (DFA addition again: LR(0)?). Let's revistit the addition-expressions from Example 4.7.6 and the grammar from equation (4.24). See in particular the LR(0)-DFA from Figure 4.24b.

State 1 in that DFA contains a complete item and an outgoing edge labelled +, so that state suffers from a LR(0) shift/reduce conflict.

In preparation of the following sections, we can consider whether it's possible to make the correct decision, given a concrete input.

In the context of Example 4.7.6, Figure 4.19a showed a run of the machine for a input **number** + **number** and pointed out that the stack content in stage 3 and 6 is the same, namely E, which corresponds to state 1 as (top)-state of the parser.

Concerning the rest of the input: in stage 3, the input continues with +, in stage 6, the complete input is parsed, i.e., the "next" symbol is **$**. Taking that into account the parser can make the correct decision: shift in state 1 with + as next symbol, and reduce, with **$** as next symbol.

Doing that systematically will lead to SLR-parsing and later to LR(1) and LALR(1) parsing. □

4.7.4 SLR parsing

LR(0) is not useful in practice, it's too weak. Basing decisions not just on the stack-content, but additionally on the next token(s), the **look-ahead**, increases expressiveness,

and the more look-ahead, the more powerful the parser. Besides having more or less look-ahead, there are also variations of *how* a look-ahead is taken into account.

Next we will present SRL(1) parsing [7] (**simple LR-parsing**), a minor variation of LR(0). The construction adds a bit of look-ahead in a particular way, which helps to avoid some reduce-reduce and shit-reduce conflict when doing a parse with the LR(0)-DFA. Not all such conflicts, SLR(1) is only moderately more powerful than LR(0). For realistic languages, it's mostly still seen as too weak, in particular since with LALR(1), there's a better alternative available, which basically does not require more resources during parsing. Of course, no amout of look-ahead can resolve all conflicts, there will always be grammars with shift-reduce conflicts, not matter the amount of look-ahead.

The SLR-parser will be based on the same principles as LR(0), but the format of the parse tables is slightly changed, compared to LR(0). However, the parser machine is *still* based on the **LR(0)-DFAs**.

The idea is pretty simple. When in doubt (i.e., when facing an LR(0)-conflict), look at the next symbol, perhaps that's enough to resolve the issue. That's what look-ahead is about anyway.

Why is that not LR(1), i.e., LR-parsing with one look-ahead? It's the "simple" way how look-ahead is used in SLR. We illustrated LR-parsing or shift-reduce parsing for the simplest case of LR(0)-parsing. There, the automaton is built using so-called LR(0)-items. Indepedent from the details of how that it done, important is that the DFA is built over the stack-content, which can be seen as an abstraction of the *past* of the bottom-up parse. But the *states* of the automaton do not contain information about the *future* of the parse, i.e., a look-ahead. That's why the items are called LR(0)-items and the machine an LR(0)-DFA. Also SLR uses this LR(0)-automaton construction, i.e., it uses a deterministic automaton machine *without* look-ahead. But, and that's the trick, it uses the look-ahead to disambiguate (some) situations, given a LR(0)-DFA state. The difference to full LR(1) is: LR(1) builds an automaton, whose **states** contains information about the past (as is the case LR(0)) and the future, in the form of the next symbol look-ahead. So, for LR(1), the look-ahead information is already used to construct the automaton, and that blows up the number of states (resp. the sized of the parser table) considerably. SLR(1), instead works with the smaller LR(0)-DFA, but tries do defuse some conflicts, if possible.

There are two conflicting situations, reduce-reduce conflicts and shift-reduce conflicts. The idea is actually quite simple for both situations. Both conflict situations involve a reduce step (there is no such thing as a shift-shift conflict). In the LR(0)-DFA, reduce steps are possible in states which contains complete items, i.e., items of the form $A \to \alpha\bullet$ with the dot at the end. Containing that dotted item is a sign that the word α is on top of the stack, and that a reduce step is possible.

In a **reduce-reduce** conflict, there are two complete items in a state, say $A \to \alpha\bullet$ and $B \to \beta\bullet$ (A and B may be the same non-terminal) (see Figure 4.32a). It also means that the top of the stack contains α and β; that's possible only one is a proper prefix of the other or both are the same word.

At any rate, the **follow sets** for the two non-terminals A and B contain all non-terminals (including **$**) that may occur in a sential form, following immedately occurences of A, resp. occurrences of B. If there is a shared symbol, contained both in the follow set of

A and of B, then the conflict remains unresolved. If, on the other hand, the two follow sets are *disjoint*, then the look-ahead to the next symbol can be used to make a decision whether to reduce with production $A \to \alpha\boldsymbol{.}$ or with $B \to \beta\boldsymbol{.}$. Of course, if there are more than two productions, one needs to check all pairs for that disjointness-condition.

Checking resp. defusing **shift-reduce** conflicts works similarly (see Figure 4.32b). The symptom for a shift-reduce conflict is a state in the LR(0)-DFA containing a complete item $A \to \alpha\boldsymbol{.}$ and a non-complete item $A \to \alpha_1\boldsymbol{.}\,\boldsymbol{a}\,\alpha_2$. Such a state has also an outgoing edge labelled with $\boldsymbol{a}$, which corresponds to a shift-step. In such a situation, the LR(0)-DFA has a choice between a shift-step with $\boldsymbol{a}$ and a reduce with the mentioned production. If a is *not* in the **follow-set** of A, then the conflict is resolved.

(a) Reduce-reduce

(b) Shift-reduce

Figure 4.32: LR(0) conflicts

The LR(0)-conflicts in the situations from Figure 4.32 are **resolved in SRL(1)**, if for reduce-reduce, resp. for the shift-reduce situation, the following conditions hold respecively:

$$\mathit{Follow}(A) \cap \mathit{Follow}(B) = \emptyset \qquad \text{resp.} \qquad \mathit{Follow}(A) \cap \{\mathbf{b_1}, \mathbf{b_2}, \ldots\} = \emptyset \tag{4.35}$$

In both cases, one has to check the *follow sets* of the involved non-terminals. If the respective intersection of sets are empty, there is no SLR(1)-conflict, otherwise there is. For the reduce-reduce situation as in Figure 4.32a, next symbol in the token input stream decides: if $\mathbf{t} \in \mathit{Follow}(A)$ then reduce using $A \to \alpha$, if $\mathbf{t} \in \mathit{Follow}(B)$ then reduce using $B \to \beta$. Similar for the shift-reduce situation as in Figure 4.32b: If $\mathbf{t} \in \mathit{Follow}(A)$ then *reduce* using $A \to \alpha$, if $\mathbf{t} \in \{\mathbf{b_1}, \mathbf{b_2}, \ldots\}$, then *shift*.

Example 4.7.20 (SLR and SLR-table: addition (one more time)). Let's revisit the addition example one more time. As hinted at already in Example 4.7.6 and as visible in the the LR(0)-DFA from Figure 4.24b, the grammar is not LR(0): the state numbered 1 contains a complete item *and* and outgoing edge labelled $+$.

But is there an SLR-conflict? The complete item in state one is $E' \to E\boldsymbol{.}$. So to apply the SLR-conditions, we need to calculate the follow set for E'. The follow-set for the (extra) start symbol is $\mathit{Follow}(E') = \{\$\}$.[9] So according to the condition for possible shift-reduce conflicts from equation (4.35), there is **no SRL(1)-conflict,** the grammar is SLR(1) parseable. Concretely, in state 1, the machine does a shift for $+$, and a reduce according $E' \to E$ for $\$$ (which corresponds to accept, in case the stack is empty).

[9] As a side remark: we don't need here to bother about the algorithms for the follow set and the first set. Under the convention that the (extra) start symbol never occurs on the right-hand side of a production, it's immediately clear that $\$$ is the only symbol in its follow set.

Next we "translate" the LR(0)-DFA to an **SLR(1) table.** See Table 4.14. It's conceptually the same format than the LR(0)-table. However, in an ok LR(0) table, i.e., one without conflicts, it was arranged a bit different. Without LR(0)-conflict, the reaction is either a reduce (with one particular rule) or a shift (though different symbols may lead to different successor states, or raising an error).

Now, with SLR(1), there can be states where both shifts can be done or a reduce, depending on the input. In the table, that's state number 1. Remember, the accepting action corresponds to a particular reduce-step. So the line labelled with 1 contains a shift and a reduce. What would also be possible, though not relevant in this example are states or lines with *different* reduce steps, depending on the input (the column of the table). That would correspond to a automaton, where reduce-reduce conflicts are resolved by the follow set criteria we discussed (see equation (4.35)).

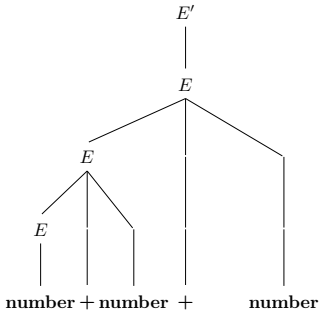
state	input			goto
	$\mathbf{n}$	$+$	$\$$	E
0	$s:2$			1
1		$s:3$	accept	
2		$r:(E \to \mathbf{n})$		
3	$s:4$			
4		$r:(E \to E+\mathbf{n})$	$r:(E \to E+\mathbf{n})$	

Table 4.14: SLR(1) parsing table (addition)

A parser-run is shown in Figure 4.33a resulting in a parse tree from Figure 4.33b.

stage	parsing stack	input	action
1	$\$_0$	$\mathbf{n}+\mathbf{n}+\mathbf{n}\$$	shift: 2
2	$\$_0\mathbf{n}_2$	$+\mathbf{n}+\mathbf{n}\$$	reduce: $E \to \mathbf{n}$
3	$\$_0E_1$	$+\mathbf{n}+\mathbf{n}\$$	shift: 3
4	$\$_0E_1+_3$	$\mathbf{n}+\mathbf{n}\$$	shift: 4
5	$\$_0E_1+_3\mathbf{n}_4$	$+\mathbf{n}\$$	reduce: $E \to E+\mathbf{n}$
6	$\$_0E_1$	$\mathbf{n}\$$	shift 3
7	$\$_0E_1+_3$	$\mathbf{n}\$$	shift 4
8	$\$_0E_1+_3\mathbf{n}_4$	$\$$	reduce: $E \to E+\mathbf{n}$
9	$\$_0E_1$	$\$$	accept

(a) Parser-run (reduction)



(b) Corresponding parse tree

Figure 4.33: Addition grammar

□

Let's have another look at the construction again, with another a grammar we have seen before.

Example 4.7.21 (SLR(1): parentheses again). Let's revisit the grammar for parentheses from equation (4.26) and Example 4.7.7. The LR(0)-DFA for the grammar has already been presented in Figure 4.23b.

The follow-set of the non-terminal S is

$$Follow(S) = \{), \$\} \ . \tag{4.36}$$

The SLR(1)-parsing table is shown in Figure 4.34a and a run of the parser machine in Figure 4.34b.

state	input			goto
	(	)	$	S
0	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	1
1			accept	
2	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	3
3		$s:4$		
4	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	5
5		$r: S \to (\ S\ )\ S$	$r: S \to (\ S\ )\ S$	

(a) SLR(1) parse table (parentheses)

stage	parsing stack	input	action
1	$\$_0$	( ) ( ) $	shift: 2
2	$\$_0({}_2$	) ( ) $	reduce: $S \to \epsilon$
3	$\$_0({}_2 S_3$	) ( ) $	shift: 4
4	$\$_0({}_2 S_3)_4$	( ) $	shift: 2
5	$\$_0({}_2 S_3)_4({}_2$	) $	reduce: $S \to \epsilon$
6	$\$_0({}_2 S_3)_4({}_2 S_3$	) $	shift: 4
7	$\$_0({}_2 S_3)_4({}_2 S_3)_4$	$	reduce: $S \to \epsilon$
8	$\$_0({}_2 S_3)_4({}_2 S_3)_4 S_5$	$	reduce: $S \to (\ S\ )\ S$
9	$\$_0({}_2 S_3)_4 S_5$	$	reduce: $S \to (\ S\ )\ S$
10	$\$_0 S_1$	$	accept

(b) Parser run ("reduction")

Figure 4.34: Parentheses

□

As seen in the examples, SLR(1) parsing tables look rather similar to the LR(0) one.. It reflects the differences of the LR(0)-criterion vs the SLR(1)-criterion for parsing, i.e. difference the LR(0)-algorithm vs. the SLR(1)-algorithm. Both tables have the same number of rows in the table, as they are based on the same DFA. Only the colums "arranged" differently. The LR(0) parser react **uniformely** per state resp. per row: it's either a shift (resp. goto) or else a reduce step (with given rule). The SRL(1)-version can react non-uniformely, **dependent** on the input.

It should be obvious, SLR(1) may resolve LR(0) conflicts, but if the follow-set conditions are not met, there are SLR(1) *reduce-reduce* and/or SLR(1) *shift-reduce* conflicts. That would result in non-unique entries in the slots of the SLR(1)-table.[10]

Figure 4.35 shows the steps of a SRL(1)-parser machine. It's a minor variation of the steps for the LR(0) variant from Figure 4.31.

Ambiguity & living with conflicts in LR-parsing

We have mentioned it multiple times: if the grammar is ambiguous, it's not useful for parsing; in particular it leads to conflicts, for bottom-up parsing as well as top-down parsing, and no matter the look-ahead. But there are ways to **live with ambiguity and conflicts**. One is, in some situations, to simply state that one wants certain associativies or precedences in an otherwise ambiguous grammar. If that works (which means, it makes sense and is understandable to speak about associativity an precedence, and one uses a tool

[10] by which it, strictly speaking, would no longer be an SLR(1)-table :-)

Let s be the current state, on top of the parse stack

1. s contains $A \to \alpha.X\beta$, where X is a terminal **and X is the next token on the input**, then
   - shift X from input to top of stack. The new *state* pushed on the stack: state t where $s \xrightarrow{X} t$
2. s contains a *complete* item (say $A \to \gamma.$) **and the next token in the input is in** $Follow(A)$: *reduce* by rule $A \to \gamma$:
   - A reduction by $S' \to S$: *accept*, if input is empty
   - else:

     pop: remove γ (including "its" states from the stack)

     back up: assume to be in state u which is *now* head state

     push: push A to the stack, new head state t where $u \xrightarrow{A} t$
3. if next token is such that neither 1. or 2. applies: **error**

Figure 4.35: SLR(1) algorithm

like a parser generator that supports that, that's s nice, and one can get rid of conflicts without massaging the grammar (or even redesigning the language). In that case, the parser generator will also not issue any warnings of potential conflicts; they are gone.

In the following we also discuss a different approach, this time really "living with a conflict", by prioritizing actions in case of conflicting situations; resp. letting the tool prioritize among them. A parser, when confronted with a conflict in the parser table, say, an LR-table, will not try out all possibilities, to find out which works, resp. if more than one works, it will not give back multiple parse trees, resp. multiple ASTs. It simply makes a choice. If the compiler writer knows how that choice is done, and if the programmer additionally know what consequence it has for the parse tree, resp. the AST (and is happy with the consequences), then it may be acceptable. For example, in an ambiguous expression grammar (we had some of those), one particular choice could make addition left-associative, and another, different one, could make it right-associative. If it's known the implementation always leads to a left-associative choice, maybe it's acceptable.

But it's risky. And the parser generator will complain about it in the form of a warning. So, I would rather not recommend it...

We discuss in the following a (well-known) situation, where the traditional choice a parser generator does leads to a desired outcome. The traditional choice of a bottom-up parser generator for shift-reduce conflict is **shift takes priority**. The example we use for illustration is the ambiguous grammar with **dangling elses**. With the grammar, in a nested situation of conditional, the else-part may or may belong to different open conditionals, it "dangles". The convention says, in such a situation, it should belong to the closest open conditional. As it turns out, prioritizing shifts over reduce steps does exactly that.

In case of reduce-reduce conflict, also the parser generator makes a choice, probably it prefers a production written earlier over alternatives written later in the grammar (like in a "first-match").

Generally, a warning about conflicts are not a good sign, so instead of thinking hard whether the conflict is resolved in an acceptable way, one should probably try to get rid of conflicts. So the example is better not read as advice how to deal with conflicts (or how to design a condition construct), but rather an opportinity to see what consequence reduces and shifts have in shaping the parse tree (and thereby the AST).

Example 4.7.22 (Ambiguous grammar (dangling else again)). We have seen the grammar illustrating the situation of dangling-else in the chapter about grammars. Let's repeat the the essence of that grammar, simplifying it a bit to keep the pictures smaller.

The grammar is given in (4.37), resp. in expanded form with the productions numbered in equation (4.38). The follow sets of the non-terminals are given in Table (4.15)

$$\begin{array}{lcl} S & \to & I \mid \textbf{other} \\ I & \to & \textbf{if } S \mid \textbf{if } S \textbf{ else } S \end{array} \qquad (4.37)$$

$$\begin{array}{lcll} S & \to & I & (1) \\ & \mid & \textbf{other} & (2) \\ I & \to & \textbf{if } S & (3) \\ & \mid & \textbf{if} S \textbf{ else } S & (4) \end{array} \qquad (4.38)$$

	Follow
S'	$\{\$\}$
S	$\{\$, \textbf{else}\}$
I	$\{\$, \textbf{else}\}$

Table 4.15: Follow sets

Figure 4.36 shows the LR(0)-DFA for the grammar. Since the grammar is ambiguous, there must be at least one SLR(1) conflict somewhere (and of course a LR(0), as well).

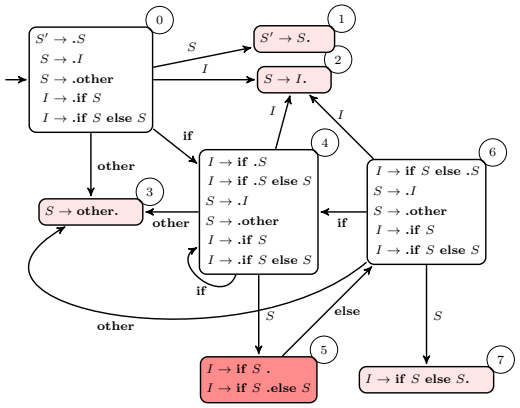


Figure 4.36: DFA of LR(0) items

Checking the previously shown conditions for SLR(1) parsing, one sees that there is a SLR(1) shift-reduce conflict in **state 5**: the follow-set of the left-hand side non-terminal I contains **else** (which represents the shift-step). In Table 4.16, *only* the shift-reaction

is added in the corresponding slot (not both shift and reduce action), since that is the default reaction of a parser tool, when facing a shift-reduce conflict.

state	input				goto	
	if	**else**	**other**	**$**	S	I
0	$s:4$		$s:3$		1	2
1				accept		
2		$r:1$		$r:1$		
3		$r:2$		$r:2$		
4	$s:4$		$s:3$		5	2
5		$s:6$		$r:3$		
6	$s:4$		$s:3$		7	2
7		$r:4$		$r:4$		

Table 4.16: SLR(1) parse table, conflict "resolved"

The *shift-reduce conflict* in state 5 is between a reduce with *rule 3* (not shown in the table) vs. a shift (to state 6), so the conflict is **resolved** in favor of *shift*. Note: the extra start symbol is left out from the table.

Figure 4.37 shows a run of the parser for nested conditionals, i.e., in a dangling-else situation. The machine resolves the conflict in favor of a shift. Figure 4.38 shows an similar run, this time favoring the reduce step.

stage	parsing stack	input	action
1	$\$_0$	**if if other else other $**	shift: 4
2	$\$_0\mathbf{if}_4$	**if other else other $**	shift: 4
3	$\$_0\mathbf{if}_4\mathbf{if}_4$	**other else other $**	shift: 3
4	$\$_0\mathbf{if}_4\mathbf{if}_4\mathbf{other}_3$	**else other $**	reduce: 2
5	$\$_0\mathbf{if}_4\mathbf{if}_4S_5$	**else other $**	shift 6
6	$\$_0\mathbf{if}_4\mathbf{if}_4S_5\mathbf{else}_6$	**other $**	shift: 3
7	$\$_0\mathbf{if}_4\mathbf{if}_4S_5\mathbf{else}_6\mathbf{other}_3$	**$**	reduce: 2
8	$\$_0\mathbf{if}_4\mathbf{if}_4S_5\mathbf{else}_6S_7$	**$**	reduce: 4
9	$\$_0\mathbf{if}_4I_2$	**$**	reduce: 1
10	$\$_0S_1$	**$**	accept

Figure 4.37: Parser run: preference on shift

Finally, Figure 4.39 shows the corresponding parse trees resulting from the two different ways to resolve the shift-reduce conflict. The one favoring the shift corresponds to the standard "dangling else" **convention**, that an **else** belongs to the last previous, still open (= dangling) if-clause.

□

The example serves two purposes: it sheds a light on how the dangling else problem can be "solved" by preferring as shift over a reduce reaction. More generally, it should,

stage	parsing stack	input	action
1	$\mathbf{\$}_0$	**if if other else other $**	shift: 4
2	$\mathbf{\$}_0\mathbf{if}_4$	**if other else other $**	shift: 4
3	$\mathbf{\$}_0\mathbf{if}_4\mathbf{if}_4$	**other else other $**	shift: 3
4	$\mathbf{\$}_0\mathbf{if}_4\mathbf{if}_4\mathbf{other}_3$	**else other $**	reduce: 2
5	$\mathbf{\$}_0\mathbf{if}_4\mathbf{if}_4S_5$	**else other $**	reduce 3
6	$\mathbf{\$}_0\mathbf{if}_4I_2$	**else other $**	reduce 1
7	$\mathbf{\$}_0\mathbf{if}_4S_5$	**else other $**	shift 6
8	$\mathbf{\$}_0\mathbf{if}_4S_5\mathbf{else}_6$	**other $**	shift 3
9	$\mathbf{\$}_0\mathbf{if}_4S_5\mathbf{else}_6\mathbf{other}_3$	**$**	reduce 2
10	$\mathbf{\$}_0\mathbf{if}_4S_5\mathbf{else}_6S_7$	**$**	reduce 4
11	$\mathbf{\$}_0S_1$	**$**	accept

Figure 4.38: Parser run: preference on reduce

(a) shift-precedence: conventional

(b) "wrong" tree

Figure 4.39: Parse trees for the "simple conditionals"

using that standard situation, give a feeling how generally a shift-vs-reduce changes the structure of the parse-tree (and indirectly most probably thereby also the AST). It's an issue of associativity and precedence (at least when dealing with binary operators), and we will see that in the following standard setting of expressions.

Using **ambiguous grammars** may have the advantages that they are often *simpler*. Of course, an ambiguous grammar is is guaranteed to have conflicts. Sometimes, ambiguous parts of a grammar can be resolved by specifying *precedence* and *associativity*, as discussed earlier, and doing so is also supported by tools like `yacc` and `CUP` ...

One can also do by instructing the parser to explicitly instruct the parser what to do in a conflicting situation. In the dangling-else example, preferring shift over reduce leads to the intended outcome, and next example shows how different conflict resolution decisions lead to different associativities and precendences.

Example 4.7.23 (Precedences and associativity). Consider the following expression grammar:

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + E \mid E * E \mid \mathbf{number} \end{array} \tag{4.39}$$

The corresponding LR(0)-DFA is shown in Figure 4.40.

Let's look at two states with conflicts. In **state 5**, the parser is in a strate that contains $E + E$ on top as can be seen from the complete item it "contains". So that means a reduce

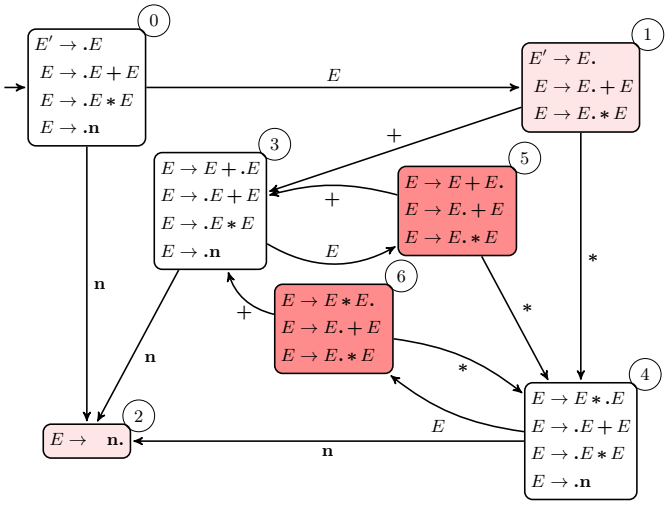


Figure 4.40: DFA for + and ×

□

steps is possible, and also a shift step, namely with + and with ∗. Based on the next symbol, i.e., by manually making the decision for the parser based on that automaton, a + as input should resolve the shift-reduce conflict by reduce, to achieve **left-associativity** of addition. For a ∗, the machine should do a shift, to achive **precedence** of multiplication over addition. Finally, for **$**, the machine must do a reduce, a shift is anyway not allowed.[11]

State 6 is analogous: the stack contains $E * E$ on top, and again there is a shift-reduce conflict and for the intended behavior, the machine should do the following. For **$** as input: do a reduce, of course. For input +, do a reduce, as ∗ has **precedence** over +, and for input ∗, do a reduce, as ∗ is **left-associative**. The corresponding parse table is shown in Table 4.17.

state	input				goto
	n	+	∗	**$**	E
0	$s:2$				1
1		$s:3$	$s:4$	accept	
2		$r: E \to \mathbf{n}$	$r: E \to \mathbf{n}$	$r: E \to \mathbf{n}$	
3	$s:2$				5
4	$s:2$				6
5		$r: E \to E + E$	$s:4$	$r: E \to E + E$	
6		$r: E \to E * E$	$r: E \to E * E$	$r: E \to E * E$	

Table 4.17: Parse table for + and ∗

The exercises extend that example to additionally deal with exponentiation, which has still a higher precendence than multiplication and is conventionally *right-associative.* □

The previous example showed in a typical situation, how to fiddle with the parse-table of

[11] LR(0)-DFAs never show transitions labelled by **$**. However, LR(0)-tables or SLR(1)-tables contain a column for **$**.

an ambiguous grammar to obtain the intended disambiguations. We know for examples like that, one can obtain the intended associativies and precedences also by massaging the grammar with a technique known as *precedence cascade*. Let's do the previous example with this technique as well, for a comparison.

Example 4.7.24 (Unambiguous grammar). The following is a reformulation of the language specified by the unambigours grammar from equation (4.39). The table on the right-hand shows the follow-sets for the involved non-terminals (we come back to the right-hand column named "states" afterwards). Note, the grammar uses E' as an extra start-symbols, something that is necessary for the LR(0)-DFA construction. As a consequence, E' has **\$** as the *only* member in its follow-set.

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * \mathbf{n} \mid \mathbf{n} \end{array} \qquad (4.40)$$

	Follow	*states*
E'	$\{\$\}$	1
E	$\{\$, +\}$	$4, 6$
T	$\{\$, +, *\}$	$3, 7$

Figure 4.41 shows the corresponding LR(0)-DFA.

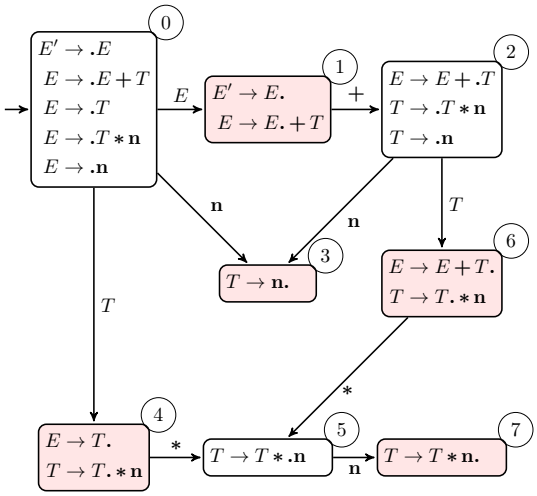


Figure 4.41: DFA for unambiguous grammar for + and ×

On closer inspection, the DFA now is SLR(1). We don't elaborate the argument for that fully (for example by filling out a SLR(1)-table and check for duplicate entries). To check it one should in particular check states with *complete* items. Those are the states *1,4,6,3, and 7*. The above table lists the follow-sets for all the non-terminals. The last column in the table indicates for which state(s) of the automaton the particular follow set is relevant. For example, the follow set $\{\$, +\}$ for E is relevant for states 4 and 6, and since there is no overlap with symbole $*$ that labels the outgoing edges from those two states, there is no SRL(1) shift-reduce conflict (only a LR(0) shift-reduce conflict). Same for the other states. Since there are no states containing two complete items, there's no need to check the corresponding SLR(1) criterion, and so the **grammar is SRL(1).** □

4.7.5 LR(1) parsing

LR(1) parsing is top-down **shift-reduce parsing with one look-ahead**. It's also known as *canonical* LR(1) parsing. It is usually considered as unecessarily complex in that it leads to parser machines with too many states and often one is content with LALR(1), which we discuss afterwards. Indeed, LR(1) is a stepping-stone towards LALR(1).

Now, to get that out of the way. We have just discussed SLR(1), a parser technique using a DFA that takes into account as look-ahead of 1 to resolve some conflicts, isn't that shift-reduce parsing with one look-head and thus LR(1)? Indeed, SLR(1) uses a *look-ahead*, but only *after* it has built a non-look-ahead DFA based on **LR(0)**-items. And LR(1) parser in contrast generalizes the idea of the LR-DFA by using a look-ahead of 1 already in the construction. The states of such an LR(1)-DFA is based not on LR(0)-items, but of so-called **LR(1)-items**. That normally leads to a machine with many more states than a LR(0)-DFA (but based on the same principles), and LALR(1)-parsing (presented later) takes that larger machine, and *strips-off* the extra look-ahead information, collapsing it to an automaton that has the same amount of states than the LR(0)-one, but still can make more fine-grained decisions than an LR(0)- or SLR(1)-parser.

> So one could say, SLR(1) is improved LR(0) parsing, LALR(1) is crippled LR(1) parsing.

Let's illustrate the LR(1) construction with the help of an example.

Example 4.7.25 (Assignment grammar fragment). Let's take the following grammar(s). The one on the left could be a fragment of a grammar for some programming language. Concretely for the construction, we focus on some simplified subset of that, shown in (4.41).

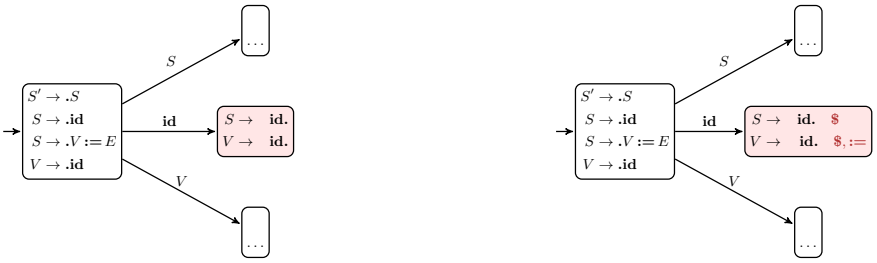
$$\begin{array}{rcl} \textit{stmt} & \to & \textit{call-stmt} \mid \textit{assign-stmt} \\ \textit{call-stmt} & \to & \textbf{identifier} \\ \textit{assign-stmt} & \to & \textit{var} := \textit{exp} \\ \textit{var} & \to & [\,\textit{exp}\,] \mid \textbf{identifier} \\ \textit{exp} & \to & \textit{var} \mid \textbf{number} \end{array} \qquad \begin{array}{rcl} S & \to & \textbf{id} \mid V := E \\ V & \to & \textbf{id} \\ E & \to & V \mid \textbf{n} \end{array} \tag{4.41}$$

Table **??** shows the follow-sets for the non-terminals, and Figure 4.42 shows one state of the LR(0)-DFA for the grammar, actually its initial state.

	First	*Follow*
S	**id**	**$**
V	**id**	**$**, :=
E	**id**, **number**	**$**

Table 4.18: Follow sets

The reddish states has a LR(0) reduce-reduce conflict, and that cannot be resolved by looking at the follow-sets, especially the follow-sets of E and V, and thus there is also an SRL(1) reduce-reduce conflict (on the symbol **$**). This shows that SLR runs into



(a) LR(0) with SLR(1) criterion

(b) Extra follow-set information

Figure 4.42: Transitions of the LR(0)-NFA

trouble with rather simple common notation. The example here is inspired by Pascal, but analogous problems exist in C or similar, so it's not an esoteric or artificial illustration of a situation, where SLR(1) is not good enough. The problematic situation in the larger of the two grammars, as we will see on the next slide, concerns identifiers (resp. variables as left-hand side of an assignment or as a call expression). In the simplified grammar, there's not call-statement, but the problem is the same.

Figure 4.42b shows the same part of the automaton, with follow-set information added. The red terminals are not part of the state, they are just shown for illustration (representing the follow symbols of S resp. of V). The LR(1) construction sketched in the following builds in one additional look-ahead symbol officially as parts of the items and thus states.

Let's look at Figures 4.44. The (sketch of the) automaton here looks pretty similar to

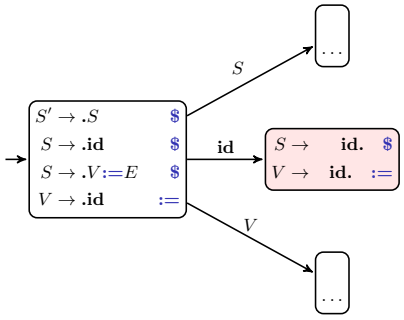


Figure 4.43: Look-ahead information as part of the items

the previous one. However, we should think *now* of the non-terminals as officially part of the items. The interesting piece in this example is the **id**-transition from the initial state to the state containing the items $\rightarrow$ **id.** and $V \rightarrow$ **id.**. That was the state which previously suffered the reduce/reduce conflict (on the following symbol **$**). Now, without showing the construction in detail, the interesting situation is, in the first state, the item $S \rightarrow \mathbf{.}V \mathbf{:=} E, \mathbf{\$}$. With the **.** in front of the V, that's when we have to take the ϵ-closure into account, basically adding also the initial items (here one initial item) for the productions for V into account. Now, by adding that item $V \rightarrow$ **.id**, we can use the additional "look-ahead piece of information" in that item to mark that V was added to the closure when being **in front of an :=**. That leads (in this situation) to the item of

the form $[V \to \textbf{.id}, \textbf{:=}]$. This information is more specific than the knowledge about the general follow-set of V, which constains **:=** and **$**. Now, by recording that extra piece of information in the closure, the state **"remembers"** that the only thing at the current state that is allowed to follow the V is the **:=**. That will defuse the discussed conflict, namely as follows: if we follow the **id**-edge, we end up in the state on the right-hand side. Such a transition does not touch the additional new look-ahead information (here the **$** resp the **:=** symbol). See also the corresponding NFA-rule later. Thus, in the state at the right-hand side, the reduce-reduce conflict has disappeared! □

So that's the core of LR(1) parser automata: adding precision in the states of the automaton already: Instead of using LR(0)-items and, when the LR(0)-DFA is done, try to add a little disambiguation with the help of the follow sets for states containing complete items, it's more expressive to **make more fine-grained items** from the very start. This is done with so-called **LR(1) items**, which are like LR(0)-items with additional follow information. That' the look-ahead. This additional look-ahead information leads to a proliferation of states, enlarging the automaton.[12] The form if the new items is as follows

$$[A \to \alpha.\beta, \mathbf{a}] \tag{4.42}$$

where **a** is terminal or token, including **$**.

Example 4.7.26 (Assignments (LR(1))). For the assignment grammar from Example, Figure 4.44 shows the LR(1)-DFA. Part of it has been depicted already in Figure 4.43.

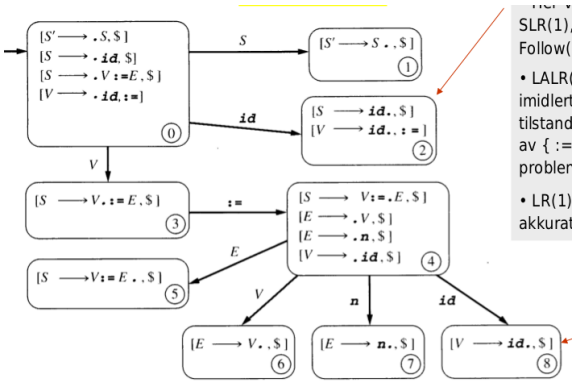


Figure 4.44: LALR(1)-DFA (or LR(1)-DFA)

In particular, as already discussed, state *2* has a SLR(1) reduce- reduce conflict, but with the LR(1) construction, the conflict has been resolved. □

Next we show the construction of the LR(1)-automaton. We only show the NFA, not the DFA as well. The construction of the transitions are shown in Figure 4.45. X represents

[12] Not to mention if we wanted look-ahead of $k > 1$, which in practice is not done, though.

terminals as well as non-terminals. For the case of the ϵ-transition in Figure 4.45b, the edge is added for all

$$B \to \beta \mid \ldots \quad \text{and all} \quad \mathbf{b} \in \mathit{First}(\gamma\mathbf{a})$$

(a) arbitrary symbol (b) ϵ-transition

Figure 4.45: LR(1) parser machine transitions

The construction from Figure 4.45b also contains the for $\gamma = \epsilon$, shown in Figure 4.46,adding a transition for all $B \to \beta \mid \ldots$.

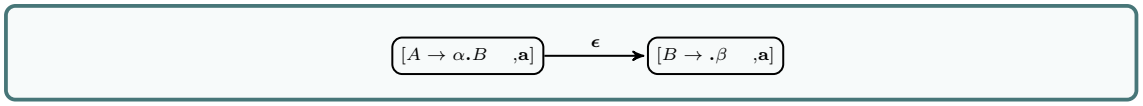


Figure 4.46: ϵ-transition, special case ($\gamma = \epsilon$)

The non-ϵ-transitions are simpler insofar that the look-ahead part of the idem does not change.

In the DFA, there are only 2 states which have a connection to ϵ-transitions. Since it's a DFA, there are technically no such transition, the items are directly part of the state (via closure). The two state where closure play a role are the states 0 and 4. In the initial state 0, the closure takes two steps. In the first step, the second and third item are added (with look-ahead **$**) and the the second step, the last item is added, with look-ahead **:=**, and the reason why it's an assign, because in the third item, the dot is in front of V **:=**, i.e., the item contains **.**V **:=**.

4.7.6 LALR(1) parsing: collapsing the LR(1)-DFA

Having covered canonical LR(1) parsing to some extent, we have a look at LALR(1) parsing. The abbreviation stands for Look-ahead LR-parsing, but the names does not say much. We hinted at core idea before: LALR(1) is another shift-reduce parser, so it works analogous to the variations covered before. The states of the corresponding DFA come form taking the states of the LR(1)-DFA and "collapse" its into a smaller one, combining multiple states of the larger automaton into a single state of the LALR(1). Let's just look at an example. To make it managable, we take a quite small grammar, the one for simple parentheses from Example 4.7.15.

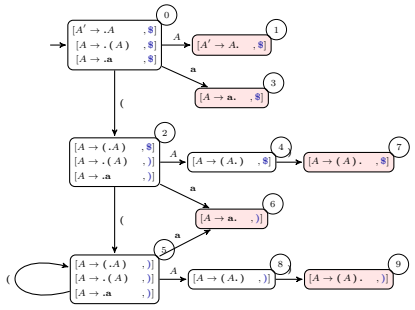
Example 4.7.27 (Simple parentheses). Let's look at the following grammar

$$A \to \mathbf{(}\ A\ \mathbf{)} \mid \mathbf{a}\ .$$

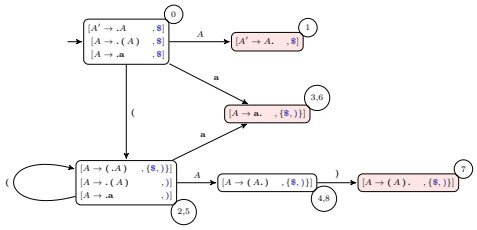
We have seen that grammar already in equation (4.34). Earlier, the simple parentheses grammar was an example of a grammar actually parseable with a LR(0)-parser, so there

is actually no need to use a more powerful one like am LR(1) or LALR(1) parser. But we use the quite simple grammar here just to illustrated the LALR(1) idea.

Figure 4.47a shows the LR(1)-DFA for the grammar with 10 states. We have sketched the construction for the non-deterministic version, here we show the DFA.



(a) LR(1) DFA



(b) LALR(1) DFA

Figure 4.47: LR(1) and LALR(1) DFAs

As explained, the states are built from LR(1)-items, i.e., LR(0)-items plus look-ahead information. The information **without** the look-ahead information is called the **core** of such an item. The core of the LR(1) item is of course an LR(0). Then the LALR(1) automaton is built by collapsing LR(1) states with the same core. The result for the example is shown in Figure **??**.

Joining states of a DFA of course should raise a red flag: it could turn a deterministic automaton into a non-deterministic one. However, two LR(1)-states with the same cores have the same outgoing edges, and those lead to states with the same core again. So, fortunately, doint the collapse result in a deterministic automaton. So the states of the almost look constructed like the ones for the LR(0)-DFA (with a detour via the LR(1)-DFA), but each individual item "inside" the state has still look-ahead information attached: the **union** of the "collapsed" items. Especially for states with *complete* items, the information in $[A \to \alpha, \mathbf{a}, \mathbf{b}, \ldots]$ is **smaller** than the follow-set of A. That results in less unresolved conflicts compared to SLR(1). □

4.7.7 Concluding remarks of LR / bottom up parsing

Before touching upon error-handling, let's wrap it up with some concluding remarks, mostly repeating things mentioned earlier. All all constructions (here) based on BNF (not EBNF). That style of parsers have shift-reduce and reduce-reduce **conflicts**, for instance due to ambiguity, and those can be solved by in different ways. One can try to reformulate the grammar, but generate the same language.[13] One may use *directives* in parser generator tools like `yacc`, `CUP`, `bison`, like specifying precedence and associativity of operators. One can also try to solve conflicts solve them later via *semantical analysis* (not discusess. Of course, not all conflics are solvable, also not in LR(1), or LR(k), for instance for ambiguous languages.

[13] If designing a new language, there's also the option to massage the language itself. Note also: there are *inherently* ambiguous *languages* for which there is no *unambiguous* grammar.

	advantages	remarks
LR(0)	defines states *also* used by SLR and LALR	not really used, many conflicts, very weak
SLR(1)	clear improvement over LR(0) in expressiveness, even if using the same number of states. Table typically with 50K entries	weaker than LALR(1). but sometimes enough. Ok for hand-made parsers for *small* grammars
LALR(1)	almost as expressive as LR(1), but number of states as LR(0)!	method of choice for most generated LR-parsers
LR(1)	*the* method covering *all* bottom-up, one-look-ahead parseable grammars	large number of states (typically 10M of entries or more), mostly LALR(1) preferred

Table 4.19: Shift-reduce parsing overview

Remember: once the *table* specific for LR(0), ... is set-up, the parsing algorithms all work *the same*

4.7.8 Error handling

Let's also talk shortly about error handling, without going very deep here. As as minimal requirement, as for top-down handling: Upon stumbling over an error, i.e., a deviation from the grammar used for parsing, one should give a *reasonable & understandable* error message, indicating also error *location*. One option is to stop parsing if that happens. An alternative is to continue, this is known as error *recovery*. Of course one cannot really recover from the fact that the program has an error, a syntax error is a syntax error, the parser could still move on. That may involve to jump over some subsequent code, until the parser can *pick up* normal parsing again. That allows to check code even following a first error. When doing recovery, one should avoid to report an avalanche of subsequent *spurious* errors, those just "caused" by the first error. It's desirable to avoid error messages that only occur because of an already reported error. In general, recovering and pick up normal course of action is harder for syntactic errors than for semantic errors, where error recovery is also a useful technique.

One should also report error as early as possible, if possible at the *first point* where the program cannot be extended to a correct program. For bottom-up parsing and when doing error recovering, one should watch out that the parser does not end up in an *infinite loop*. without reading any input symbols.

Earlier, in connection with top-down parsing, we already remarked that it's not always clear what is a good error message, and illustrated that with a small example.

Error recovery in bottom-up parsing

We shortly look at a simple form of error recovery for shift-reduce parsing, known as **panic recovery**. When hitting an error, the parser does not stop, but reacts with the following

recovery steps. It pops parts of the stack, ignores parts of the following input until the parser is or seem to be *back on track.* We shortly look how that is done.

A additional technical problem with error recovery is the question of (*non-*)determinism. Of course the construction of the parser table leads to a deterministic parser machine, with all conflicts resolved, but the construction is done assuming **under normal operation.** When hitting an error and upon clearing parts of the stack and ignoring input), there is a priori no guarantee that it's still deterministic in the larger picture, for instance, it may not be determined when and how to continue normal operations. The reaction is typically done by some **heuristic** needed, like indeed panic mode recovery.

It's easy to see when the parser stumbles on an error; it happens in the presentation so far if there is no entry in the parser table covering the situation. It's less easy when it's time to try a **fresh start**, i.e. determining how much of the stack content should be popped and how much input should be ignored.

The idea of panic mode recovery is to take a possible **goto action as promising fresh start**. So, after detecting an error, the parser backs off and takes the **next such goto-opportunity**. Here a bit more detailed:

Panic mode recovery heuristic

1. *Pop* states for the stack *until* a state is found with non-empty **goto** entries
2. - If there's legal action on the current input token from one of the goto-states, push token on the stack, *restart* the parse.
   - If there's several such states: *prefer shift* to a reduce
   - Among possible reduce actions: prefer one whose associated non-terminal is least general
3. if no legal action on the current input token from one of the goto-states: *advance input* until there is a legal action (or until end of input is reached)

Let's have a look at an artificial example. Let's assume the parse table from Table 4.20 and a run as shown in Table 4.21.

state	input					goto			
	...	)	**f**	**g**	...	...	*A*	*B*	...
...									
3							*u*	*v*	
4			–				–	–	
5			–				–	–	
6		–	–				–	–	
...									
u		–	–	reduce ...					
v		–	–	shift : 7					
...									

Table 4.20: Shift-reduce parse table

At the beginning of the shown behavior, in the first line, the machine is in state 6, where the next input **f** cannot be handled; the corresponding slot in the parse table is empty and also a goto-move is not an option.

	parse stack	input	action
1	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3(_4\mathbf{d}_5\mathbf{e}_6$	**f) g h** ... **$**	no entry for **f**
2	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3B_v$	**g h** ... **$**	back to normal
3	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3B_v\mathbf{g}_7$	**h** ... **$**	...

Table 4.21: Panic mode recovery run

What happens in that situation, that the part of the stack highlighted on red is popped off. The head-states for the corresponding stack contents do not support a goto-step. The first such state that does support a goto move is state 3. In that state there are two goto-options, with A and with B, and B is chosen (the example does not show the grammar, so it does not provide details why that would be). At any rate, the parser then is in state v. Since in this state, **f** and **)** on the input cannot hand, the parser jumps over those until seeing the next acceptable input, which is **g**. Concerning the **f** and **)** that cannot be handled, the parser does not complain and report them as additional errors.

Panic mode recovery is a simple heuristic, but one has to watch out, as without further precautions, the recovering parser **may loop forever**. That is illustrated in Table 4.23. It shows a looping run for a parser for expression, a relevant part of the parse Table is sketched in Table 4.22.

	exp	*term*	*factor*
goto to	10	3	4
with **n** next: action there	—	reduce r_4	reduce r_6

Table 4.22: Parse table fragment

	parse stack	input	action
1	$\$_0$	**(n n) $**	
2	$\$_0(_6$	**n n) $**	
3	$\$_0(_6\mathbf{n}_5$	**n) $**	
4	$\$_0(_6factor_4$	**n) $**	
6	$\$_0(_6term_3$	**n) $**	
7	$\$_0(_6exp_{10}$	**n) $**	panic!
8	$\$_0(_6factor_4$	**n) $**	been there before: stage 4!

Table 4.23: Panic mode loop

An error is raised in in stage 7, where no legal action is possible; also informally one sees that at that point, the input cannot be extended to a syntactically correct expression. The panic reaction is to pop off state pop-off exp_{10}. In the new head state 6, there are two possible goto reactions. Since there is no shift step possible, the parser needs to decide between the two reduces. Since *factor* is less general, the parser makes a reduce step that replaces *exp* by *factor*. At that point the parser is in a configuration which is identical one one from before.

How to avoid looping panic? One sure has to take precautions to detect the loop (i.e.

check if one revisits previously seen configurations. If a loop detected: doen't repeat but do something else, for instance, pop-off more from the stack, and try again or pop-off and *insist* that a shift is part of the options. Or of course, give up.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools*. Pearson,Addison-Wesley, second edition.

[2] Appel, A. W. (1998a). *Modern Compiler Implementation in Java*. Cambridge University Press.

[3] Appel, A. W. (1998b). *Modern Compiler Implementation in ML/Java/C*. Cambridge University Press.

[4] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samuelson, K., Wegstein, B. V. J. H., van Wijngaarden, A., and Woodger, M. (1963). Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6:1–17.

[5] Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(113–124).

[6] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

[7] DeRemer, F. L. (1971). Simple lr(k) grammars. *Communications of the ACM*, 14(7).

[8] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

Index