Course Script

INF 5110: Compiler construction

INF5110, spring 2024

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

The chapter resp. the text about parsing is under re-work. That particular holds for the section about first-and-follow set. The *content* will be unchanged, but the "story-line" (the way definitions are motivated and explained) is being rewritten. Also some definitions (of the first- and follow sets) are presented simpler. But as said, the core is unchanged. (9.02.2024)
Later in the semester, an **updated** version will be replace this one (as currently not all texts are adapted to the changed presentation, and there are some loose ends).

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 an unwelcome property of a grammar, we did not really impose restrictions on grammars. That will change in this chapter. For parsing, the full expressiveness 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 to 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, like giving insight into the nature of the syntactic violation, 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*.

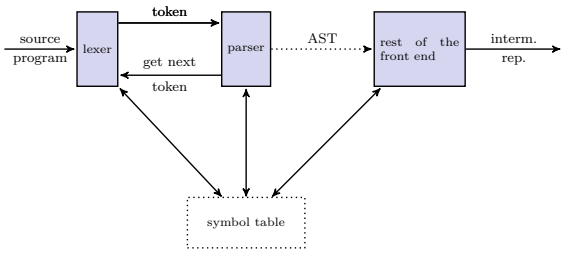


Figure 4.1: Lexer, parser, and the rest

The *symbol* table is an important data structure, *shared* between phases. It 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 unambiguous 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 (and typically the abstract syntax tree, this one for real, not just conceptually).

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 context-free languages is not really needed or aimed at 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 execution time in a compiler may be better spent elsewhere, like for optimization or semantical analysis).

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

Concerning grammar classes, *maaaany* have been proposed & studied, including their relationships, parseability, 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: (Some) 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 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 expressiveness of a grammar class. Another thing one can see on the picture: with the same amount on look-ahead, bottom-up parsing (the LR-kind) is more expressive that top-down parsing (the LL-kind).

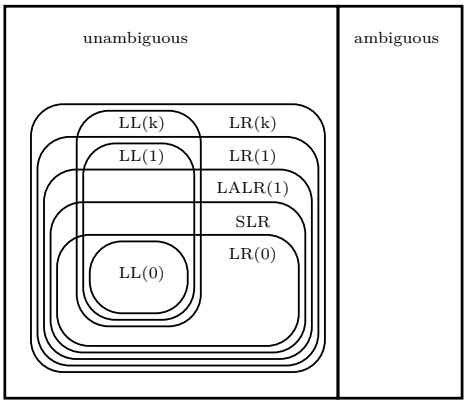


Figure 4.2: Relationship of some grammar (not language) classes (taken from [1])

The picture mentions SLR and LALR(1), positioned between LR(0) and LR(1). The LR- and the LL-parsers realize, in a way, the concept of bottom-up, resp. top-down parsing in a pure or canonical way. SLR and LALR(1) are variations of 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 well, and, 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-upe. Parsing is about the following: given a word (as a sequence of tokens or non-terminals), determine whether that word can be *derived* in the given grammar. Being derivable means there exists a derivation and thus a parse tree. In all but degenerated cases, each word can be derived in different ways; in particular, there can be *different orders of expansion steps* in a derivation. For an unambiguous grammar, however, there will be only *one* derivation *tree* or parse tree for each derivable word; that's the definition of being unambiguous. 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 concretely 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 explored or traversed, rather than actually built. Still, the traversal is top-down. For bottom-up parsing, the traversal works bottom-up.

Before we look in more detail at how that works, 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 concretely on a sequence of *tokens* not characters (but how the input pieces are called is not central to the idea of such a machine).

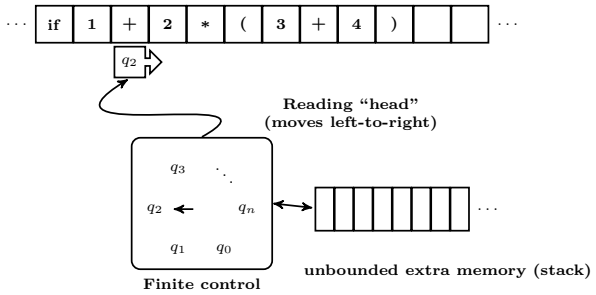


Figure 4.3: Schematic view on a "parser machine"

The picture hints at a fact about the kind of "machines" needed for parsing. For scanning, the machines were finite-state automata; see the very similar corresponding schematic picture for lexers resp. finite state automata from earlier.

For recognizing context-free languages, i.e., for parsing, we **don't define** the corresponding machine model in its generality. They are called **push-down** automata, which are finite-state automata equipped with an additional component of unbounded memory. The memory can be read and written to, but only in stack-like manner. So the extra and unbounded 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 strictly more expressive than deterministic ones (which implies, there's no hope to carry over the powerset construction we encountered earlier to such machines now). 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.

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. 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. Recursion, of course, implicitly uses a *stack*, though one does not explicitly operate with a push-down machine. Later, for bottom-up parsing, we will indeed work with some automata equipped with a stack, thus some form of push-down machines. We will, however, be more interested in the working and construction of the particular ones needed for bottom-up parsing, so neither there will we introduce the *general* concept of pushdown automaton.

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 the chapter about grammars. There it was introduced in discussing the concept of **precedence cascade.** The grammar here, while using factors and terms, is not the

one mentioned earlier, 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 4.2.1 (Derivation (factors and terms)). Let's assume the following grammar:

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

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

$\underline{\mathit{exp}} \Rightarrow$
$\underline{\mathit{term}}\ \mathit{exp}' \Rightarrow$
$\underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ \underline{\mathit{term}'}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ \cancel{\epsilon}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ \underline{\mathit{exp}'} \Rightarrow$
$\mathbf{number}\ \underline{\mathit{addop}}\ \mathit{term}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ \cancel{+}\ \mathit{term}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \underline{\mathit{term}}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ \underline{\mathit{term}'}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ \underline{\mathit{mulop}}\ \mathit{factor}\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ \cancel{*}\ \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathit{exp}\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathit{exp}\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \underline{\mathit{exp}}\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \underline{\mathit{term}}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number}\ \underline{\mathit{term}'}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number}\ \cancel{\epsilon}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number}\ \underline{\mathit{exp}'}\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number}\ \underline{\mathit{addop}}\ \mathit{term}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number}\ \cancel{+}\ \mathit{term}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \underline{\mathit{term}}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ \underline{\mathit{term}'}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ \cancel{\epsilon}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ \underline{\mathit{exp}'}\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ \cancel{\epsilon}\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ )\ \mathit{term}'\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ )\ \underline{\mathit{term}'}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ )\ \cancel{\epsilon}\ \mathit{exp}' \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ )\ \underline{\mathit{exp}'} \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ )\ \cancel{\epsilon} \Rightarrow$
$\mathbf{number}\ + \mathbf{number}\ * (\ \mathbf{number} + \mathbf{number}\ )$

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

The shown deriviation is **left-most**, and it corresponds to the parse tree of Figure 4.4.

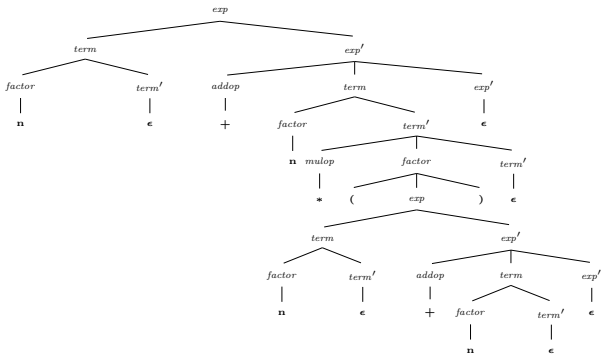


Figure 4.4: Parse tree

The tree (as usual) no longer contains 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.[1] The tree is an example of a *parse tree.* □

In the previous example, we mentioned that we are doing a left-most derivation, without really explaining what that is. It is easy enough, but before we define it, let's discuss aspects of derivations, which leads to the definition of left-most (or right-most etc.) derivations.

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?

In an unambigous grammar, the input *determines* the parse tree, but does a given parse tree determine the derivation? Well, **no**, in general there are different derivations for a given derivation tree.[2] In general one parse tree corresponds to multiple derivations, as

[1] The order is not important. Actually, it's not even important, once finished, that a the tree was done in a top-down manner.

[2] Some degenerated grammars (which ones?) may have exactly one derivation per word and per parse-tree, but it's an aberration and unintesting

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 decisions?

The issues hang together, but let's start with the first item. The general form of a derivation of a word w 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 $A \to \beta$. Given a target word to derive w and having arrived at α in the process, one can distinguish **two principle sources of non-determinism** for doing the **next step:**

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, one can count 3 issues. 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. The word w, however, 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 needs 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-terminals 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 independent from the shape of the surrounding letters. As 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 4.2.2 (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 $\Rightarrow_r$.

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, as A is the left-most non-terminal. As mentioned, the derivation in Example 4.2.1 was left-most.

That was the *easy* part of non-determinism: 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 4.2.3 (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 are connected to 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 (for instance left-most) non-terminal? Choosing different rules will lead to *different* parse-trees. Assuming an unambiguous grammar, there is only one tree for a given word, however. So if there is a choice between two productions and assuming that the word of terminals is ultimately derivable, **there is only one right choice,** all others would be wrong.

That's the core problem of parsing: how to arrive at that right decision?

Let's assume then for the discussion an unambiguous grammar and let's focus on the "which-production-to-apply" non-determinism as the only interesting one. Assume further a left-most derivation in the form of equation (4.3). Now, what can influence the decision of choosing $A \to \beta$ as in the derivation over an alternative, say $A \to \gamma$?

Doing a left-most derivation, the left of the non-terminal A at that point in the derivation is a word of *terminals* (called w_1 in the derivation from equation (4.3)). The target word w is of the form $w_1 w_2$, i.e. w_1 is a prefix of w.[3] The prefix w_1 represents at the point of decision making, the *past* of the derivation, and w_2 the still-to-parse *future*. very generally speaking, with the derivation ending in $w = w_1 w_2$, the target word w should determine the next step at each point.

[3] That's because we are dealing with *context-free* grammars. It would also hold for context-sensitive grammars, but not for the most general grammars (level 0).

The parser has a memory (states plus an unbounded stack), where it remembers information about the past of the parse, and as we insist on deterministic machines, the past parse determines the current configuration of the parser. The decision of what to do next is based of course on that configuration, which represents the past, but a parser can and should also take the "future" into account (in the derivation, the word w_2). Looking into the future is called the **look-ahead**. As a minimal requirement for a (deterministic) top-down parser is that taking into account all of the past plus all of the future must determine the next step. That's just a reformulation of the requirement that during the derivation, the word w must determine the next step of the parser.

If that requirement is met, of course each word in the language of the given grammar has exactly one parse tree (determined by the deterministic parsing process). In other words the grammar is unambiguous (which is what we assume anyway). However, a parser that bases its decisions on an unbounded look-ahead (and an unbounded memory to remember an arbitrarily long past) unrealistic. Realistic parsers use 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 such a powerful parser make always the right decision (for an unambiguous grammar)?

> It may seem plausible that a top-down parser with unbounded look-ahead, 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. What makes that plausible for an unambigous grammar, there exists at most one parse tree anyway, and the parse tree is build up by the top-down parsing process. However: **that's not true!**

But: if all information is available, what is missing?

We illustrate that with the following example, where a derivation will show a situation where even taking into account all information is not enough to make the right decision. A derivation like the one from the example is sometimes called an **oracular derivation.** The illustration is not in itself an *explanation* what's missing. With all information available, there's nothing really missing, it just illustrates that *deterministic* top-down parsers cannot recognize all unambiguous context-free grammars. In particular, being deterministic is a restriction. The word "oracular" hints at that: there are situations where all information is not good enough and to complete a parse successfully, the parser must currectly **guess** the right production, resp. getting help from an oracle to divinate the right decision. If one would tolerate non-determinism in such parsing devices, one can parse such examples, since by definition, a word is recognized by a parser machine, if there *exists* a successful parse. In other words, non-deterministic (and thus unpractical) top-down parsers are strictly more expressive than deterministic ones. This aspect of push-down automata and context-free languages is in constrast with finite-state automata and regular languages, where determinism is no restriction.

Example 4.2.4 (Oracular derivation). Consider the following grammar, another variation on expressions and terms, it's basically a slightly rewritten version of a expression grammar

discussed in the chapter about grammars.

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

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

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

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, and the down-arrow $\downarrow$ (used just here in this example) indicates where the "head" of the parser is currently positioned. By convention, the symbol to the right of $\downarrow$ is the one processed next.

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: for some grammars, not even perfect foresight (= look-ahead) plus perfect recall allows the top-down parser to make the right decision!

The parser process could use all look-ahead there is, namely until the very end of the word and could have remembered all the letters of the word processed so far. 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 **top-down** parsing.[4]

[4] 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 here? The reason why the parser could not make a uniform decision in this example (comparing line 7 and 8) comes from the fact that these two particular lines are connected by the step $\Rightarrow_4$, which corresponds to the production

$$term \to term * factor \ .$$

There, the derivation step replaces the non-terminal *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*

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, another lesson from the previous example is:

> left-recursion destroys top-down parseability.

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

Side remark 4.2.5 (Nit-picking about silly grammars)**.** As side remark, to split a few hairs: 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 agreed that in general do not consider such stupid grammars (without bothering to define exactly what constitutes a non-stupid grammar and what we consider as 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 (at least when doing top-down).

But then, what can be done? We simply put our foot down and rule out grammars where the decision cannot be done. Let's assume we are working with a left-most derivation (which is what top-down parsing resp. LL-parsing does) and assume that 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:

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

Assuming that the grammar is 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 for top-down parsing: 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 practice, 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.

Having defined our expectations for 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 determine whether or not a grammar meets that requirement. So far it's a wish rather than a criterion to apply to a grammar to see whether it works or not.

In the following Section 4.3, we introduce concepts that allow to exactly that: **determine** whether a given grammar is LL(k) or not. Whether a grammar is LL(k) is 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 decide the analogous question: is a grammar bottom-up parseable with a fixed look-ahead.

4.3 First- and follow-sets

We had a general look of what a look-ahead is, and how it helps in top-down parsing (actually and not surprisingly with bottom-up parsing as well). 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 considerations leading to a useful criterion for top-down parsing (and later also bottom-up parsing) without backtracking will involve the definitions of two related concepts of *first-sets* and *follow-sets*.

The definitions, as mentioned, will help to figure out if a grammar is top-down parseable. With one look-ahead only, such a grammar will then be called an LL(1) grammar (for top-down). 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. As mentioned, the first- and follow-set definition will *also* be used when discussing *bottom-up* parsing later (which will be called LR-parser).

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 symbol X is the set of terminal symbols can appear at the **start** of strings *derived from* X.
- 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.

The calculation of those sets is mostly straightforward, there will only one slight complication to watch out for, on connection with empty words $\boldsymbol{\epsilon}$ and so-called *nullable* symbols.

Definition 4.3.1 (Nullable)**.** Given a grammar G. A non-terminal $A \in \Sigma_N$ is *nullable*, if $A \Rightarrow^* \epsilon$.

4.3.1 First-sets (only solo)

The first-sets are conceptually simpler than the follow-sets, so let's start with those. The following is a straightforward formalization of the informal description from above.

Definition 4.3.2 (First set)**.** Given a grammar G and a symbol X. The *first-set* of X, written $First_G(X)$ is defined as

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

Note that for terminal symbols, say $\mathbf{a} \in \Sigma_T$, $First_G(a) = \{a' \mid a \Rightarrow_G^* a'\} = \{a\}$ and there is only one $\Rightarrow_G^*$-derivation, the empty one.

In the following, if the grammar G clear from the context, we write $\Rightarrow^*$ for $\Rightarrow_G^*$ and *First* for $First_G$; analogously later for the follow-sets.

The definition, unfortunately is not immediately to calculate the first sets, as it talks about arbitrary derivations, of which there are in general *infinitely* many. What we need is a calculation of the first-sets not based on derivations, but a calculation based on a grammar itself. It can be done quite easily by looking at a grammar

What we need is a calculation of the first-sets not based on derivations, but a calculation based on a **grammar** itself. It can be done quite easily, and we try shed light on the problem with the help of an example. The following example does not feature nullable symbols, as this is just a minor complication, adding "special cases", without changing the core of the problem.

Example 4.3.3 (Expression grammar). Let's start with the expression grammar from Example 4.2.4 on page 10 (see equation (4.4)). Just by looking at the right hand sides of two productions for *factor*

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

it's easy to figure out that the first-set for *factor* is the two-element set { **(**, **number** }.

Actually even "more immediate" is that the first-sets for the terminal symbols, for instance **(** resp. **number**, are the sets { **(** } resp. { **number** }. The treatment of terminals, where the first set is "most immediate" will serve as base case in a recursive characterization (resp. as initialization for an iterative algorithm).

Now we proceed to look at *term* , which has two productions:

$$term \rightarrow term * factor \mid factor$$

As we have convinced ourselves that the first set of *factor* is the set { **(**, **number** }, also the first-set for *term* contains those 2 terminals. And similarly, one can conclude that that this is also the same set for *exp* (because of the production $exp \rightarrow term$). So in summary, it's not hard to figure out that all non-terminals have the same first-set[5] (the first sets for the terminals **(**, **)**, **number** etc. are not shown in this small table):

	First
exp	{ **(**, **number** }
term	{ **(**, **number** }
factor	{ **(**, **number** }

□

The example was simple enough to determine the first-sets by just looking at the grammar. Actually, when thinking about how to solve the problem algorithmically, the grammar is *too* simple, and that's not just because it does not feature ϵ (which would be a minor complication). The real aspect why the grammar over-simplistic is something else.

What the example does *not* show is that calculating the first-set in general may involve **circular dependencies** of the different sets. Basically, the example presented the problem in a way that one starts by determining the first-set for *factor* first (resp. for the terminal symbols and, based on that, the first-set for *factor*. Afterwards, the first-set for *term* and then the first-set for *exp*, and then one is done. This layered calculation reflects the fact that the first-set of *exp* depends on the first-set of *term*, which in turn depends on the first-set of *factor*, which only depends on terminals. Note that the first-set of *factor* does not depend on the first-set of *exp*, despite the fact that the in the grammar, one rule for *factor* mentions *exp* on its right-hand side. One could illustrate this way of determining the first-sets in the examply by the graph of Figure 4.5. We could call graphs like that a form of dependency graphs. We don't want to introduce such representations formally here, we just want to illustrate the problems of first-set calculation in a graphical form.

As said, the previous example is deceptively simply in giving the impression that the first-sets can be calculated in this "staged" manner, like "bottom up" in the graph of Figure 4.5. That's not always the case. So we slightly modify the expression grammar to drive home that point.

[5] That should actually not be a surprise: the expression-term-factors cascade was a formulation to make a simple expression grammar unambiguous, by fix precendences and associatvities) and were *exp*, *term*, and *factor* all represent "expression" only at different levels of precedence.

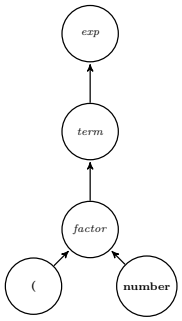


Figure 4.5: First-set dependencies from the grammar of Example 4.3.3

Example 4.3.4. The following is a minor variation of the expression grammar from Example 4.3.3 (or from Example 4.2.4)

$$\begin{array}{rcl} \mathit{exp} & \to & -\,\mathit{exp} \mid \mathit{exp} + \mathit{term} \mid \mathit{exp} - \mathit{term} \mid \mathit{term} \\ \mathit{term} & \to & \mathit{term} * \mathit{factor} \mid \mathit{factor} \\ \mathit{factor} & \to & (\,\mathit{exp}\,) \mid \mathbf{number} \mid \mathit{exp} \end{array} \tag{4.8}$$

We add two additional productions (the first and the last one here), not to make the grammar more useful in practice, but to illustrate the calculation of first sets on a simple though artificial example.[6]

We can illustrat the dependencies of the first sets in an analogous way than before, see Figure 4.6. Now there is a **cycle** of dependencies: not only does the first-set for expressions depends on the first-set for terms and the first set for terms depends on that for factors, but also, and that is new, the first set of *factor* depends on the first set of *exp*, closing the cycle. The figure also shows that the first-set of *exp* depends "on itself" and the same for *term*. Those small self-loops should actually have been present also in the previous Figure 4.5. But we left them out and glossed over that to keep the line of argument simple (cheating a little thereby).

With the new, more complex grammar, it's still straightforward to determine the first sets by just looking at it. The following table summarizes the results for the non-terminals:

	First
exp	{ (, **number**, − }
term	{ (, **number**, − }
factor	{ (, **number**, − }

Table 4.2: First-set for the non-terminals

But how to **algorithmically** approach the problem and to calculate the first sets? In face of the cyclic nature of the problem, more precisely the cyclic dependency of the first-set

[6] You may remember: the expression grammar from Examples 4.2.4 resp. 4.3.3 (without those two extra productions) used terminals *exp*, *term*, and *factor* as a way do avoid ambiguity by fixing operator precedence and (left) associativy. The additional two rules make the grammar ambiguous again, thus the split into expressions, terms and factors makes no real sense here. But the purpose of the grammar is to discuss the first-sets, not ambiguity or precedence, etc.

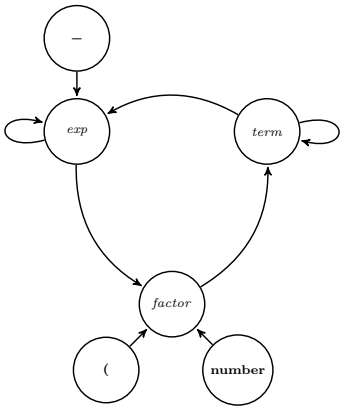


Figure 4.6: First-set dependencies

solutions for the different symbols, there is no starting point or natural order in which to proceed. That's unlike in the graph from Figure 4.5 where we could proceed with the calculation of the results from bottom to top.[7] For the cyclic dependencies , one could calculate the result as follows:

```
F_factor := { "(",  } ∪ { "number" }
F_term   :=  F_factor
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp // update with new info

F_term   :=  F_factor
F_expr   :=  { "−" } ∪ F_term
```

The first three lines set the three variables `F_factor`, `F_term`, and `F_exp` to sets of terminals. For the simpler grammar, i.e., the non-cyclic graph from earlier (Figure 4.5), this propagation of information would be all we need. Now, that the result for factors depends additionally on the result for expressions (i.e. the value of `F_factor` depends on the value of `F_exp`), the original value of `F_factor`, set in the first line, needs to be improved and **updated** with the information of the first-set information for `F_exp` (which includes −). In this small example `F_factor` is assigned to **twice.**

In this way, one can arrange the calculation of the first-sets for this example, but it's not a real algorithm yet. But it shows a crucial aspect of an algorithmic treatment: arriving at a solution collects the information gradually, and requires sometimes to **update** previously calculated **approximations** of a previosly calculated estimation. In the simple example, and with the chosen order, `F_factor` had to be updated twice, but if we started at a different point in the cycle of dependencies, an analogous improvement of a previosuly calcuated estimation would have happened elsewhere. For more complex grammars (maybe with multiple cyclic dependencies), it can lead to situations, where particular estimations need **multiple times.**

How often one needs to update the current estimations depends on the grammar (and on the order in which one treats the dependencies and updates). Consequently, the algorithm needs to know when it's enough, i.e., when the approximations are not longer approximate, but reflect the correct first-sets. In the above simple example, we simply stopped after

[7] If we ignore the fact that we have left out the "self-dependencies" which are not shown in the graph.

having updated `F_factor` the second time. But for an algorithm, we need a general **termation criterion**, to end the approximation loop or to end a corresponding recursive formulation. □

Next we spell out which conditions between the first sets of the different symbols of a grammar, similar to the informal treatment in the previous example, where for instance the first set of *factor* most *contain* the first-set of *exp* due to the production $factor \to exp$ from the grammar.

Before we do that, we point out which role **nullability** plays for that. That's easy enough. For a production $A \to X\beta$, where the right-hand side starts with symbol X, all elements from $First(X)$ must be contained in $First(A)$. In a situation like $A \to \alpha X\beta$, we need to take $First(X)$ into account for $First(A)$ nonetheless, **if α is nullable.** As a special case, that covers that $\alpha = \boldsymbol{\epsilon}$. This leads to the following formulation:

Definition 4.3.5 (First-set constraints)**.** Given a grammar G, The *first-set* of a symbol X, written $First(X)$, satisfies the following conditions:

1. If $X \in \Sigma_T$, then $First(X) \supseteq \{X\}$.
2. If $A \in \Sigma_N$: For each production, if $A \to \alpha X\beta$ where α is nullable, then $First(A) \supseteq First(X)$.

This is a set of constraints, specifying conditions that a solution must have (the solution being first-sets for each symbol, which play the role of variables for which a satisfying solution needs to be found). Constraint solving is a broad field in itself (and many problems in compilers can be phrased a constraints or also as constraints with the requirement of finding an optimal or at least good solution). Techniques for contraint solving depend on the form of constraints and the domain over which one tries to look for solutions (here sets of symbols): special cases allows for for specialized solving techniques.

For us it's good enough to know that our problem can be seen a some (actually simple) form of constraints.

As discussed or sketched in connection with the last Example 4.3.4, an approach to calculate the first-sets involves updating current estimations, enlarging previously obtained approximations. One open issue was the **termination criterion**, i.e., when to stop the updating of the current estimation. The answer is: one stops when the all constraints are satified, in other words:

terminate when the constraint system is solved!

So an algorithm must, by repeatedly going through some loop, increase current estimations of the first sets by enlarging them, until the constraint system is solved. There's just one (maybe less obvious) open issue, not really addressed in the example, namely where to start? What should be the **initial values** for the first-sets, i.e., the program variables used to store the first-set information?

Initially, without exploring the rules of the grammar, there's *no* information available, and the first-sets for all non-terminals are set to be empty. For each terminal symbol a, the initial first-set is $\{ a \}$. Since that information is never updated, it's not only the initial

information, but at the same time also the final one and one which directly corresponds to "base" case 1 from the constraints of Definition 4.3.5.

A an algorithm based on the presented ideas is shown in Listing 4.1.

```
for all a ∈ Σ_T do First[a] := {a};
for all A ∈ Σ_N do First[A] := { };
end;

while First[A] ⊂ First [X] for some production A → αXβ where α is nullable
  do
      First[A] := First[A] ∪ First[X]
  end;
end
```

Listing 4.1: First sets

As data structure, we assume an arrow `First`, indexed by the symbols of the grammar, and initialized in the first two lines as described. The main body updates and increased the information for non-terminals, as long as there are productions and combinations of symbols where the conditional constraint from case 2 of Definition 4.3.5 is violated. Upon termination, the negation of the loop condition hold, which means, when terminating the array `First` contains a solution to the constraint system, and we have calculated the first-sets. Another (but equivalent way) to perceive when to stop is the following: starting from minimal estimations of the first sets, the loop body increase locally the informating for some non-terminal A. Since the update is done when $First(A) \subset First(X)$ (and where there is a production $A \to \alpha X \beta$), the assignment *properly increases* the first-set for that non-terminal (as opposed to have no effect). So each iteration properly increases the information about first sets and the process stops, in that doing updates $First[A] := First[A] \cup First[X]$ (under the given side conditions) has no effect. Basically, one iterates and increases the amount of information, until continuing that way brings no more improvements (which is the point one has solved the constraints ...).

Side remark 4.3.6 (Constraint solving)**.** The (non-deterministic) algorithm basically describes a method for constraint solving for constraints of a particular form. Other kinds of constraints would require different (often more complex) constraint solving techniques. We won't dig deeper like explaining under which circumstances and for which kind of constraints the shown approach works and for which not. It will at least also work for the follow-set calculation. Actually it is an approach that can be used for many problems inside a compiler. In this lecture we will encounter *live variable analysis* that falls into this category, It is a problem that does not belong to parsing, but to later stages of a compiler, and is an example of a general class of analyses known as *data-flow analyses.* □

4.3.2 Follow-set (only solo)

Next we discuss the follow-sets, starting off by giving a more concise definition. Compared to the informal definition from page 14, a small detail is added, the use of some additional symbol **$** to represent the "end-of-parse". Files are often terminated by an end-of-file marker (EOF); here we represent the end of the words (a parser in practice would handle a token stream) my a special symbol **$**, which is treated as an extra terminal symbol.

Definition 4.3.7 (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

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

As we did for the first-sets, the following is contraint formulation, i.e. giving conditions that the follows sets for non-terminal symbols of a grammar have to satisfy.

Definition 4.3.8 (Follow set)**.** Given a grammar G and nonterminal A. The *Follow-set* of A, written $Follow(A)$ has to satisfy:

1. If A is the start symbol, then $Follow(A)$ contains \$.
2. If there is a production $B \to \alpha A \beta$, then $Follow(A) \subseteq First(\beta)$.
3. If there is a production $B \to \alpha A \beta$ such that β is nullable, then $Follow(A) \subseteq Follow(B)$.

The "base" case stipulates that the start symbol is followed by \$ (case 1). For an A occurring on the right-hand side of a production with some β to its right, what follows A depends on the first set of β (case 2). Case 3 covers the case where β to the right of A is *nullable*. In that case, since A can appear at the end of words derived from B, the follow-set of A contains the follow-set of B from the left-hand side.

```
initialize(Follow);

while
  there exists a production B → αAβ s.t.
      (                          Follow[A] ⊂ First[β])
         or
      (β nullable     and Follow[A] ⊂ Follow[B])
do
    Follow[A] := Follow[A] ∪ First[β])
    if β nullable, Follow[A] := Follow[A] ∪ Follow[B])
end
```

Listing 4.2: Follow sets (polish that formulation)

4.4 Massaging grammars

We discussed first- and follow-sets as "tools" to capture relevant aspects 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 helps 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 parsing methods. 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.

That could be generalized for 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 about longer words, rather than one symbol. In practice, one is often content with $k = 1$, so we don't bother about generalizing the setting.

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 discuss, how to massage such troublesome grammars in a way that gets 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 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 we mentioned it already: top-down parsing does not work with left-recursive grammars.[8]

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).

At 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

[8] Remember also the discussion around "oracular" derivations, which was about grammars where no amount of look-ahead would disambiguate a situation in a top-down parse, and which involved left-recursion.

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 trying to turn 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:

$$\begin{array}{rcl} \mathit{if\text{-}stmt} & \to & \textbf{if} \ (\ \mathit{exp}\ )\ \mathit{stmt}\ \textbf{end} \\ & \mid & \textbf{if} \ (\ \mathit{exp}\ )\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt}\ \textbf{end} \ . \end{array}$$

□

Next an expression grammar we have seen before in the chapter about grammars.

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 formulated equivalently as follows:

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{term}\ \mathit{exp}' \\ \mathit{exp}' & \to & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\ \mathit{addop} & \to & + \mid - \\ \mathit{term} & \to & \mathit{factor}\ \mathit{term}' \\ \mathit{term}' & \to & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\ \mathit{mulop} & \to & * \\ \mathit{factor} & \to & (\ \mathit{exp}\ ) \mid \textbf{number} \end{array} \tag{4.12}$$

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 ϵ-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. □

4.4.1 Left-recursion removal

Next we present a transformation 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) ϵ-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, resp. for a special case of recursion, immediate recursion. The following transformation removes **immediate** left-recursion on A in the presence of a second production involving A, which is assumed not to exhibit left recursion.

$$A \rightarrow A\alpha \mid \beta \qquad (4.13) \qquad\qquad A \rightarrow \beta A' \qquad (4.14)$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

The transformation is easy enough, introducing a new non-terminal, which uses a right-recursive procedure instead. The effect of the transformation on the shape of corresponding parse trees is illustrated in Figure 4.7.

(a) before (b) after

Figure 4.7: Left-recursion removal

Both grammars generate the same context-free language, i.e., the same set of words over terminals.

The previous situation was simplified: It dealt with immediate left-recursion, and additionally there was only *one* production suffering from direct left-recursion for the considered non-terminal. 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:

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

before

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

after

Indirect recursion is a bit more tricky, and it uses the removal of direct left recursion as subroutine. See Definition 4.4.5 resp. Listing 4.3.

Definition 4.4.5 (Left-recursion removal: indirect recursion)**.** Left recursion is removed by the code from Listing 4.3 (which also involves removing direct recursion from Definition 4.4.4).

```
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 // j < i
    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 4.3: Removal of left recursion (indirect case)

To deal with indirect recursion, we need to look at productions $A_i \to A_j\beta$ as this could indicate indirect left recursion. This include the situation $i = j$, which of course is direct left recursion. In Listing 4.3, direct left recursion is handled at the end of the code, after the main part of the algorithm, which consists of two loops, going through combinations of A_i and A_j. The inner body of the two loops handles combinations of A_i and A_j with $j < i$ are handleBy "current rules" it's meant the rules at the current stage of the algorithm.

Let's illustrate the procedure with a small example.

Example 4.4.6 (Left recursion removal). Consider the following grammar, which contains direct and indirect left recursion:

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

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

The following three grammars are the "transformation stages" of the grammar. With only 2 symbols, there are 3 combinations of non-terminals to be treated: A_1, A_1, A_2, A_1, and A_2, A_2 (resp A, A, A, B, and B, B).

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

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 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

Common left factors are likewise 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 ($\neq \boldsymbol{\epsilon}$ of course). The grammar can easily enough be transformed, "factoring out" the common prefix α:

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

It's assumed 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 4.4.7 (Sequence of statements). The grammar on the left represents non-empty sequences of statements, with semicolon as separator.

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

The grammar on the right-hand side has the common prefix of the two right-hand sides factored out. □

Example 4.4.8 (Conditionals). The following grammar covers conditionals supporting a one-armed and a two-armed variant.

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

That can be rewritten as follows:

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if (}\ \textit{exp}\ \textbf{)}\ \textit{stmts}\ \textit{else-or-empty} \\ \textit{else-or-empty} & \rightarrow & \textbf{else}\ \textit{stmts} \mid \boldsymbol{\epsilon} \end{array} \tag{4.18}$$

As you may remember, that the last versions of the conditionals suffer from being ambiguous and it's known as 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.

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if (}\ \textit{exp}\ \textbf{)}\ \textit{stmts}\ \textbf{end} \\ & \mid & \textbf{if (}\ \textit{exp}\ \textbf{)}\ \textit{stmts}\ \textbf{else}\ \textit{stmts}\ \textbf{end} \end{array} \tag{4.19}$$

The one without a common left factor looks as follows:

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if (}\ \textit{exp}\ \textbf{)}\ \textit{stmts}\ \textit{else-or-end} \\ \textit{else-or-end} & \rightarrow & \textbf{else}\ \textit{stmts}\ \textbf{end} \mid \textbf{end} \end{array} \tag{4.20}$$

□

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 is doable in **one step**. The following artificial example illustrates that.

Example 4.4.9 (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.

$$A \rightarrow \mathbf{abc}B \mid \mathbf{ab}C \mid \mathbf{a}E \qquad \begin{array}{rcl} A & \rightarrow & \mathbf{ab}A' \mid \mathbf{a}E \\ A' & \rightarrow & \mathbf{c}B \mid C \end{array} \qquad \begin{array}{rcl} A & \rightarrow & \mathbf{a}A'' \\ A'' & \rightarrow & \mathbf{b}A' \mid E \\ A' & \rightarrow & \mathbf{c}B \mid C \end{array}$$

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 4.4.

```
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 4.4: Left factorization

The algorithm is pretty straightforward (despite all the indices). 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 realizations of LL(1) resp. top-down parsing here: *recursive descent* and *table-based* LL(1) parser. They are different realizations of the same principles.

To be 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.

To do top-down parsing with a look ahead (for grammars that are LL(1)-parseable) is straightforward. A look ahead of 1 is not much of a look-ahead anyway, it's just the next 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 that, makes a decision. There's the special situation when 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 one of the expression grammars involving terms and factors, for instance in Example 4.4.3. Let's focus for now on the non-terminal *factor*, which is covered by 2 productions:

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

When parsing an input expecting a factor next, the decision that needs 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 its actions trying to parse the rest as expression *exp* and, when that succeeds, parsing the rest as **)**. If we assumed 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 token 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*, and
> - 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 structure 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, an 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, let's say in a Java-like or C-like language. For concreteness sake, we sketch also implementation for the tokens, here contansts (with the keyword `final`), all would be arranged in classes, but that's not shown here, as it's a sketch.

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

Listing 4.5: Data types for the tokens

```
void factor () {
    switch (tok) {
    case LPAREN: eat(LPAREN);expr();eat(RPAREN);
    case NUMBER: eat(NUMBER);
    }
}
```

Listing 4.6: 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 and others are 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 4.7: 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 4.8: Recursive descent for factors

□

We ignored tha 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 addressing abstract syntax trees, we first continue with top-down parsing itself. The situation is not always as immediate as in the previous 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.

But that's not all. It can be the case that the non-terminal in question is **nullable,** i.e., it has **no initial terminal.** 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 $\alpha_1 \neq \alpha_2$:*

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

And here the definition for the general case, for grammars that may contain nullable symbols.

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 don't 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 4.2.4). 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 situation of a production with direct left-recursion will diverge. Same for productions with indirect left recursion.

Let's revisit these situations with the LL(1)-criterion at hand. For instance the grammar for conditionals from equation (4.17) from Example 4.4.8 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

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

That could lead to a recursive-descend code as sketched in Listing 4.9.

```
procedure ifstmt()
  begin
    match ("if");
    match  ("(");
    exp();
    match (")");
    stmt();
    if    token = "else"
    then match ("else");
         stmt()
    end
  end;
```

Listing 4.9: 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 earlier. Now consider treatment of *exp*, i.e., the productions

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

It's 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)**-parseable (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:

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

The two non-terminal can be covered by the following procedures:

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

Listing 4.10: Procedure for *exp*

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

Listing 4.11: Procedure for *exp'*

While kind of massaging may help with making the grammar LL-parseable, it's problematic. 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 presentation, 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 does not work for top-down parsing.

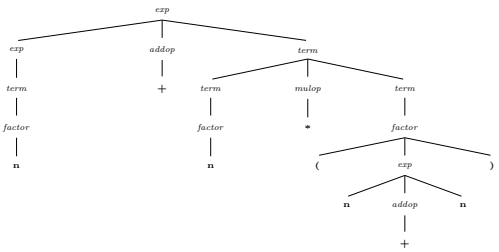


Figure 4.8: Parse tree for $1 + 2 * (3 + 4)$ for the left-recursive version of the grammar

Parse trees from the "flat" formulation of the grammar, without the cascade of terms and factors, look even a bit nicer (see Figure 4.9). But then the grammar is ambiguous, and thus likewise unsuitable for parsing.

□

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

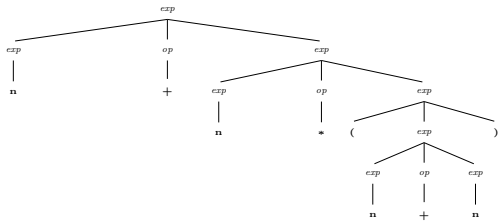
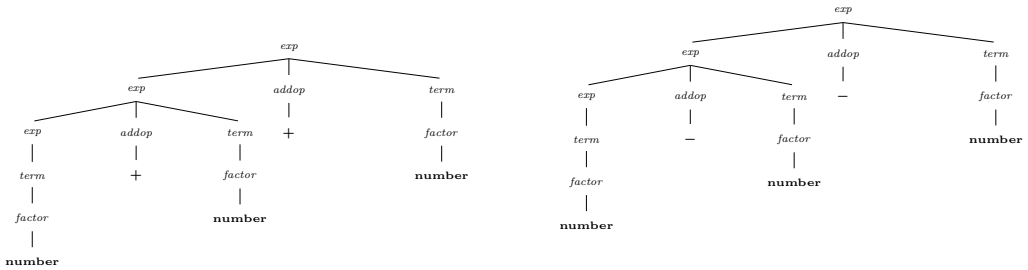


Figure 4.9: Parse tree for $1 + 2 * (3 + 4)$ for the "flat" grammar

was discussed 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 ambiguous. 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 earlier. 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 4.10.



(a) $3 + 4 + 5$ (b) $3 - 4 - 5$

Figure 4.10: 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 $3 - (4 - 5)$.

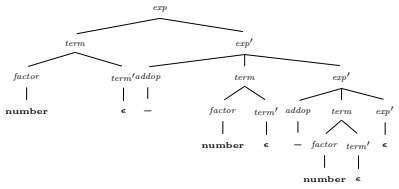


Figure 4.11: Factors and terms: right-associative parse tree for $3 - 4 - 5$

For parse trees, there's not much one can do about that: right-recursion corresponds to right-associativity in the parse tree. But one is not primarily interested in the parse-tree and 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, doing 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 discuss 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, the parsers returns 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 4.10b, evaluating the expression would be very easy. However, the parse-tree represents right-associativity, see Figure 4.11 and a straightforward, bottom-up evaluation will give the wrong result. The idea of the correct evaluation is shown in Figure 4.12.

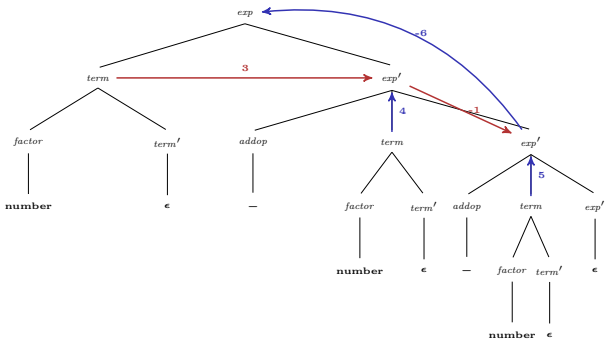


Figure 4.12: Repairing associativity (evaluation)

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 4.12 (at least 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 4.12: 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 *evaluates* them while doing that. In most cases in a f parser, one needs an AST, not a number, but the problem is analogous and the "accumulator"-pattern illustrated here in the evaluation setting can help out with ASTs.

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.[9] 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 its grammar for parsing are fixed, the parse trees are fixed, as well. 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 mentioned that earlier. It's also possible to have the AST being basically equivalent to the CST, in which case the AST becomes straightforward. At least if the grammar for parsing is not designed "weirdly" or having the wrong associativity.

For instance, looking at the parse tree of Figure 4.11, 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.

[9] 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`.

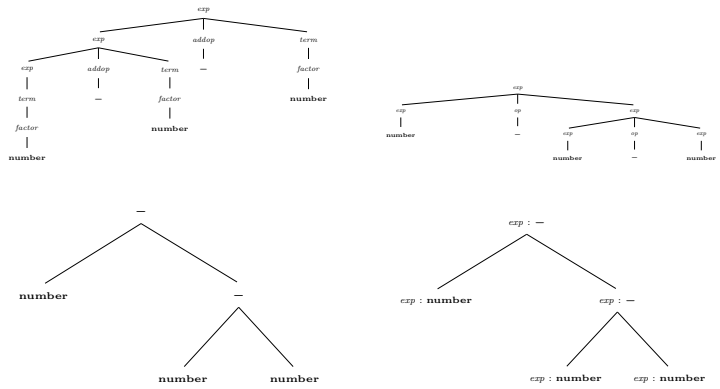


Figure 4.13: Different more or less abstract trees

But anyway, the figures are just illustrations how one can imagine that 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 sub-typing. 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.

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

Abstract syntax trees are *trees,* so here we use the BNF notation not as specification for the parser, but as description 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, with a field for token's *value* and a corresponding constructor

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 -> "*"
}
```

The following shows how to build an abstract syntax tree for expressions, with the corresponding constructors. That's basically what the parser should do during its run:

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

Listing 4.13: AST for $3 - (4 - 5)$

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 the recipy 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. So 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.[10] 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 4.14: Evaluating expressions

[10] 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 4.15: 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 4.16: 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"
```

```
  then match "else";
       elseChild(temp) := stmt();
  else elseChild(temp) := nil;
  end
  return temp;
end
```

Listing 4.17: 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 in principle but as far as the implementation is concerned. Instead of doing 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 look slightly different and which will be an LR-table or one of its variants.

Now, using an *explicit stack*, 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, e.g., 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 presentation, 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) parseable (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**.

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* **$**), then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$

Table 4.3: 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 4.4: 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 4.4.8.

	First	*Follow*
stmt	**other**, **if**	**$**, **else**
if-stmt	**if**	**$**, **else**
else−part	**else**, $\boldsymbol{\epsilon}$	**$**, **else**
exp	**0**, **1**	**)**

Table 4.5: First- and follow sets

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 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 construct 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 4.6: 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, removing 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 version of 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*, as information 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 straightforward to formulate an algorithm based on that, see Listing 4.18.

```
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 → X1 X2 ... 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 4.18: LL(1) table-based algorithm

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

Table 4.7: Illustration of a run of the algo

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 $else{-}part \to \textbf{else}\ stmt$ and not the alternative $else{-}part \to \epsilon$. 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.

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

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 4.8: First- and follow-sets

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 4.9: LL(1) parse table

4.6 Error handling (no pensum)

The error handling section is not part of the pensum (it never was), and it will not be asked in the exam, written or otherwise. The slides are not presented in detail in class.

That does not mean that we don't expect some halfway decent error handling for the compiler in the oblig. 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 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 or speculate what the programmer had in mind or should have written instead of what actually has been coded. 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 recovery, 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 an 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, when control returns from method `exp()`, it fails to encounter a parenthesis **)**. That's of course a syntax error, and the parser could report

```
right parenthesis missing!
```

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

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

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. The error may be in the extra `c` or perhaps that the programmer had forgotten some operator between `b` and `c` or something else. The parser cannot know, but a better message could be `error in expression, or right parenthesis missing`.

4.7 Bottom-up parsing

4.7.1 Introduction

This 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-up parsers or LR-parsers are probably the most common and default parser technology, realized by tools like yacc (and CUP ...). Another name for that kind of parsing is **shift-reduce** parsing. While top-down parsing and recursive descent-parsers are conceptually straightforward, LR-parsing is more subtle.

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 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. LR(0)-parsers are quite weak and thus really useful (but not as ridiculously weak as LL(0)). Still we cover them 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 4.10 shows an overview over the covered techniques. 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; after all, standard languages don't use LR(0)? The answer is, the more expressive parsers SLR(1) and LALR(1) use the *same* number of states (LR(1) has significantly more).

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

Also for top-down parsers, we looked at table-based realizations, namely LL(1)-tables. Those represent a different angle on the idea of recursively descending the parse-tree. 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

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 many 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 4.10: LR or shift-reduce parsers

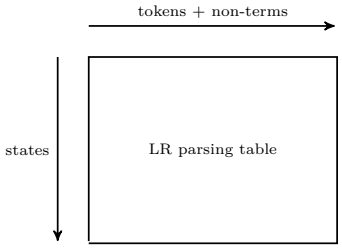


Figure 4.14: LR parsing table (schematic)

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 from Figure 4.14, we see that the lines corresponds to states and the columns to terminals (i.e., tokens) and non-terminals .

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.

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

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

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

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. Everything just smoother (and the construction of the LR-automaton is slightly more straighforward) if one obeys that convention. □

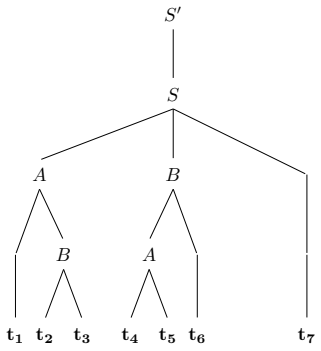


Figure 4.15: Parse tree for $\mathbf{t_1 \dots t_7}$

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:**

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

For such a derivation α is of the form $\alpha' \ a_1 a_2 \dots a_n$, where all the a_i's are terminals or tokens. The notion of **derivations** (left-most, right-most, or otherwise) is top-down: replacing non-terminals (parent nodes) by right-hand sides (children-nodes). But when building the tree bottom-up, the "replacement" goes the other way around: replacing right-hand sides of a production, the children nodes, but a left-hand side (a non-terminal and the parent node). The whole parsing process, building the tree, starts at the bottom of the tree. In other words, when doing the parse, the parser (implicitly) builds a **right-most derivation in reverse** and 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 4.16.

The slides version shows 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 generating or **deriving** words and which corresponds to the direction of how top-down parsers work.

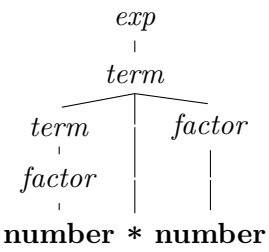


Figure 4.16: Parse tree

See Figure 4.17 for the processes of reduction resp. derivation. The underlined parts

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

(a) Reduction

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

(b) Right derivation

Figure 4.17

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 the reduction step. Note also that the "handle"-part β can be *empty*, i.e. β can be $\boldsymbol{\epsilon}$. 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.[11] In the parse-tree, A is a parent node to the children that correspond to β. Thus, 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 a **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!

The construction comes later, we first continue describing generally the parser behavior, without concretely showing the individual states or how they are constructed.

Example 4.7.4 (Run of a bottom-up parser). Let's revisit the artificial grammar from Example 4.7.1. Table 4.11 contains schematically the steps of a shift-reduce parser for the parse-tree of Figure 4.15.

The table contains two columns, the one on 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.

The parser starts with an empty stack, i.e., the stack consists of only **$**, and the full input $\mathbf{t}_1 \dots \mathbf{t}_7$, 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

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

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

Table 4.11: Schematic run (reduction from top to bottom)

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).

As mentioned, the state of the parser was not illustrated in the previous example and will neither be covered in the following schematic illustration. That the machine is in particular states during the run is central to make 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 4.14).

Since for the moment, we explain aspects of LR-parsing *without* referring to the state, we assume in the following, that the parse tree is somehow already known and we show runs where the machines simply does the right decisions that leads to the intended parse-tree. We did a similar assumption earlier, discussing **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 a particular production $A \to \beta$): remove the symbols from β from the stack (children) and **replace** it by A (parent).

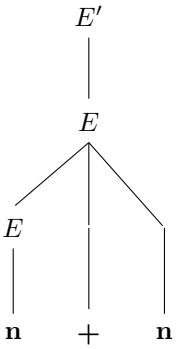
The replacement of the top of the stack β by A, moving up the tree, can also be interpreted as a combination of "pop + push": popping β followed by pushing A.

That's the general behavior, but later we need to address the question, how the parser makes its decisions: Should it do a shift or a reduce and if the latter, reduce with which 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 children-parent 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 4.7.5 (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 4.18

Figure 4.18 shows a parse tree for a simple expression like $1 + 2$ and a corresponding run of a shift-reduce parser. The run in Figure 4.18b 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.[12]

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.

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 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 way the states of the LR-parser will be constructed entails that 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 $\boldsymbol{\epsilon}$-transitions.

Example 4.7.6 (Example with $\boldsymbol{\epsilon}$-transitions: parentheses). Let's have a look at the following grammar for parentheses:

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

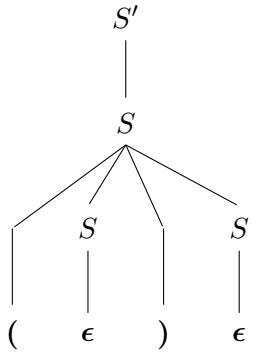
As a side remark: unlike the previous grammar, the grammar here has productions with *two* non-terminals on the right. Consequently, there is now a difference between left-most and right-most derivations (and mixed ones).

Note in particular the 2 reduce steps for the $\boldsymbol{\epsilon}$ 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:

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

□

[12] 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.



(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 4.19: parentheses

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 4.7.5 again.

Example 4.7.7 (Right-sentential forms and parser configurations). Figure 4.18b 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 4.18b, 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 reproduced here from equation (4.25)) "contains" only the reduce-steps from the run of Figure 4.18b, 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 $\mathbf{.}$ 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 4.20 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 ...

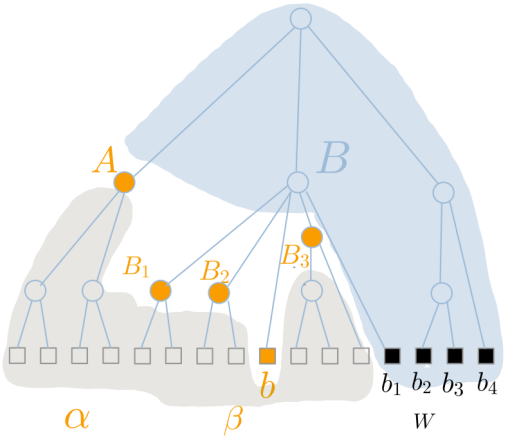


Figure 4.20: A typical situation during LR-parsing

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?

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?

It will be a subtle construction, and it is connected to a strange property of push-down automata. Push-down automata as general mechanism accept context-free languages. The language refers of course to the words accepted as *input*. This is done with the help of the stack, that growns and shrinks during the run of the machine. A language being accepted as input of such machines is context-free, of course. The stack which accepting a word from the input contains words of symbols as well, namely words over terminals and non-terminals (the input of course contains only terminals). That means, given a PDA, the possible stack contents forms a *language*. It's a strange and beautiful fact that such *stack languages* (aka pushdown store languages) are **regular**!

General fact: For push-down automata in general (and in particular for the LR-parser machine), the "stack-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**! The states of parser machines are the states of 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 trick will be to construct that automaton.

That the language of stack contents is regular is not a restriction, it's a subtle observation on what the stack can contain.

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 languages, 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 construction 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.

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 4.7.8 (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, β 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 4.7.9 (Items). For the grammars from Example 4.7.5 (parentheses) resp. from Example 4.7.6 (addition), the lists of items look as follows:

E'	$\to$	$\mathbf{.}E$	S'	$\to$	$\mathbf{.}S$
E'	$\to$	$E\mathbf{.}$	S'	$\to$	$S\mathbf{.}$
E	$\to$	$\mathbf{.}E + \mathbf{number}$	S	$\to$	$\mathbf{.(}\,S\,\mathbf{)}\,S$
E	$\to$	$E\mathbf{.} + \mathbf{number}$	S	$\to$	$\mathbf{(.}S\,\mathbf{)}\,S$
E	$\to$	$E + \mathbf{.number}$	S	$\to$	$\mathbf{(}\,S\mathbf{.)}\,S$
E	$\to$	$E + \mathbf{number.}$	S	$\to$	$\mathbf{(}\,S\,\mathbf{)}.S$
E	$\to$	$\mathbf{.number}$	S	$\to$	$\mathbf{(}\,S\,\mathbf{)}\,S\mathbf{.}$
E	$\to$	$\mathbf{number.}$	S	$\to$	$\mathbf{.}$

Both grammars lead (coincidentally) to altogether 8 items. Note that the production $S \to \boldsymbol{\epsilon}$ from the second example gives $S \to \mathbf{.}$ as item, and not two items $S \to \boldsymbol{\epsilon}\mathbf{.}$ and $S \to \mathbf{.}\boldsymbol{\epsilon}$). 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 (understood as) operating 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

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

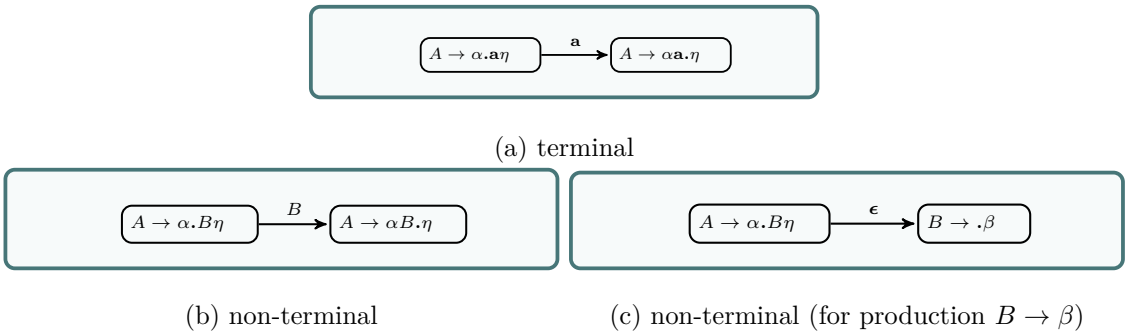
where $A \to \beta\gamma$ is a production of the given grammar. The marker **.** is meant to represent the split between the current stack content as the "past" and the expected future input, if we visualize the stack on the left and the rest of the input on the right. More precisely, the β is supposed to be the current top of the stack, and γ is the part of input 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.[13]

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 4.21.

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 Figure 4.21a. The marker is right in fron of a terminal symbol. Remember that the marker is intended to represent the position of the parser-machine. So in the start-state

[13] 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 4.21: Transitions of the LR(0)-NFA

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 4.21a and 4.21b, 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 4.21b and 4.21c. 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 4.21c 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 4.21b and 4.21c, and assume a item state of the form

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

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:

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

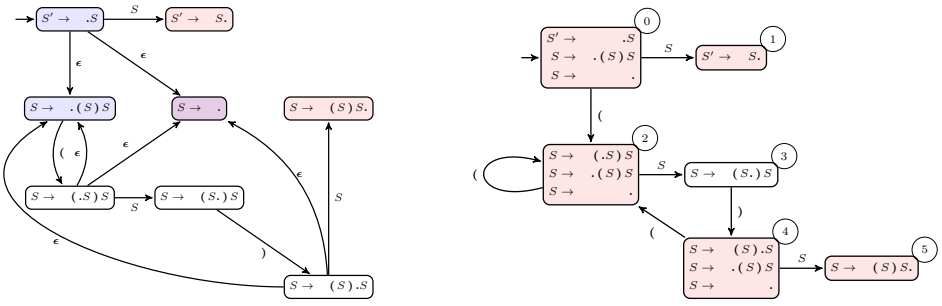
The B-step in the middle has no effect on the real input, it's just the internal book-keeping over the stack content, but the subsequent $\boldsymbol{a}$-transition corresponds to shift step. I.e. the behavior of equation (4.32) states that the `next token` on the input is $\boldsymbol{a}$. That's not the only possible behavior starting from the state of equation (4.31). Alternatively it could start with a $\boldsymbol{\epsilon}$-transition according to Figure 4.21c:

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

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** $B \to . \beta$ afterward, the **.** is in frontI say "assumed"

and moving to the item $.B$ places the **.** in front of the right-hand sides B of any production for the non-terminal B. Note that item-states $A \to \alpha . B \eta$ have at least kinds of outgoing transitions, one according Figure 4.21b and an ϵ-transition according to Figure 4.21c, according to each production of B. (4.31)

Example 4.7.10 (LR(0)-NFA for parentheses grammar). Let's do the construction for the parethesis from Example 4.7.6. The corresponding LR(0)-NFA is shown in Figure 4.22a. We also show the DFA in Figure 4.22b, though we are currently just discussing the NFA.



(a) NFA: parentheses

(b) DFA: parentheses

Figure 4.22: Parentheses

In the (NFA) figure, we use colors for illustration, 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 in Figure 4.22a: there is one initial item state per production of the grammar.[14] The initial items is where the ϵ-transisitions go into (as defined in Figure 4.21c), but *with exception* of the initial state (with S'-production). And there are no outgoing edges from those complete item states. Note the uniformity of the ϵ-transitions in the following sense. 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 (three ingoing ϵ transitions into the states with items $S \to \mathbf{.}S\,(\,S\,)$ and $S \to \mathbf{.}\,$). □

The initial state, as indicated in the figures of the example, is determined by the start-symbol of the grammar, resp. the corresponding initial item. Remember the assumption that the start symbol does not occur on the right-hand side of any production. That's often assured by augmenting the grammar by one *extra* start symbol say S'.

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

Side remark 4.7.11 (Nitpicking)**.** The nitpicking mind may comment: for the automaton from Figure 4.22a, we did not mention any accepting states. That would mean the automaton accepts not words.

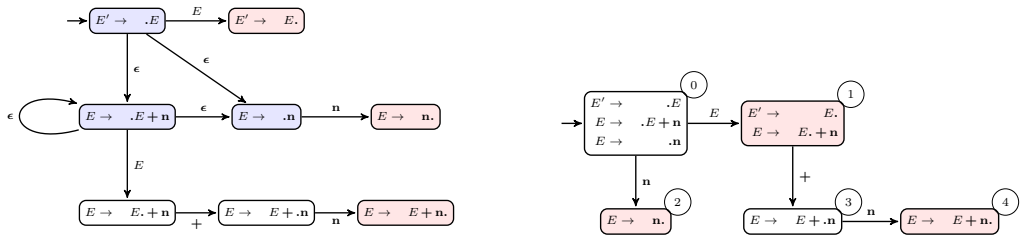
Well, the machine that accepts input is not the NFA (or DFA), but the parse machine, the stack automaton. The NFA here is a vehicle to construct and define the states of the stack automaton, "working" on the stack content. It's best to imagine that *all* states in the NFA (or DFA later) are *accepting*. 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. □

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 4.7.12 (LR(0)-NFA for the addition grammar)*.* Let's show the construction also for the addition grammar from Example 4.7.5. Figure 4.23a shows the corresponding LR(0)-NFA (and for comparison, we show alsp a deterministic machine in Figure 4.23b. □

[14] If the grammar is not "silly".



(a) NFA (b) DFA

Figure 4.23: Addition

The parser stack machine will not work with non-deterministic automata, but with a deterministic version. In that case, the stack content **determines** the state of the automaton.[15] In the examples with the LR(0)-NFAs, we also showed the corresponding DFAs.[16] 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 not 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 $q_1 \neq q_2$, 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 $A \to \alpha\boldsymbol{.}B\gamma$ is an item in a state, and the grammar has productions $B \to \beta_1 \mid \beta_2 \ldots$. Then **add** items $B \to \boldsymbol{.}\beta_1$, $B \to \boldsymbol{.}\beta_2 \ldots$ to the state. Continue that process, **until saturation**.

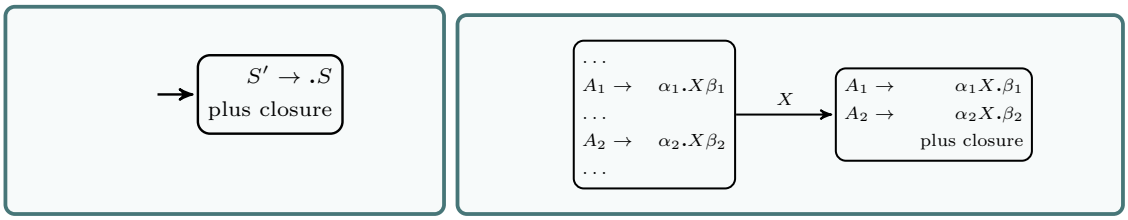
The initial state and the transitions are shown in Figures 4.24a and 4.24b. The symbol X represents a terminal or a non-terminal. Both are treated uniformely.

In Figure 4.24b, the picture is meant to indicate that *all* items of the form $A \to \alpha\boldsymbol{.}X\beta$ 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).

One can re-check the previous examples first doing the NFA, then the DFA), in particular Figures 4.22b and 4.23b: the outcome is the same.

[15] At the same time, if the construction does suffer from no so-called conflicts, the parsed input determines the overall state of the parser machine. We encountered conflicts for top-down parsers. A symptom for such conflicts were non-unique slots in the LL-parse table. That will be analogous for LR-parse tables and the tables are nothing else than a tabular representation of the DFAs. So char

[16] We don't show a *total* transition function, we left out the error state.



(a) initial state (b) transitions

Figure 4.24: LR(0)-DFA

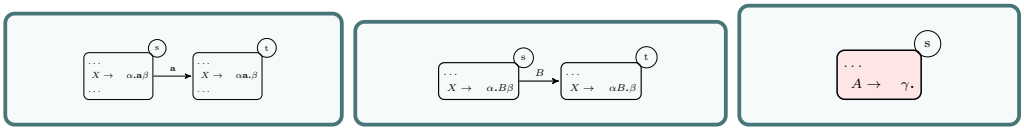
The LR(0) parser stack automaton

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, its 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 "operates" 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, i.e., the stack content **determines** state of the DFA. Of course, each prefix of the stack also determines uniquely a state and the state after scanning the complete stack content is also called **top state**. The top state is taken as state current overall **state of the LR(0) parser machine** and it is crucial when determining *reaction* of that machine.

Figure 4.25 shows three different situations allowing three different reactions. The "source" state labelled s is interpreted here in the discussion as *top-state*.



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

Figure 4.25: Parser machine transitions

The state of Figure 4.25a contains an item $X \to \alpha.\,\boldsymbol{a}\,\beta$ as source and, assuming that's the top-state, a **shift** is allowed. After eating $\boldsymbol{a}$ and pushing it to the stack, the machine is in the new (top-)-state t. What happens if the next symbol in the input would not be **a** but **b** and if the state s would not contain another transition labelled **b**? We are currently doing LR(0), without look-ahead. That means, the overall machine can do a reduce step

independent from the next letter. In case of an **a**, the next (top) state would be t, in case of **b** or another unexpected non-terminal, the shift-step would result in an error step.[17]

Transitions for non-terminals work similar. State s in Figure 4.25b contains an item $X \to \alpha.B\beta$ and when following the shown transition, the input is left unchanged. Those kind of transitions are also called **goto**-steps, and will show up under this name in the LR-parse tables later.

Figure 4.25c shows a state containing a **complete**-item. That is characteristic for **reduce**-steps. Since reduce-steps are not reflected as transitions in the LR(0)-automaton, the figure shows no transitions. Of course, doing a "reduce" is a step or transition of 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 4.25c, 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 γ by A on the stack. With A on top of the stack instead of γ, there will be in general a **new top state**! So, if a reduce happens, the parser engine *changes state*![18] 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 by outgoing arrows of completed items.

Example 4.7.13 (Example: LR parsing for addition). Let's revisit Example 4.7.5, which we used to illustrate 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 4.18b. In the meantime, we have constructed the LR(0)-DFA from Figure 4.23b, 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 4.18b, interesting in the run are stages 3 and 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 4.23b that's state 1. State 1 illustrates a **shift/reduce conflict.** Remember: reduce-steps are *not* represented in the LR(0)-automaton via *transitions* and only implicitly represented by *complete items*. So such conflicts are characterized by an **outgoing edge (labelled by a terminal symbol), starting at a state containing a complete item**. □

The point being made when looking at state 1 is the following: the state is a state containing a complete item. Besides that, there is an outgoing edge labelled with a terminal. 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 closer look at the situation and taking a look-ahead into account may defuse the conflict. Those

[17] If the DFA is complete, the situation "no **b** transition starting from s" would rather be a **b** transition to an error state in theory. In practice, some exception would be raised.

[18] At least in general. There may be situations, where the the parser happens to move from one state to the same state again.

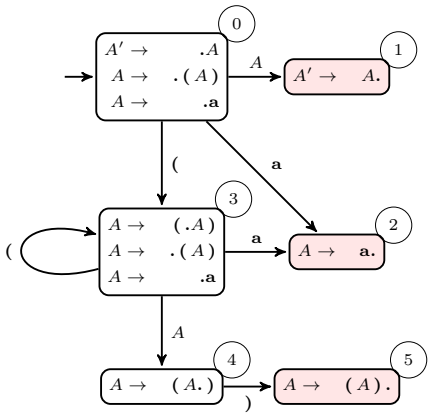
more fine-grainend considerations will lead to extensions of the plain LR(0)-parsing (like SLR(0), or LR(1) and LALR(1)).

Let's have a look at another example. Compared to an earlier Example 4.7.6 involving parentheses, te next example captures a more simplistic use of parentheses.

Example 4.7.14 (Simple parentheses). Consider the following simple grammar:

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

Figure 4.26a shows the LR(0)-DFA for the grammar, and Figure 4.26b possible reactions for each each (top)-state. For completeness sake, we show also the corresponding NFA in Figure 4.27. □



(a) LR(0)-DFA

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

(b) (Re-)action per state

Figure 4.26: Simple parentheses

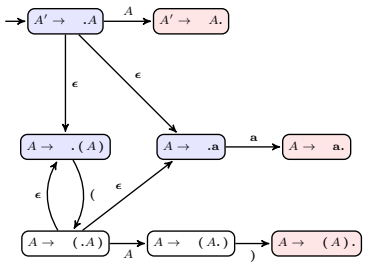


Figure 4.27: NFA for simple parentheses (bonus)

In the same way as for scanning automata, the reactions of the machine can also be represented in **tabular form**, here called **LR(0) parsing table**. Later there will be table variants for SLR(1), LALR(1), and LR(1), 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.

Example 4.7.15 (LR(0)-table for simple parentheses). The syntax for the simple parentheses have been defined in Example 4.7.14. The parse table from Table 4.12 is nothing else than the LR(0)-DFA from Figure 4.26a 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 4.12: LR(0) parsing table (cf. Figure 4.26a)

The shift-steps are listed 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.**

Figure 4.28a shows a run or parse of the machine. 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 is shown 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** (empty 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 4.28b. □

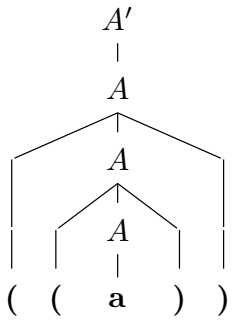
The LR-table shown contains **empty** slots. Another way to see it is that for the deterministic automata, like the one from Figure 4.26a is not complete, i.e., it has not explicit error state resp. we did not bother to show it. The DFA is constructed to define the legal stack content. Thus, hitting 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.

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(= reduce $A' \to A$)

(a) Run



(b) Parse tree

Figure 4.28: Simple parentheses

Example 4.7.16 (Simple parentheses: parsing of erroneous input). Let's pick up on the simple parenthesis syntax again from Example 4.7.14. Figure 4.29 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 4.29: Parse errors

Figure 4.30 summarizes the steps of an LR(0)-parser machine.

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

Example 4.7.17 (DFA (non-simple) parentheses again: is it LR(0)?). Let's revistit the non-simple parentheses from Example 4.7.6 and the grammar from equation (4.26). See in particular the LR(0)-DFA from Figure 4.22b. 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 4.7.18 (DFA addition again: is it LR(0)?). Let's revistit the addition-expressions from Example 4.7.5 and the grammar from equation (4.24). See in particular the LR(0)-DFA from Figure 4.23b.

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 4.30: Steps of the LR(0) parser machine

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.

To prepare for the following sections, we can consider whether it's possible to make the correct decision, given a concrete input. For Example 4.7.5, Figure 4.18a showed a parser run for **number** + **number** as input 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 input is parsed completely, i.e., the next symbol is **$**. Taking that one symbol **look-ahead** into account the parser can make the correct decision: shift in state 1 with + as next symbol, and reduce, with **$** 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 too weak to be useful in practice. 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. Of course, no amount of look-ahead can resolve all conflicts, there will always be grammars with shift-reduce conflicts, not matter the amount of look-ahead. 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 [2] (**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 conflicts when doing a parse with the LR(0)-DFA. SLR(1) is only moderately more powerful than LR(0) and for realistic languages, it's mostly still seen as too weak, in particular since with LALR(1), there's a better alternative available, which does not require more resources during parsing.

SLR-parsers are based on the same principles as LR(0), but the format of the parse tables is slightly different. The parser machine, however, is *still* based on **LR(0)-DFAs**. The idea

is 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 thew simplistic way how look-ahead is used in SLR. We illustrated LR 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 size of the parser table considerably. SLR(1), instead works with the smaller LR(0)-DFA, but tries do defuse some conflicts, if possible. I

There are reduce-reduce and shift-reduce conflicts and 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 4.31a). 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.[19]

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 both follow sets, 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 $A \to \alpha\bullet$ or with $B \to \beta\bullet$. 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 4.31b). The symptom for a shift-reduce conflict is a state in the LR(0)-DFA containing a complete item $A \to \alpha\bullet$ and a non-complete item $A \to \alpha_1 \bullet \boldsymbol{a}\, \alpha_2$. 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.

The LR(0)-conflicts from Figure 4.31 are **resolved in SRL(1)**, if for reduce-reduce, resp. for the shift-reduce situation, the following conditions hold respectively:

[19] You may reflect on why that is.

(a) Reduce-reduce

(b) Shift-reduce

Figure 4.31: LR(0) conflicts

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

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 4.31a, next symbol in the token input stream decides: if $\mathbf{t} \in Follow(A)$ then reduce using $A \to \alpha$, if $\mathbf{t} \in Follow(B)$ then reduce using $B \to \beta$. Similar for the shift-reduce situation as in Figure 4.31b: If $\mathbf{t} \in Follow(A)$ then *reduce* using $A \to \alpha$, if $\mathbf{t} \in \{\mathbf{b_1}, \mathbf{b_2}, \ldots\}$, then *shift*.

Example 4.7.19 (SLR and SLR-table: addition (one more time)). Let's revisit the addition example one more time. As hinted at already in Example 4.7.5 and as visible in the LR(0)-DFA from Figure 4.23b, 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 1 is $E' \to E.$. So to apply the SLR-conditions, we need to calculate the follow set for E'. The follow-set for the (extra) start symbol is $Follow(E') = \{\$\}$.[20] 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 on rule $E' \to E$ for **$** (which corresponds to accept, in case the stack is empty).

Table 4.13 shows the **SLR(1) table** that corresponds to the LR(0)-DFA. Conceptually, it's in the same format than the LR(0)-table. However, in an ok LR(0) table, i.e., one without conflicts, and it is arranged a bit different. Without LR(0)-conflict, the reaction for a given state is either a reduce (with one particular rule) (or an error) or else qa shift (though different symbols may lead to different successor states) (or an error).

Now, with SLR(1), there can be states where both shifts or reduces are possible, 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 (the one that correspond to acceptance). What would also be possible, though not relevant in this example are states or lines with *different* reduce steps, depending on the input. That would correspond to a automaton, where reduce-reduce conflicts are resolved by the follow set criteria we discussed (see equation (4.35)).

A parser-run is shown in Figure 4.32a resulting in a parse tree from Figure 4.32b. □

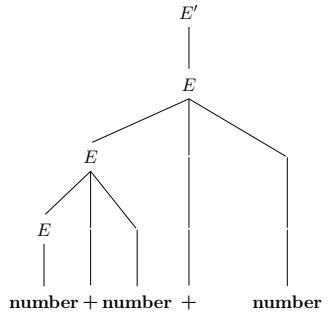
[20] As a side remark: we don't need here to bother about the algorithms for the follow set and the first set. Under the convention that the (extra) start symbol never occurs on the right-hand side of a production, it's immediately clear that **$** is the only symbol in its follow set.

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 4.13: SLR(1) parsing table (addition)

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 4.32: Addition grammar

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

Example 4.7.20 (SLR(1): parentheses again). Let's revisit the grammar for parentheses from equation (4.26) and Example 4.7.6. The LR(0)-DFA for the grammar has already been presented in Figure 4.22b.

The follow-set of the non-terminal S is

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

The SLR(1)-parsing table is shown in Figure 4.33a and a run of the parser machine in Figure 4.33b.

□

As seen in the examples, SLR(1) parsing tables look rather similar to the LR(0) ones. 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.

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)

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

(b) Parser run ("reduction")

Figure 4.33: Parentheses

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.[21]

Figure 4.34 shows the steps of a SRL(1)-parser machine. It's a minor variation of the steps for the LR(0) variant from Figure 4.30.

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

     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 4.34: SLR(1) algorithm

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**

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

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 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 can live 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 conditionals, 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, conventionally preferring a production written earlier over alternatives written later in the grammar (like in a "first-match").

Generally, warnings 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, but rather an opportinity to get more insight what consequence reduce- and shift-steps have in shaping the parse tree (and thereby the AST).

Example 4.7.21 (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 (4.14)

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

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

	Follow
S'	{**$**}
S	{**$**, **else**}
I	{**$**, **else**}

Table 4.14: Follow sets

Figure 4.35 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).

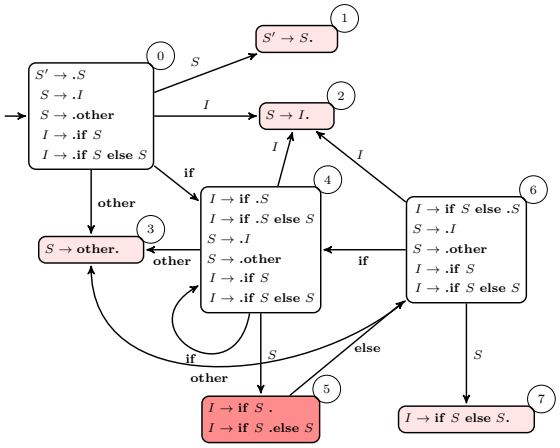


Figure 4.35: DFA of LR(0) items

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 4.15, *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 4.15: 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 4.36 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 4.37 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	$\$_0$**if**$_4$	**if other else other $**	shift: 4
3	$\$_0$**if**$_4$**if**$_4$	**other else other $**	shift: 3
4	$\$_0$**if**$_4$**if**$_4$**other**$_3$	**else other $**	reduce: 2
5	$\$_0$**if**$_4$**if**$_4 S_5$	**else other $**	shift 6
6	$\$_0$**if**$_4$**if**$_4 S_5$**else**$_6$	**other $**	shift: 3
7	$\$_0$**if**$_4$**if**$_4 S_5$**else**$_6$**other**$_3$	**$**	reduce: 2
8	$\$_0$**if**$_4$**if**$_4 S_5$**else**$_6 S_7$	**$**	reduce: 4
9	$\$_0$**if**$_4 I_2$	**$**	reduce: 1
10	$\$_0 S_1$	**$**	accept

Figure 4.36: Parser run: preference on shift

stage	parsing stack	input	action
1	$\$_0$	**if if other else other $**	shift: 4
2	$\$_0$**if**$_4$	**if other else other $**	shift: 4
3	$\$_0$**if**$_4$**if**$_4$	**other else other $**	shift: 3
4	$\$_0$**if**$_4$**if**$_4$**other**$_3$	**else other $**	reduce: 2
5	$\$_0$**if**$_4$**if**$_4 S_5$	**else other $**	reduce 3
6	$\$_0$**if**$_4 I_2$	**else other $**	reduce 1
7	$\$_0$**if**$_4 S_5$	**else other $**	shift 6
8	$\$_0$**if**$_4 S_5$**else**$_6$	**other $**	shift 3
9	$\$_0$**if**$_4 S_5$**else**$_6$**other**$_3$	**$**	reduce 2
10	$\$_0$**if**$_4 S_5$**else**$_6 S_7$	**$**	reduce 4
11	$\$_0 S_1$	**$**	accept

Figure 4.37: Parser run: preference on reduce

Finally, Figure 4.38 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 and more importantly, it should give a feeling how generally a shift-vs-reduce changes the structure of the parse-tree (and indirectly most probably 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.

(a) shift-precedence: conventional

(b) "wrong" tree

Figure 4.38: Parse trees for the "simple conditionals"

Using **ambiguous grammars** may have the advantages that they are often *simpler*. Of course, an ambiguous grammar 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 precedences. That is illustrated in the next example.

Example 4.7.22 (Precedences and associativity). Consider the following expression grammar:

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

The corresponding LR(0)-DFA is shown in Figure 4.39.

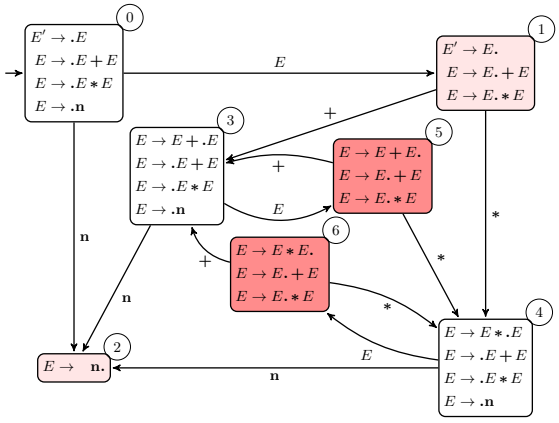


Figure 4.39: DFA for + and ×

Let's look at two states with conflicts. In **state 5**, the parser is in a state that contains $E + E$ on top as can be seen from the complete item it "contains". So that means a reduce 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 achieve **precedence** of multiplication over addition. Finally, for **$**, the machine must do a reduce, a shift is anyway not allowed.[22]

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 4.16.

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 4.16: 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 an ambiguous grammar to obtain the intended disambiguations. We know from earlier sections, that 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, just for comparison.

Example 4.7.23 (Unambiguous grammar). The grammar from equation (4.40) is a reformulation of the language specified by the ambigous grammar from equation (4.39) into an unambigous one. The table on the right-hand shows the follow-sets for the involved nonterminals (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 4.40 shows the corresponding LR(0)-DFA.

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,*

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

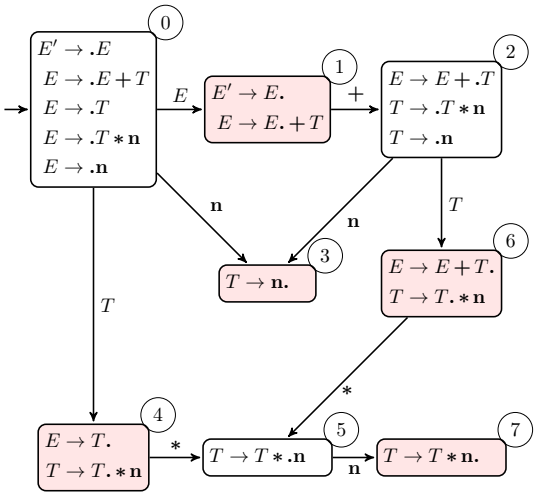


Figure 4.40: DFA for unambiguous grammar for + and ×

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 symbol $*$ 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 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) can be understood as stepping-stone towards LALR(1).

To get that out of the way. We 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 on 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). 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. One might 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.

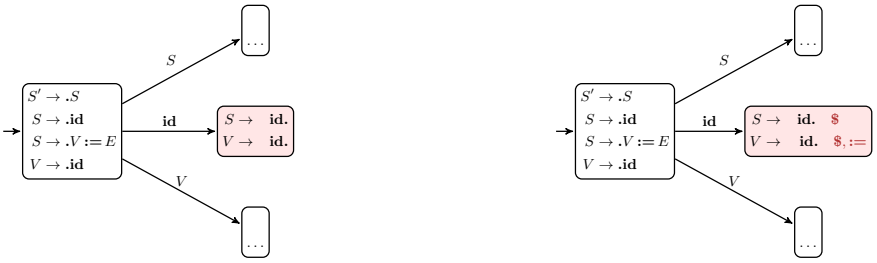
Example 4.7.24 (Assignment grammar fragment). Let's take the following grammar(s). The one on the left could be a fragment of a grammar for some programming language. For the construction, we focus on some simplified subset of that, shown in equation (4.41).

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

Table 4.17 shows the follow-sets for the non-terminals, and Figure 4.41 shows some portion of the LR(0)-DFA for the grammar.

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

Table 4.17: Follow sets



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

(b) Extra follow-set information

Figure 4.41: Transitions of the LR(0)-NFA

The reddish state in Figure 4.41a has an 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-parsing runs into trouble with rather simple common notations found in different programming languages. 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 problem in the larger of the two grammars, as we will see, concerns identifiers (resp. variables as left-hand side of an assignment or as a call expression). In the simplified grammar, there's no call-statement, but the problem is the same.

Figure 4.41b 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 4.42. The (sketch of the) automaton here looks pretty similar to the

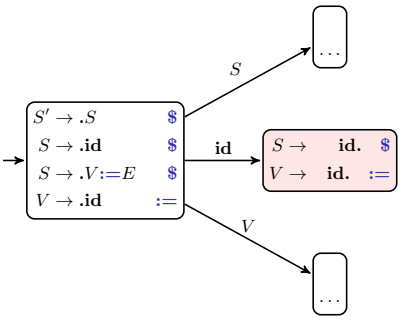


Figure 4.42: Look-ahead information as part of the items

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 $S \to \mathbf{id}.$ and $V \to \mathbf{id}.$, 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 $S \to .V \mathbf{:=} E, \mathbf{\$}$. 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.

> Now, when adding that item $V \to \mathbf{.id}$, 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 a :=**. That leads (in this situation) to the item of the form $[V \to \mathbf{.id}, \mathbf{:=}]$.

This information is more specific than the general follow-set of V, which contains both **:=** and **$**. Now, by recording that extra piece of information in the closure, the state **"remembers"** that the only thing at the current state that is allowed to follow the V is the **:=**. That will defuse the conflict: if we follow the **id**-edge, we end up in the state on the right-hand side. Such a transition does not touch the additional look-ahead information (here the **$** respthe **:=** symbol). See also the corresponding NFA-rule later. Thus, in the state at the right-hand side, the reduce-reduce conflict isappeared! □

So that's the core of LR(1) parser automata: adding precision in the states of the automaton already: Instead of using look-ahead as **afterthought** to an LR(0)-construction with the help of the follow sets for states containing complete items, it's more expressive by **with 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. This additional look-ahead information leads to a proliferation of states, enlarging the automaton.[23] The form of the new items is as follows

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

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

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

Example 4.7.25 (Assignments (LR(1))). For the assignment grammar from Example 4.7.24, Figure 4.43 shows the LR(1)-DFA. Part of it has been depicted already in Figure 4.42.

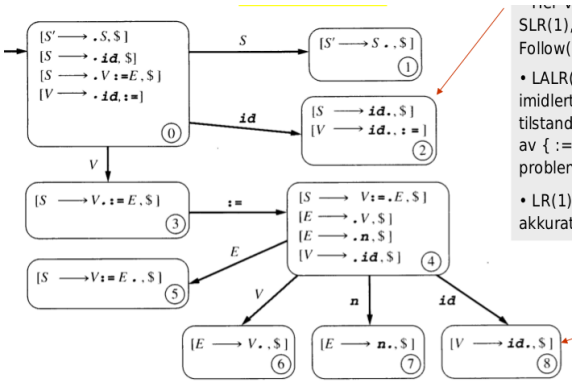


Figure 4.43: LR(1)-DFA

□

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 4.44. X represents a terminal or a non-terminal. For the case of the ϵ-transition in Figure 4.44b, the edge is added for all

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

(a) arbitrary symbol (b) ϵ-transition ($B \to \beta$ and $\mathbf{b} \in \mathit{First}(\gamma\mathbf{a})$)

Figure 4.44: LR(1) parser machine transitions

The construction from Figure 4.44b also contains the special case for $\gamma = \epsilon$, shown in Figure 4.45, adding a transition for all $B \to \beta$.

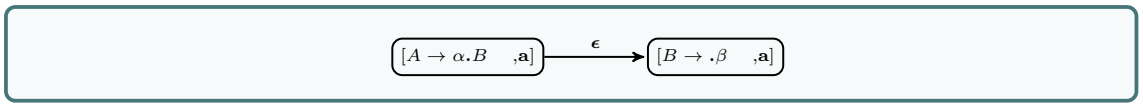


Figure 4.45: ϵ-transition, special case ($\gamma = \epsilon$)

In the resulting DFA from the previous example, 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

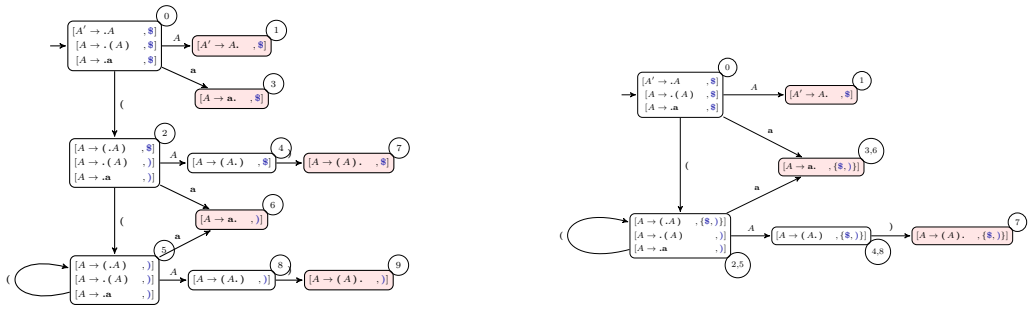
After canonical LR(1) parsing, let's have a look at LALR(1) parsing. The abbreviation stands for *look-ahead LR-parsing*, but the names does not say much. We hinted at core idea before: LALR(1) is shift-reduce parser and works analogous to the other variations. The states of the corresponding DFA come from taking the states of the LR(1)-DFA and "collapse" some of them, 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 4.7.14.

Example 4.7.26 (Simple parentheses). Let's look at the following grammar:

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

We have seen that grammar already in equation (4.34).[24]

Figure 4.46a shows the corresponding LR(1)-DFA with 10 states. 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 is called the **core** of such an item. The core of the LR(1) item is of course an LR(0) item. Then the LALR(1) automaton is built by **collapsing** LR(1) states with the **same core**. The result is shown in Figure 4.46b.



(a) LR(1) DFA

(b) LALR(1) DFA

Figure 4.46: LR(1) and LALR(1) DFAs

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, doing the collapse results in a deterministic automaton.

The states almost seem 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). □

[24] 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 illustrate the LALR(1) idea.

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) are 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.[25] 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.

	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 4.18: 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, and 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 simple 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, but 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 after encountering a first error. When doing recovery, one should avoid to report an avalanche of subsequent *spurious* errors, just "caused" by the first error. In general 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.

[25] 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.

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 seems 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 **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 clear when the parser stumbles on an error. It happens if there is no entry in the parser table covering the situation. It's less easy to determine 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* over 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 4.19 and a run as shown in Table 4.20.

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.

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

Table 4.19: Shift-reduce parse table

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

Table 4.20: Panic mode recovery run

In that situation, 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 handled, 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 4.22. It shows a looping run for a parser for expression, a relevant part of the parse Table is sketched in Table 4.21.

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

Table 4.21: Parse table fragment

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.

	parse stack	input	action
1	$\$_0$	**(n n) $**	
2	$\$_0(_6$	**n n) $**	
3	$\$_0(_6\mathbf{n}_5$	**n) $**	
4	$\$_0(_6 factor_4$	**n) $**	
6	$\$_0(_6 term_3$	**n) $**	
7	$\$_0(_6 exp_{10}$	**n) $**	panic!
8	$\$_0(_6 factor_4$	**n) $**	been there before: stage 4!

Table 4.22: Panic mode loop

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] Appel, A. W. (1998). *Modern Compiler Implementation in ML/Java/C*. Cambridge University Press.

[2] DeRemer, F. L. (1971). Simple LR(k) grammars. *Communications of the ACM*, 14(7).

[3] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

Index