Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

Chapter 4 Parsing

What is it about?

Learning Targets of this Chapter

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

Contents

4.1 Introduction to parsing

What's a parser generally doing

task of parser = syntax analysis

- input: stream of **tokens** from lexer
- output:
  - **abstract syntax tree**
  - or meaningful diagnosis of source of *syntax error*

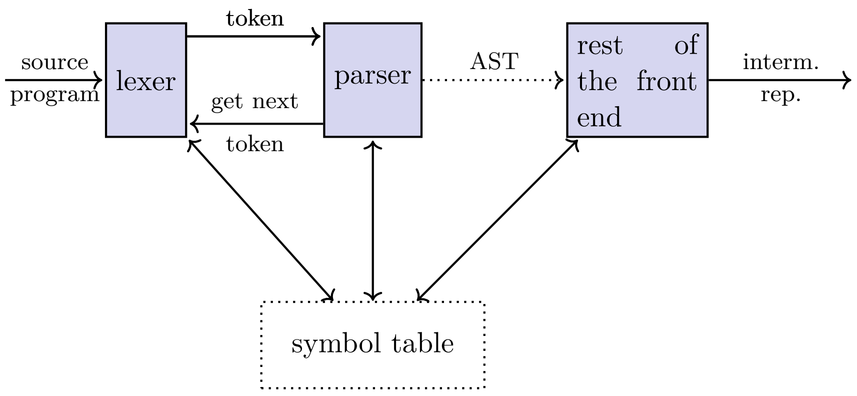
- the full "power" (i.e., expressiveness) of CFGs not used
- thus:
  - consider *restrictions* of CFGs, i.e., a specific subclass, and/or
  - *represented* in specific ways (no left-recursion, left-factored . . .)

Syntax errors (and other errors)

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. 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 potential locations of the error and potential reasons. In the most minimal way, the parser should inform the programmer where the parser tripped, i.e., telling how far, from left to right, it was able to proceed and informing where it stumbled: "parser error in line xxx/at character position yyy"). One typically has higher expectations for a real parser than just the line number, but that's the basics.

It may be noted that also the subsequent phase, the *semantic analysis*, which takes the abstract syntax tree as input, may report errors, which are then no longer syntax errors but 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 the file *again*.

Lexer, parser, and the rest



Top-down vs. bottom-up

- all parsers (together with lexers): *left-to-right*
- remember: parsers operate with *trees*
  - parse tree (concrete syntax tree): representing grammatical derivation
  - abstract syntax tree: data structure
- 2 fundamental classes
- while parser eats through the token stream, it grows, i.e., builds up (at least conceptually) the parse tree:

Bottom-up

Parse tree is being grown from the leaves to the root.

Top-down

Parse tree is being grown from the root to the leaves.

AST

Parsing restricted classes of CFGs

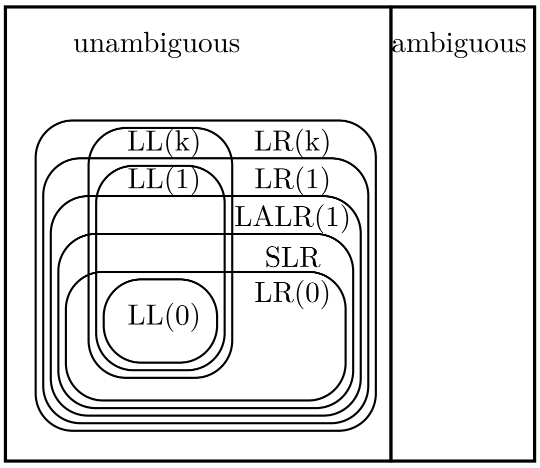
- parser: better be "efficient"
- full complexity of CFLs: not really needed in practice
- classification of CF languages vs. CF grammars, e.g.:
  - left-recursion-freedom: condition on a grammar
  - ambiguous language vs. ambiguous grammar
- classification of grammars ⇒ classification of *languages*
  - a CF language is (inherently) ambiguous, if there's no unambiguous grammar for it
  - a CF language is top-down parseable, if there exists a grammar that allows top-down parsing . . .
- in practice: classification of parser generating tools:
  - based on accepted notation for grammars: (BNF or some form of EBNF etc.)

Concerning the need (or the lack of need) for very expressive grammars, one should consider 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. And time in a compiler may be better spent elsewhere (optimization, semantical analysis).

Classes of CFG grammars/languages

- *maaaany* have been proposed & studied, including their relationships
- lecture concentrates on
  - top-down parsing, in particular
    - **LL(1)**
    - **recursive descent**
  - bottom-up parsing
    - **LR(1)**
    - **SLR**
    - **LALR(1)** (the class covered by yacc-style tools)
- grammars typically written in *pure* BNF

Relationship of some grammar (not language) classes



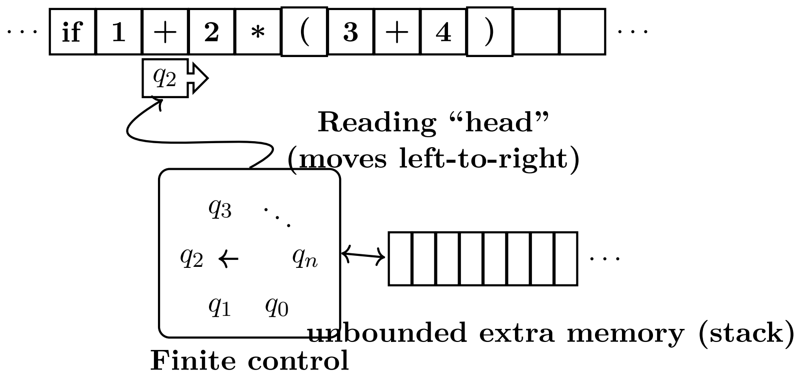
taken from [?]

4.2 Top-down parsing

General task (once more)

- Given: a CFG (but appropriately restricted)
- Goal: "systematic method" s.t.
  1. for every given word w: check syntactic correctness
  2. [build AST/representation of the parse tree as side effect]
  3. [do reasonable error handling]

Schematic view on "parser machine"



Note: sequence of *tokens* (not characters)

Derivation of an expression

Derivation

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

factors and terms

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

Remarks concerning the derivation

Note:

- input = stream of tokens
- there: **1**... stands for token class **number** (for readability/concreteness), in the grammar: just **number**
- in full detail: pair of token class and token value $\langle \mathbf{number}, 1\rangle$

Notation:

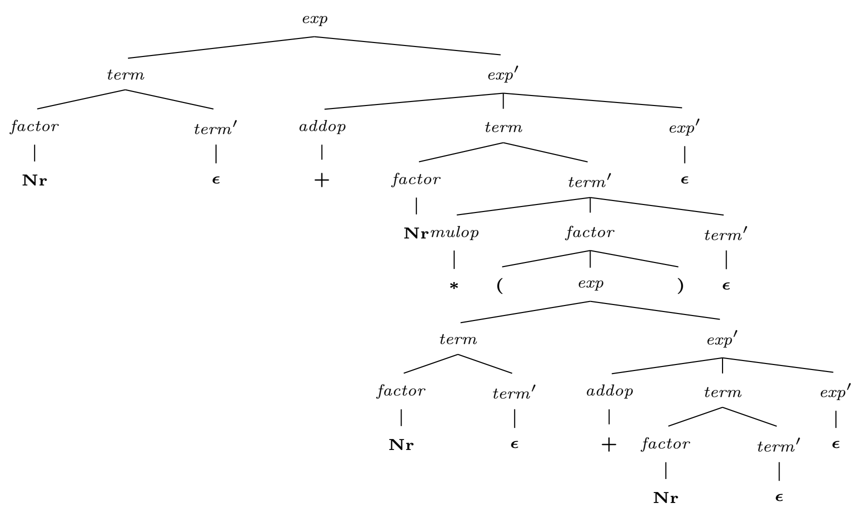
- underline: the *place* (occurrence of *non-terminal* where production is used)
- ~~*crossed out*~~:
  - *terminal* = *token* is considered treated
  - parser “moves on”
  - later implemented as `match` or `eat` procedure

Not as a “film” but at a glance: reduction sequence

$$
\begin{array}{ll}
\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} * \cancel{(}\ \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}\ \cancel{)}\ \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}\ ) &
\end{array}
$$

Besides this derivation sequence, the slide version contains also an “overlay” version, expanding the sequence step by step. The derivation is a *left-most* derivation.

Best viewed as a tree



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

Non-determinism?

- not a "free" expansion/reduction/generation of some word, but
  - reduction of start symbol towards the *target word of terminals*

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

  - i.e.: input stream of tokens "guides" the derivation process (at least it fixes the target)
- but: how much "guidance" does the target word (in general) gives?

Oracular derivation

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

$$\begin{array}{lll}
\underline{\mathit{exp}} & \Rightarrow_1 & \downarrow \mathsf{1} + \mathsf{2} * \mathsf{3} \\
\underline{\mathit{exp}} + \mathit{term} & \Rightarrow_3 & \downarrow \mathsf{1} + \mathsf{2} * \mathsf{3} \\
\underline{\mathit{term}} + \mathit{term} & \Rightarrow_5 & \downarrow \mathsf{1} + \mathsf{2} * \mathsf{3} \\
\underline{\mathit{factor}} + \mathit{term} & \Rightarrow_7 & \downarrow \mathsf{1} + \mathsf{2} * \mathsf{3} \\
\mathbf{number} + \mathit{term} & & \downarrow \mathsf{1} + \mathsf{2} * \mathsf{3} \\
\mathbf{number} + \mathit{term} & & \mathsf{1} \downarrow + \mathsf{2} * \mathsf{3} \\
\mathbf{number} + \underline{\mathit{term}} & \Rightarrow_4 & \mathsf{1} + \downarrow \mathsf{2} * \mathsf{3} \\
\mathbf{number} + \underline{\mathit{term}} * \mathit{factor} & \Rightarrow_5 & \mathsf{1} + \downarrow \mathsf{2} * \mathsf{3} \\
\mathbf{number} + \underline{\mathit{factor}} * \mathit{factor} & \Rightarrow_7 & \mathsf{1} + \downarrow \mathsf{2} * \mathsf{3} \\
\mathbf{number} + \mathbf{number} * \mathit{factor} & & \mathsf{1} + \downarrow \mathsf{2} * \mathsf{3} \\
\mathbf{number} + \mathbf{number} * \mathit{factor} & & \mathsf{1} + \mathsf{2} \downarrow * \mathsf{3} \\
\mathbf{number} + \mathbf{number} * \underline{\mathit{factor}} & \Rightarrow_7 & \mathsf{1} + \mathsf{2} * \downarrow \mathsf{3} \\
\mathbf{number} + \mathbf{number} * \mathbf{number} & & \mathsf{1} + \mathsf{2} * \downarrow \mathsf{3} \\
\mathbf{number} + \mathbf{number} * \mathbf{number} & & \mathsf{1} + \mathsf{2} * \mathsf{3} \downarrow
\end{array}$$

The derivation shows a left-most derivation. Again, the "redex" is underlined. In addition, we show on the right-hand column the input and the progress which is being done on that input. The subscripts on the derivation arrows indicate which rule is chosen in that particular derivation step.

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

What does that mean? It means, that the look-ahead did not help the parser! It used all look-ahead there is, namely until the very end of the word. And it *still* cannot make the right decision with all the knowledge available at that given point. Note also: choosing wrongly (like $\Rightarrow_5$ instead of $\Rightarrow_4$ or the other way around) would lead to a failed parse (which would require *backtracking*). That means, it's unparseable without backtracking (and not amount of look-ahead will help), at least we need backtracking, if we do left-derivations and top-down.

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

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

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

$$\mathit{term} \rightarrow \mathit{term} * \mathit{factor}$$

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

Note also: the grammar is not *ambigious* (without proof). If a grammar is ambiguous, also then parsing won't work properly (in this case neither will bottom-up parsing), but ambiguity is not the problem right here.

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.

As side remark, for being super-precise: If a grammar contains left-recursion on a non-terminal which is "irrelevant" (i.e., no word will ever lead to a parse invovling that particular non-terminal), in that case, obviously, the left-recursion does not hurt. Of course, the grammar in that case would be "silly". We in general do not consider grammars which contain such irrelevant symbols (or have other such obviously meaningless defects). But unless we exclude such silly grammars, it's not 100% true that grammars with left-recursion cannot be treated via top-down parsing. But apart from that, it's the case:

> left-recursion destroys top-down parseability

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

Two principle sources of non-determinism

Using production $A \to \beta$

$$S \Rightarrow^* \alpha_1\ A\ \alpha_2 \Rightarrow \alpha_1\ \beta\ \alpha_2 \Rightarrow^* w$$

Conventions

- $\alpha_1, \alpha_2, \beta$: word of terminals and nonterminals
- w: word of terminals, only
- A: one non-terminal

2 choices to make

1. **where**, i.e., on **which occurrence of a non-terminal** in $\alpha_1 A \alpha_2$ to apply a production
2. **which production** to apply (for the chosen non-terminal).

Note that α_1 and α_2 may contain non-terminals, including further occurrences of A. However, the words w_1 and w_2 contain terminals, only. By convention, A, B, etc. are non-terminal symbols, w ... are words of terminals, and greek-lettered symbols α, β ... represent words of terminals and non-terminals.

Left-most derivation

- that's the *easy* part of non-determinism
- taking care of "where-to-reduce" non-determinism: *left-most* derivation
- notation $\Rightarrow_l$
- some of the example derivations earlier used that

Non-determinism vs. ambiguity

- Note: the "where-to-reduce"-non-determinism $\neq$ ambiguitiy of a grammar
- in a way ("theoretically"): where to reduce next is *irrelevant*:
  - the order in the sequence of derivations *does not matter*
  - what does matter: the **derivation tree** (aka the **parse tree**)

Lemma 4.2.1 (Left or right, who cares)**.** *$S \Rightarrow_l^* w$ iff $S \Rightarrow_r^* w$ iff $S \Rightarrow^* w$.*

- however ("practically"): a (deterministic) parser implementation: must make a *choice*

Using production $A \to \beta$

$$S \Rightarrow^* \alpha_1 \; A \; \alpha_2 \Rightarrow \alpha_1 \; \beta \; \alpha_2 \Rightarrow^* w$$

$$S \Rightarrow_l^* w_1 \; A \; \alpha_2 \Rightarrow w_1 \; \beta \; \alpha_2 \Rightarrow_l^* w$$

Remember the notational conventions used here: w stand for words containing *terminals* only, whereas α represents arbitrary words.

What about the "which-right-hand side" non-determinism?

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

Is that the correct choice?

$$S \Rightarrow_l^* w_1 \; A \; \alpha_2 \Rightarrow w_1 \; \beta \; \alpha_2 \Rightarrow_l^* w$$

- reduction with "guidance": don't loose sight of the target w
  - "past" is fixed: $w = w_1 w_2$
  - "future" is not:

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

Needed (minimal requirement):

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

Deterministic, yes, but still impractical

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

- the "target" w_2 is of *unbounded length*!

⇒ impractical, therefore:

Look-ahead of length k

resolve the "which-right-hand-side" non-determinism inspecting only fixed-length prefix of w_2 (for *all* situations as above)

LL(k) grammars

CF-grammars which *can* be parsed doing that.

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

4.3 First and follow sets

The considerations leading to a useful criterion for top-down parsing with backtracking will involve the definition of the so-called "first-sets". In connection with that definition, there will be also the (related) definition of *follow-sets*.

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

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

Besides that, in this section we will also discuss *what to do* if the grammar is not LL(1). That will lead to a transformation removing left-recursion. That is not the only defect that one wants to transform away. A second problem that is a show-stopper for LL(1)-parsing is known as "common left factors". If a grammar suffers from that, there is another transformation called *left factorization* which can remedy that.

First and Follow sets

- general concept for grammars
- certain types of analyses (e.g. parsing):
  - info needed about possible “forms” of *derivable* words,

First-set of A

which terminal symbols can appear at the start of strings *derived from* a given nonterminal A

Follow-set of A

Which terminals can follow A in some *sentential form.*

Remarks

- sentential form: word *derived from* grammar’s starting symbol
- later: different algos for first and follow sets, for all non-terminals of a given grammar
- mostly straightforward
- one complication: *nullable* symbols (non-terminals)
- Note: those sets depend on grammar, not the language

First sets

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

$$\mathit{First}_G(A) = \{a \mid A \Rightarrow^*_G a\alpha, \quad a \in \Sigma_T\} + \{\epsilon \mid A \Rightarrow^*_G \epsilon\} \ . \tag{4.2}$$

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

Nullable

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

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

First and follow sets

One point worth noting is that the first and the follow sets, while seemingly quite similar, *differ* in one important aspect (the follow set definition will come later). The first set is about words derivable from a given non-terminal A. The follow set is about words derivable from the starting symbol! As a consequence, non-terminals A which are not *reachable* from the grammar's starting symbol have, by definition, an empty follow set. In contrast, non-terminals unreachable from a/the start symbol may well have a non-empty first-set. In practice a grammar containing unreachable non-terminals is ill-designed, so that distinguishing feature in the definition of the first and the follow set for a non-terminal may not matter so much. Nonetheless, when *implementing* the algo's for those sets, those subtle points do matter! In general, to avoid all those fine points, one works with grammars satisfying a number of common-sense restructions. One are so called *reduced grammars*, where, informally, all symbols "play a role" (all are reachable, all can derive into a word of terminals).

Examples

- Cf. the Tiny grammar
- in Tiny, as in most languages

$$\mathit{First}(\mathit{if\text{-}stmt}) = \{\text{"}\mathbf{if}\text{"}\}$$

- in many languages:

$$\mathit{First}(\mathit{assign\text{-}stmt}) = \{\mathbf{identifier}, \text{"("}\}$$

- typical *Follow* (see later) for statements:

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

Remarks

- note: special treatment of the empty word $\boldsymbol{\epsilon}$
- in the following: if grammar G clear from the context
  - $\Rightarrow^*$ for $\Rightarrow^*_G$
  - *First* for First_G
  - ...
- definition so far: "top-level" for start-symbol, only
- next: a more general definition
  - definition of First set of arbitrary symbols (and even words)

  - and also: definition of First for a symbol *in terms of* First for "other symbols" (connected by *productions*)

⇒ **recursive** definition

A more algorithmic/recursive definition

- grammar *symbol* X: terminal or non-terminal or $\boldsymbol{\epsilon}$

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

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

$$X \to X_1 X_2 \ldots X_n$$

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

Recursive definition of *First*?

The following discussion may be ignored, if wished. Even if details and theory behind it is beyond the scope of this lecture, it is worth considering the above definition more closely. One may even consider if it is a definition at all (resp. in which way it is a definition).

One naive first impression may be: it's a kind of a "functional definition", i.e., the above Definition 4.3.3 gives a recursive definition of the function *First*. As discussed later, everything gets rather simpler if we would not have to deal with nullable words and ϵ-productions. For the point being explained here, let's assume that there are no such productions and get rid of the special cases, cluttering up Definition 4.3.3. Removing the clutter gives the following simplified definition:

Definition 4.3.4 (First set of a symbol (no $\boldsymbol{\epsilon}$-productions))**.** Given a grammar G and grammar symbol X. The *First-set* of $X \neq \boldsymbol{\epsilon}$, written $\mathit{First}(X)$ is defined as follows:

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

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

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

Compared to the previous condition, I did the following minor adaptation (apart from cleaning up the $\boldsymbol{\epsilon}$'s): I replaced the English word "contains" with the superset relation symbol $\supseteq$.

Now, with Definition 4.3.4 as a simplified version of the original definition being made slightly more explicit: in which way is that a definition at all?

For being a definition for $\mathit{First}(X)$, it seems awfully lax. Already in (1), it "defines" that $\mathit{First}(X)$ should "at least contain X". A similar remark applies to case (2) for non-terminals. Those two requirements are as such well-defined, but *they don't define* $\mathit{First}(X)$ *in a unique manner!* Definition 4.3.4 defines what the set $\mathit{First}(X)$ should *at least* contain!

So, in a nutshell, one should not consider Definition 4.3.4 a "recursive definition of $\mathit{First}(X)$" but rather

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

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

Now one may think: the problem is thats definition is just "sloppy". Why does it use the word "contain" resp. the $\supseteq$-relation, instead of requiring equality, i.e., $=$? While plausible at first sight, unfortunately, whether we use $\supseteq$ or set equality $=$ in Definition 4.3.4 does not change anything.

Anyhow, the core of the matter is not $=$ vs. $\supseteq$. The core of the matter is that "Definition" 4.3.4 is *circular!*

Considering that definition of $\mathit{First}(X)$ as a plain functional and recursive definition of a procedure missed the fact that grammar can, of course, contain "loops". Actually, it's almost a characterizing feature of reasonable context-free grammars (or even regular grammars) that they contain "loops" – that's the way they can describe infinite languages.

In that case, obviously, considering Definition 4.3.3 with $=$ instead of $\supseteq$ as the recursive definition of a function leads immediately to an "infinite regress", the recursive function won't terminate. So again, that's not helpful.

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

For words

Definition 4.3.5 (First set of a word)**.** Given a grammar G and word α. The *first-set* of

$$\alpha = X_1 \ldots X_n \ ,$$

written $\mathit{First}(\alpha)$ is defined inductively as follows:

1. $\mathit{First}(\alpha)$ contains $\mathit{First}(X_1) \setminus \{\boldsymbol{\epsilon}\}$

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

Concerning the definition of First

The definition here is of course very close to the definition of the inductive case of the previous definition, i.e., the first set of a non-terminal. Whereas the previous definition was a recursive, this one is not.

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

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

The second question has no "real" answer, it's a choice which is being made which could be made differently. What the definition from equation (4.3.1) in fact says is: "give the set of terminal symbols in the derivable word **and** indicate whether or not the start symbol is *nullable*." The information might as well be interpreted as a *pair* consisting of a set of terminals *and* a boolean (indicating nullability). The fact that the definition of *First* as presented here uses $\boldsymbol{\epsilon}$ to indicate that additional information is a particular choice of representation (probably due to historical reasons: "they always did it like that ..."). For instance, the influential "Dragon book" [**?** , Section 4.4.2] uses the $\boldsymbol{\epsilon}$-based definition. The texbooks [**?**] (and its variants) don't use $\boldsymbol{\epsilon}$ as indication for nullability.

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

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

As a further point concerning the first sets: The slides give 2 definitions, Definition 4.3.1 and Definition 4.3.3. Of course they are intended to mean the same. The second version is a more recursive or algorithmic version, i.e., closer to a recursive algorithm. If one takes the first one as the "real" definition of that set, in principle we would be obliged to prove that both versions actually describe the same same (resp. that the recurive definition *implements* the original definition). The same remark applies also to the non-recursive/iterative code that is shown next.

Pseudo code

```
for all $X \in A \cup \{\epsilon\}$ do
    First[X] := X
end;

for all non-terminals $A$ do
  First[A] := {}
end
while there are changes to any First[A] do
  for each production $A \to X_1 \ldots X_n$ do
    k := 1;
    continue := true
    while continue = true and $k \leq n$ do
      First[A] := First[A] ∪ First[$X_k$] \ {$\epsilon$}
      if $\epsilon \notin$ First[$X_k$] then continue := false
      k := k + 1
    end;
    if  continue = true
    then First[A] := First[A] ∪ {$\epsilon$}
  end;
end
```

If only we could do away with special cases for the empty words . . .

for a grammar without ϵ-*productions*.[1]

```
for all non-terminals $A$ do
  First[A] := {}        // counts as change
end
while there are changes to any First[A] do
  for each production $A \to X_1 \ldots X_n$ do
      First[A] := First[A] ∪ First[$X_1$]
  end;
end
```

This simplification is added for *illustration*. What makes the algorithm slightly more than just immediate is the fact that symbols can be *nullable* (non-terminals can be nullable). If we don't have ϵ-transitions, then no symbol is nullable. Under this simplifying assumption, the algorithm looks quite simpler. We don't need to check for nullability (i.e., we don't need to check if ϵ is part of the first sets), and moreover, we can do without the inner while-loop, walking down the right-hand side of the production as long as the symbols turn out to be nullable (since we know they are not).

Example expression grammar (from before)

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{exp}\ \mathit{addop}\ \mathit{term} \mid \mathit{term} \\ \mathit{addop} & \to & + \mid - \\ \mathit{term} & \to & \mathit{term}\ \mathit{mulop}\ \mathit{factor} \mid \mathit{factor} \\ \mathit{mulop} & \to & * \\ \mathit{factor} & \to & (\ \mathit{exp}\ ) \mid \mathbf{number} \end{array} \tag{4.3}$$

[1]A production of the form $A \to \epsilon$.

Example expression grammar (expanded)

$$\begin{array}{rcl} exp & \to & exp\ \ addop\ \ term \\ exp & \to & term \\ addop & \to & + \\ addop & \to & - \\ term & \to & term\ \ mulop\ \ factor \\ term & \to & factor \\ mulop & \to & * \\ factor & \to & (\ exp\ ) \\ factor & \to & \mathbf{n} \end{array} \tag{4.4}$$

"Run" of the algo

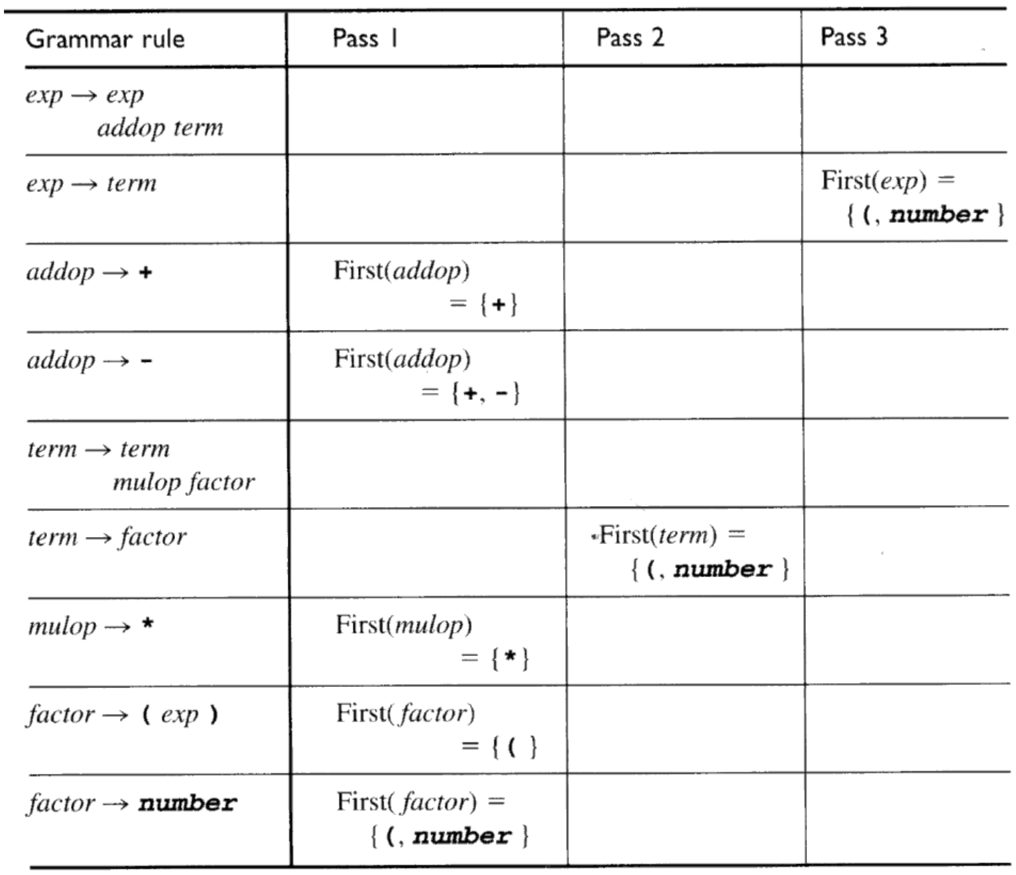
nr		pass 1	pass 2	pass 3
1	$exp \to exp\, addop\, term$			
2	$exp \to term$			
3	$addop \to +$			
4	$addop \to -$			
5	$term \to term\, mulop\, factor$			
6	$term \to factor$			
7	$mulop \to *$			
8	$factor \to (\ exp\ )$			
9	$factor \to \mathbf{n}$			

How the algo works

The first thing to observe: the grammar does not contain ϵ-productions. That, very fortunately, simplifies matters considerably! It should also be noted that the table from above is a schematic illustration of a particular *execution strategy* of the pseudo-code. The pseudo-code itself leaves out details of the evaluation, notably *the order* in which non-deterministic choices are done by the code. The main body of the pseudo-code is given by two nested loops. Even if details (of data structures) are not given, one possible way of interpreting the code is as follows: the outer while-loop figures out which of the entries in the `First`-array have "recently" been changed, remembers that in a "collection" of non-terminals A's, and that collection is then worked off (i.e. iterated over) on the inner loop. Doing it like that leads to the "passes" shown in the table. In other words, the two dimensions of the table represent the fact that there are 2 nested loops.

Having said that: it's not the only way to "traverse the productions of the grammar". One could arrange a version with only one loop and a collection data structure, which contains all productions $A \to X_1 \dots X_n$ such that `First[A]` has "recently been changed". That data structure therefore contains all the productions that "still need to be treated". Such a collection data structure containing "all the work still to be done" is known as *work-list*, even if it needs not technically be a list. It can be a queue, i.e., following a FIFO strategy, it can be a stack (realizing LIFO), or some other strategy or heuristic. Possible is also a randomized, i.e., non-deterministic strategy (which is sometimes known as chaotic iteration).

"Run" of the algo



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

Collapsing the rows & final result

- results per pass:

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

- final results (at the end of pass 3):

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

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

Work-list formulation

```
for all non-terminals $A$ do
  First[A] := {}
  WL       := $P$    // all productions
end
while WL $\neq \emptyset$ do
  remove one $(A \to X_1 \dots X_n)$ from WL
  if    First[A] $\neq$ First[A] $\cup$ First[$X_1$]
  then  First[A] := First[A] $\cup$ First[$X_1$]
        add all productions $(A \to X'_1 \dots X'_m)$ to WL
  else  skip
end
```

- no ϵ-productions
- worklist here: "collection" of productions
- alternatively, with slight reformulation: "collection" of non-terminals instead also possible

Follow sets

Definition 4.3.6 (Follow set)**.** Given a grammar G with start symbol S, and a non-terminal A.

The *follow-set* of A, written $\mathit{Follow}_G(A)$, is

$$\mathit{Follow}_G(A) = \{a \mid S\,\$ \Rightarrow^*_G \alpha_1 A a \alpha_2, \quad a \in \Sigma_T + \{\,\$\,\}\} \,. \tag{4.5}$$

- $ as special end-marker
- typically: start symbol *not* on the right-hand side of a production

Special symbol $

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

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

Coming back to the condition here concerning **$**: even if a grammar would not immediatly adhere to that condition, it's trivial to transform it into that form by adding another symbol and make that the new start symbol, replacing the old. We will do that sometimes in exercises and examples later

Follow sets, recursively

Definition 4.3.7 (Follow set of a non-terminal)**.** Given a grammar G and nonterminal A. The *Follow-set* of A, written $\mathit{Follow}(A)$ is defined as follows:

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

- **$**: "end marker" special symbol, only to be contained in the follow set

More imperative representation in pseudo code

```
Follow[$S$] := {$}
for all non-terminals $A \neq S$ do
  Follow[$A$] := {}
end
while there are changes to any Follow−set do
  for each production $A \to X_1 \ldots X_n$ do
    for each $X_i$ which is a non−terminal do
      Follow[$X_i$] := Follow[$X_i$] ∪ (First($X_{i+1} \ldots X_n$) \ {$\epsilon$})
      if $\epsilon \in$ First($X_{i+1} X_{i+2} \ldots X_n$)
      then Follow[$X_i$] := Follow[$X_i$] ∪ Follow[$A$]
    end
  end
end
```

Note! $First() = \{\epsilon\}$

"Run" of the algo

nr		pass 1	pass 2
1	$exp \to exp\ addop\ term$		
2	$exp \to term$		
5	$term \to term\ mulop\ factor$		
6	$term \to factor$		
8	$factor \to (\ exp\ )$		

Explanations

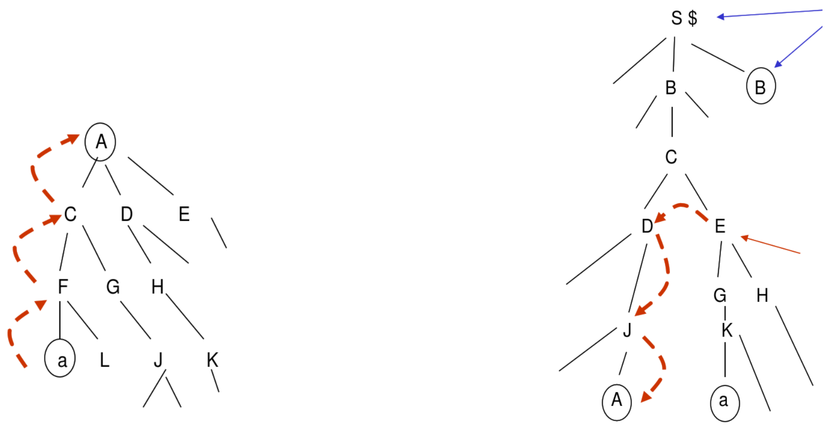
The table omits productions which have terminals only on their right-hand side. The algo does not do anything in those cases anyway. The grammar does not contain nullable symbols, which means, the algo is a bit more simple. We remember, that the first-procedure used ϵ for nullable symbol. However, the first procedure here is used non on non-terminals, but on *words*. And that word $X_{i+1} \ldots X_n$ may itself be ϵ, and that is where the last clause of the algo kicks in.

Recursion vs. iteration

“Run” of the algo

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

Illustration of first/follow sets



- red arrows: illustration of *information flow* in the algos
- run of *Follow*:
  - relies on *First*
  - in particular $a \in First(E)$ (right tree)
- $\$ \in Follow(B)$

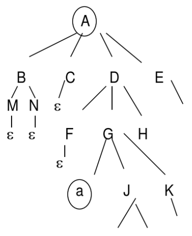
The two trees are just meant a illustrations (but still correct). The grammar itself is not given, but the tree shows relevant productions.

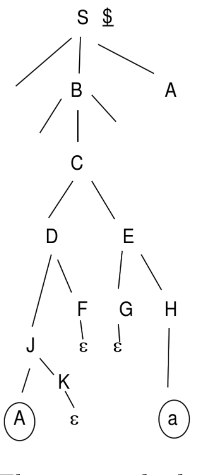
In case of the tree on the left (for the first sets): A is the root and must therefore be the start symbol. Since the root A has three children C, D, and E, there must be a production $A \to C\ D\ E$. etc.

The first-set definition would "immediately" detect that F has **a** in its first-set, i.e., all words derivable starting from F start with an a (and actually with no other terminal, as F is mentioned only once in that sketch of a tree). At any rate, only *after* determining that **a** is in the first-set of F, then it can enter the first-set of C, etc. and in this way percolating upwards the tree.

Note that the tree is insofar specific, in that all the internal nodes are *different* non-terminals. In more realistic settings, different nodes would represent the same non-terminal. And also in this case, one can think of the information percolating up.

More complex situation (nullability)





In the tree on the left, B, M, N, C, and F are *nullable*. That is marked in that the resulting first sets contain $\boldsymbol{\epsilon}$. There will also be exercises about that.

4.4 Massaging grammars

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

By diagnosis, I mean in particular: the concepts can be used to check whether or not it's possible to make parse a given grammar with a look-ahead of one symbol. The whole picture could more or less straightforwardly be generalized for a longer look-ahead: top-down parsing or bottom-up parsing with a look-ahead of k would require approporiate generalizations of the first-sets and follow-sets to speak not about $k = 1$ symbol but longer words. In practice, one mostly is content with $k = 1$, which is also why we don't bother about generalizing the setting. And actually, of one understands the concept of one look-ahead, nothing conceptually changes when going to $k > 1$.

As said, the first- and follow set are relevant for both top-down and bottom-up parsers. Here, however, we are in the part covering top-down parsing, which has slight different challenges than bottom-up. Before we come actually to top-down parsing, we discuss, what are problematic pattern 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 we will discuss that way are *left-recursive* grammars and grammars with common *left factors*. We will also dicsuss, how to massage troublesome grammars in a way to get rid of those patterns.

Some forms of grammars are less desirable than others

- **left-recursive** production:

$$A \to A\alpha$$

more precisely: example of *immediate* left-recursion

- 2 productions with **common "left factor"**:

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

Left-recursive and unfactored grammars

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

Why common left-factors are undesirable should at least intuitively be clear: we see this also on the next slide (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 intiutively 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). Ambiguous grammars are generally unwelcome as specification for parsers.

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

Some simple examples for both

- left-recursion

$$exp \to exp + term$$

- classical example for common left factor: rules for conditionals

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

We had a version of conditionals earlier, there

Transforming the expression grammar

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

- obviously left-recursive
- remember: this variant used for proper **associativity**!

After removing left recursion

$$
\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \rightarrow & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \boldsymbol{\epsilon} \\
\mathit{addop} & \rightarrow & + \mid - \\
\mathit{term} & \rightarrow & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \rightarrow & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \boldsymbol{\epsilon} \\
\mathit{mulop} & \rightarrow & * \\
\mathit{factor} & \rightarrow & (\ \mathit{exp}\ ) \mid \mathbf{n}
\end{array}
$$

- still *unambiguous*
- unfortunate: *associativity* now different!
- note also: $\boldsymbol{\epsilon}$-productions & nullability

Left-recursion removal

Left-recursion removal

A transformation process to turn a CFG into one without left recursion

Explanation

- price: $\boldsymbol{\epsilon}$-productions
- *3 cases* to consider
  - immediate (or direct) recursion
    - simple
    - general
  - *indirect* (or mutual) recursion

Left-recursion removal: simplest case

Before

$$A \rightarrow A\alpha \mid \beta$$

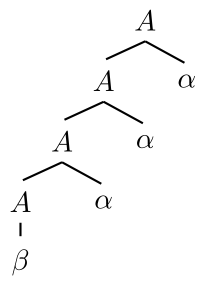
After

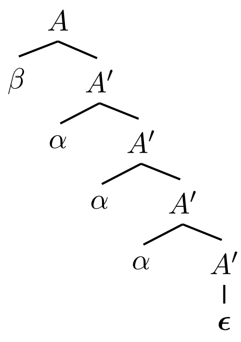
$$
\begin{array}{rcl}
A & \rightarrow & \beta A' \\
A' & \rightarrow & \alpha A' \mid \boldsymbol{\epsilon}
\end{array}
$$

Schematic representation

$$A \;\rightarrow\; A\alpha \mid \beta$$

$$\begin{array}{rcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon \end{array}$$





Remarks

- both grammars generate the same (context-free) language (= set of words over terminals)
- in EBNF:

$$A \rightarrow \beta\{\alpha\}$$

- two *negative* aspects of the transformation
  1. generated language unchanged, but: change in resulting structure (parse-tree), i.a.w. change in **associativity**, which may result in change of *meaning*
  2. introduction of ϵ-productions
- more concrete example for such a production: grammar for expressions

Left-recursion removal: immediate recursion (multiple)

Before

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

space

After

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

EBNF

Note: can be written in *EBNF* as:

$$A \to (\beta_1 \mid \dots \mid \beta_m)(\alpha_1 \mid \dots \mid \alpha_n)^*$$

Removal of: general left recursion

Assume non-terminals $A_1, \dots, A_m$

```
for i := 1 to m do
   for j := 1 to i−1 do
      replace each grammar rule of the form $A_i \to A_j\beta$ by  // $i < j$
      rule $A_i \to \alpha_1\beta \mid \alpha_2\beta \mid \dots \mid \alpha_k\beta$
           where $A_j \to \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$
           is the current rule(s) for $A_j$  // current
   end
   { corresponds to $i = j$ }
   remove, if necessary, immediate left recursion for $A_i$
end
```

"current" = rule in the current stage of algo

Example (for the general case)

Let $A = A_1$, $B = A_2$.

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

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

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

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

Left factor removal

- CFG: not just describe a context-free languages
- also: intended (indirect) description of a **parser** for that language

⇒ common left factor undesirable

- cf.: *determinization* of automata for the lexer

Simple situation

before

$$A \to \alpha\beta \mid \alpha\gamma \mid \ldots$$

after

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

Example: sequence of statements

sequences of statements

Before

$$\begin{array}{rcl} \mathit{stmt\text{-}seq} & \to & \mathit{stmt}\,\mathbf{;}\,\mathit{stmt\text{-}seq} \\ & \mid & \mathit{stmt} \end{array}$$

After

$$\begin{array}{rcl} \mathit{stmt\text{-}seq} & \to & \mathit{stmt}\ \mathit{stmt\text{-}seq}' \\ \mathit{stmt\text{-}seq}' & \to & \mathbf{;}\,\mathit{stmt\text{-}seq} \mid \epsilon \end{array}$$

Example: conditionals

Before

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

After

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

Example: conditionals (without else)

Before

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

After

$$\begin{array}{rcl} \mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt\text{-}seq}\ \mathit{else\text{-}or\text{-}empty} \\ \mathit{else\text{-}or\text{-}empty} & \to & \mathbf{else}\ \mathit{stmt\text{-}seq} \mid \epsilon \end{array}$$

Not all factorization doable in "one step"

Starting point

$$A \rightarrow \mathbf{abc}B \mid \mathbf{ab}C \mid \mathbf{a}E$$

After 1 step

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

After 2 steps

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

longest left factor

- note: we choose the *longest* common prefix (= longest left factor) in the first step

Left factorization

```
while  there are changes to the grammar  do
   for  each nonterminal $A$  do
      let  $\alpha$ be a prefix of max. length that is shared
                    by two or more productions for $A$
      if    $\alpha \neq \epsilon$
      then
           let  $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ be all
                    prod. for $A$ and suppose that $\alpha_1, \ldots, \alpha_k$ share $\alpha$
                    so that $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_k \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$ ,
                    that the $\beta_j$'s share no common prefix, and
                    that the $\alpha_{k+1}, \ldots, \alpha_n$ do not share $\alpha$.
           replace rule $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ by the rules
           $A \rightarrow \alpha A' \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$
           $A' \rightarrow \beta_1 \mid \ldots \mid \beta_k$
      end
   end
end
```

The algorithm is pretty straightforward. The only thing to keep in might is that what is called α in the pseudo-code needs to be 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.

Parsing LL(1) grammars

- *this lecture*: we don't do LL(k) with $k > 1$
- LL(1): particularly easy to understand and to implement (efficiently)
- 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**.

- two flavors for LL(1) parsing here (both are top-down parsers)
  - *recursive descent*
  - *table-based* LL(1) parser
- *predictive* parsers

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

Sample expression grammar again

factors and terms

$$\begin{array}{rcl}
exp & \to & term\ exp' \\
exp' & \to & addop\ term\ exp' \mid \boldsymbol{\epsilon} \\
addop & \to & + \mid - \\
term & \to & factor\ term' \\
term' & \to & mulop\ factor\ term' \mid \boldsymbol{\epsilon} \\
mulop & \to & * \\
factor & \to & (\ exp\ ) \mid \mathbf{n}
\end{array} \tag{4.6}$$

Look-ahead of 1: straightforward, but not trivial

- look-ahead of 1:
  - not much of a look-ahead, anyhow
  - just the "current token"

⇒ read the next token, and, based on that, decide

- but: what if there's *no more symbols*?

⇒ read the next token if there is, and decide based on the token *or else* the fact that there's none left[2]

Example: 2 productions for non-terminal *factor*

$$factor \rightarrow (\, exp\,) \mid \mathbf{number}$$

That situation here is more or less *trivial*, but that's not all to LL(1) ...

Recursive descent: general set-up

1. global variable, say `tok`, representing the "current token" (or pointer to current token)
2. parser has a way to *advance* that to the next token (if there's one)

Idea

For each *non-terminal nonterm*, write one procedure which:

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

- ignored (for now): when doing the above successfully, build the *AST* for the accepted nonterminal.

Recursive descent (in C-like)

method `factor` for nonterminal *factor*

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

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

[2]Sometimes "special terminal" **$** used to mark the end (as mentioned).

Recursive descent (in ocaml)

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

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

Slightly more complex

- previous 2 rules for *factor*: situation not always as immediate as that

LL(1) principle (again)

given a non-terminal, the next *token* must determine the choice of right-hand side.

When talking about the next token, it must be the next token/terminal in the sense of *First*, but it need not be a token *directly* mentioned on the right-hand sides of the corresponding rules.

⇒ definition of the *First* **set**

Lemma 4.5.1 (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 $\alpha_1 \neq \alpha_2$:*

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

Common problematic situation

- often: common *left factors* problematic

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

- requires a look-ahead of (at least) 2
- ⇒ try to rearrange the grammar
  1. *Extended* BNF ([3] suggests that)

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

  1. *left-factoring*:

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

Recursive descent for left-factored *if-stmt*

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

Left recursion is a no-go

factors and terms

$$
\begin{aligned}
exp &\rightarrow exp\ \ addop\ \ term \mid term \\
addop &\rightarrow + \mid - \\
term &\rightarrow term\ \ mulop\ \ factor \mid factor \\
mulop &\rightarrow * \\
factor &\rightarrow (\, exp \,) \mid \textbf{number}
\end{aligned}
\tag{4.7}
$$

- consider treatment of *exp*: $First(exp)$?
- whatever is in $First(term)$, is in $First(exp)$[3] recursion.

Left-recursion

Left-recursive grammar *never* works for recursive descent.

Removing left recursion may help

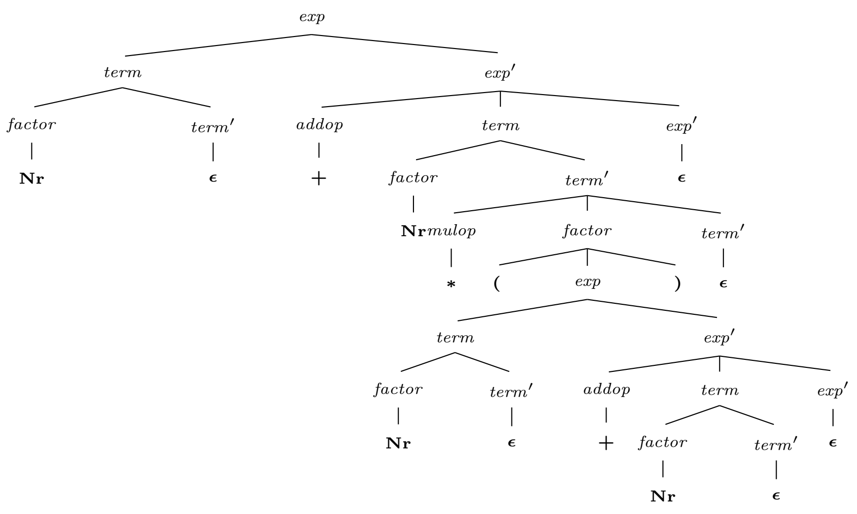
$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow (\, exp \,) \mid \textbf{n}
\end{aligned}
$$

[3]And it would not help to *look-ahead* more than 1 token either.

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

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

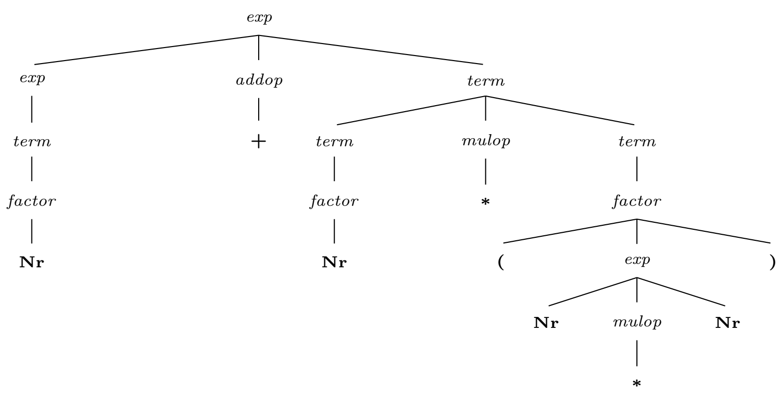
Recursive descent works, alright, but . . .



. . . who wants this form of trees?

Left-recursive grammar with nicer parse trees

$$1 + 2 * (3 + 4)$$

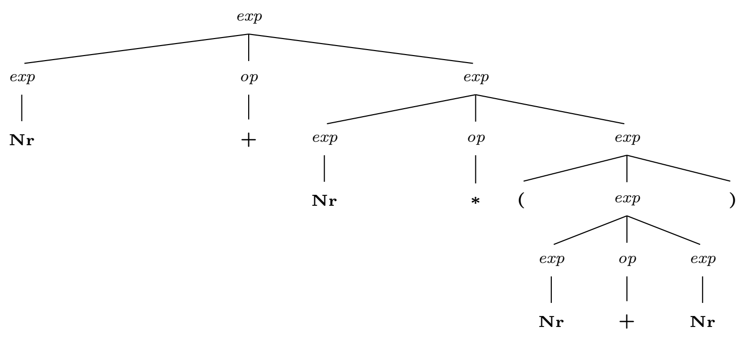


The simple “original” expression grammar (even nicer)

Flat expression grammar

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

$$1 + 2 * (3 + 4)$$



Associtivity problematic

The issues here, including associativity, have been touched upon already when discussing ambiguity.

Precedence & assoc.

$$\begin{array}{rcl} exp & \rightarrow & exp\ \ addop\ \ term \ \mid\ term \\ addop & \rightarrow & + \mid - \\ term & \rightarrow & term\ \ mulop\ \ factor \ \mid\ factor \\ mulop & \rightarrow & * \\ factor & \rightarrow & (\, exp\,) \ \mid\ \textbf{number} \end{array}$$

Formula

$$3 + 4 + 5$$

parsed "as"

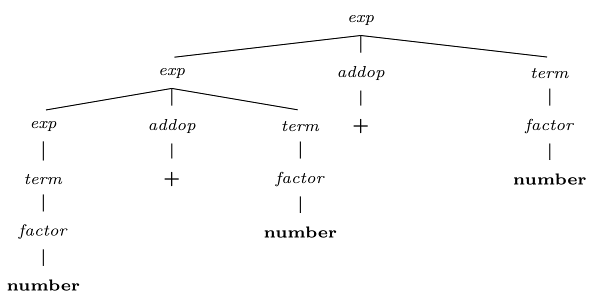
$$(3 + 4) + 5$$

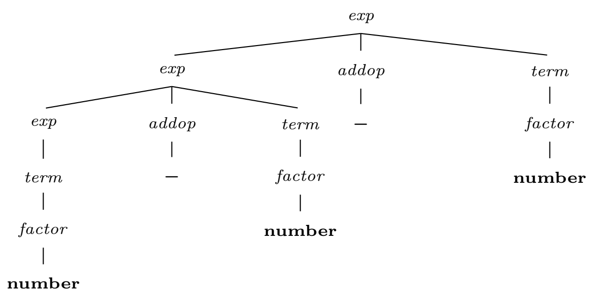
$$3 - 4 - 5$$

parsed "as"

$$(3 - 4) - 5$$

Tree





Now use the grammar without left-rec (but right-rec instead)

No left-rec.

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

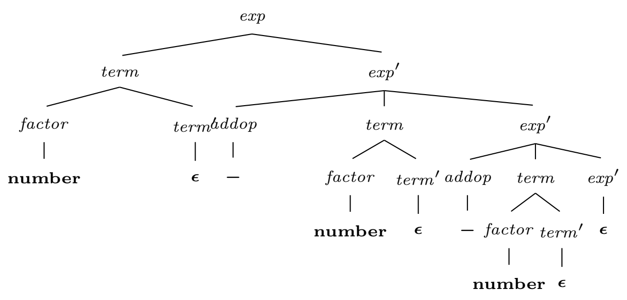
Formula

$$3 - 4 - 5$$

parsed "as"

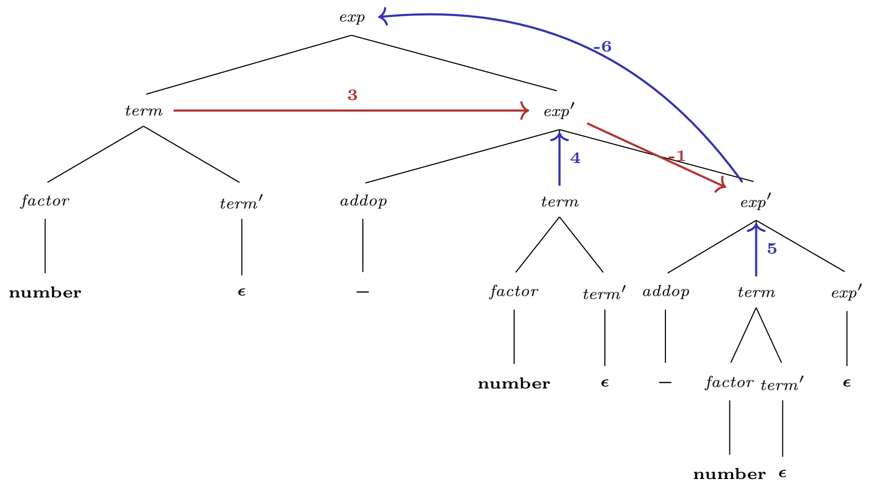
$$3 - (4 - 5)$$

Tree



But if we need a "left-associative" AST?

- we want $(3 - 4) - 5$, *not* $3 - (4 - 5)$



Code to "evaluate" ill-associated such trees correctly

```
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;
```

- extra "accumulator" argument `valsofar`
- instead of evaluating the expression, one could build the AST with the appropriate associativity instead:
- instead of `valueSoFar`, one had `rootOfTreeSoFar`

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

"Designing" the syntax, its parsing, & its AST

trade offs:

1. starting from: design of the language, how much of the syntax is left "implicit"[4]
2. which language class? Is LL(1) good enough, or something stronger wanted?
3. how to parse? (top-down, bottom-up, etc.)
4. parse-tree/concrete syntax trees vs. ASTs

AST vs. CST

- once steps 1.–3. are fixed: *parse-trees* fixed!
- parse-trees = *essence* of grammatical derivation process
- often: parse trees only "conceptually" present in a parser
- AST:
  - *abstractions* of the parse trees
  - *essence* of the parse tree

[4]Lisp is famous/notorious in that its surface syntax is more or less an explicit notation for the ASTs. Not that it was originally planned like this ...

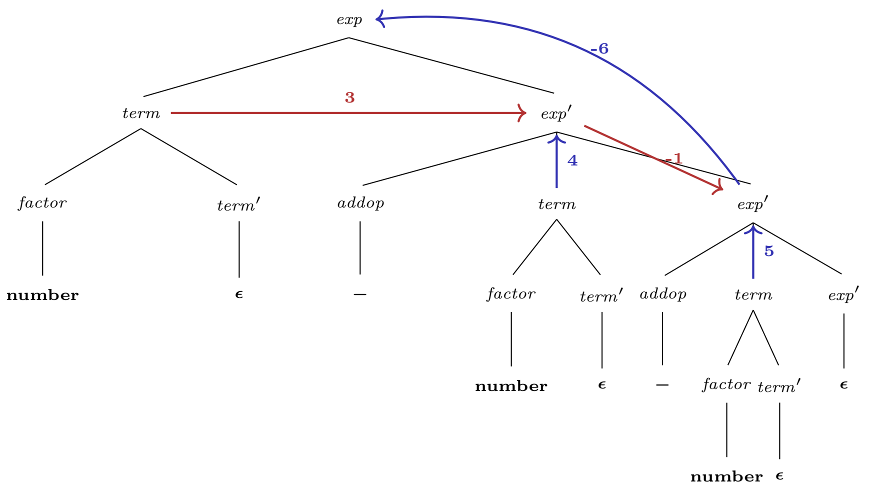
  - actual tree data structure, as output of the parser
  - typically on-the fly: AST built while the parser parses, i.e. while it executes a derivation in the grammar

AST vs. CST/parse tree

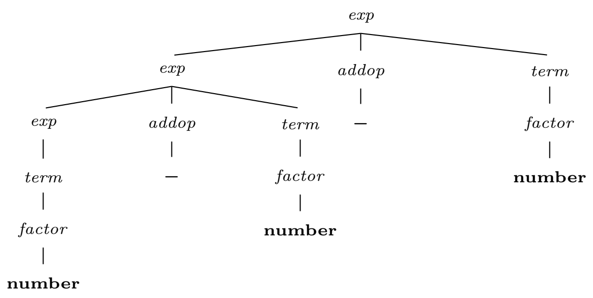
Parser "**builds**" the AST data structure while "**doing**" the parse tree

AST: How "far away" from the CST?

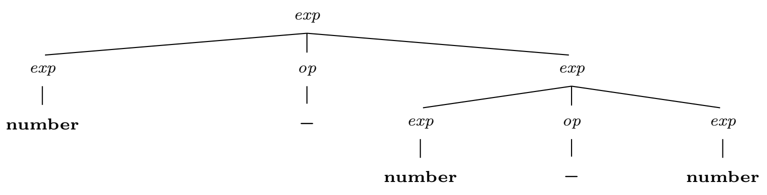
- AST: only thing relevant for later phases ⇒ better be *clean* . . .
- AST "=" CST?
  - building AST becomes straightforward
  - possible choice, **if** the grammar is not designed "weirdly",



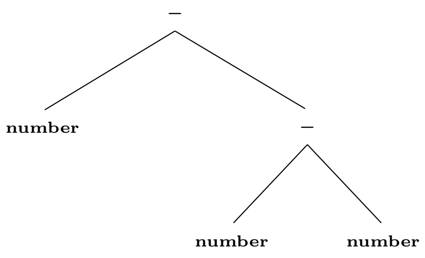
parse-trees like that better be cleaned up as AST

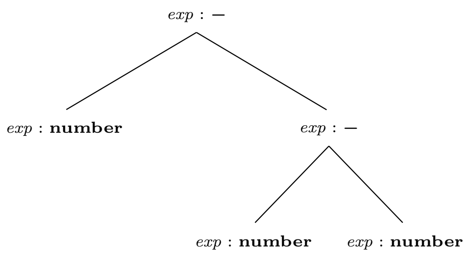


slightly more reasonably looking as AST (but underlying grammar not directly useful for recursive descent)



That parse tree looks reasonable clear and intuitive





Certainly minimal amount of nodes, which is nice as such. However, what is missing (which might be interesting) is the fact that the 2 nodes labelled "−" are *expressions!*

This is how it's done (a recipe)

Assume, one has a "non-weird" grammar

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

- typically that means: assoc. and precedences etc. are fixed *outside* the non-weird grammar
  - by massaging it to an equivalent one (no left recursion etc.)
  - or (better): use parser-generator that allows to *specify* assoc ... like " "$*$" *binds stronger* than "$+$", it *associates* to the left ... " , without cluttering the grammar.
- if grammar for *parsing* is not as clear: do a second one describing the ASTs

Remember (independent from parsing)

BNF describe **trees**

This is how it's done (recipe for OO data structures)

Recipe

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

Example in Java

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

```
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 latter classes are perhaps pushing it too far. It's done to show that one can mechanically use the *recipe* once grammar is given, so it's a clean solution (perhaps one get better efficiency if one would not make classes / objects out of everything, though).

$3 - (4 - 5)$

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

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

⇒ that class is *not* needed

- some might prefer an implementation of

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

as simply integers, for instance arranged like

```
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.

Recipe for ASTs, final words:

- space considerations for AST representations are irrelevant in most cases
- clarity and cleanness trumps "quick hacks" and "squeezing bits"
- some deviation from the recipe or not, the advice still holds:

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 (nonterminal *exp* represented by class `Exp`, non-terminal *stmt* by class `Stmt` etc)

Extended BNF may help alleviate the pain

BNF

$$\begin{array}{rcl} exp & \to & exp\ addop\ term \mid term \\ term & \to & term\ mulop\ factor \mid factor \end{array}$$

EBNF

$$\begin{array}{rcl} exp & \to & term\{\ addop\ term\ \} \\ term & \to & factor\{\ mulop\ factor\ \} \end{array}$$

but remember:

- EBNF just a notation, just because we do not see (left or right) recursion in { ... }, does not mean there is no recursion.
- not all parser generators support EBNF
- however: often easy to translate into loops- [5]
- does not offer a *general* solution if associativity etc. is problematic

Pseudo-code representing the EBNF productions

```
procedure exp;
begin
  term ;        { recursive call }
  while token = "+" or token = "-"
  do
    match(token);
    term;       // recursive call
  end
end
```

```
procedure term;
begin
  factor;        { recursive call }
  while token = "*"
  do
    match(token);
    factor;       // recursive call
  end
end
```

[5]That results in a parser which is somehow not “pure recursive descent”. It’s “recursive descent, but sometimes, let’s use a while-loop, if more convenient concerning, for instance, associativity”

How to produce "something" during RD parsing?

Recursive descent

So far (mostly): RD = top-down (parse-)tree traversal via recursive procedure.[6] Possible outcome: termination or failure.

- Now: instead of returning "nothing" (return type `void` or similar), return some meaningful, and build that up during traversal
- for illustration: procedure for expressions:
  - return type `int`,
  - while traversing: *evaluate* the expression

Evaluating an *exp* during RD parsing

```
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
```

Building an AST: expression

```
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
```

[6] 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.

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

Building an AST: factor

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

```
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
```

Building an AST: conditionals

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

```
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
```

Building an AST: remarks and "invariant"

- LL(1) requirement: each procedure/function/method (covering one specific non-terminal) decides on alternatives, looking only at the current token
- call of function `A` for non-terminal A:
  - upon entry: first terminal symbol for A in `token`
  - upon exit: first terminal symbol *after* the unit derived from A in `token`
- `match("a")` : checks for `"a"` in `token` *and eats* the token (if matched).

LL(1) parsing

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

- remember LL(1) grammars & LL(1) parsing principle:

LL(1) parsing principle

1 look-ahead enough to resolve "which-right-hand-side" non-determinism.

- instead of recursion (as in RD): *explicit stack*
- decision making: collated into the **LL(1) parsing table**
- LL(1) parsing table:
  - finite data structure M (for instance, a 2 dimensional array)

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

  - $M[A, a] = w$
- we assume: pure BNF

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

Construction of the parsing table

Table recipe

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 $S\,\$ \Rightarrow^* \beta A \mathbf{a} \gamma$ (where $\mathbf{a}$ is a token (=non-terminal) *or* $\$$), then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$

Table recipe (again, now using our old friends *First* and *Follow*)

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 $M[A, \mathbf{a}]$.

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 4.5.1.

Example: if-statements

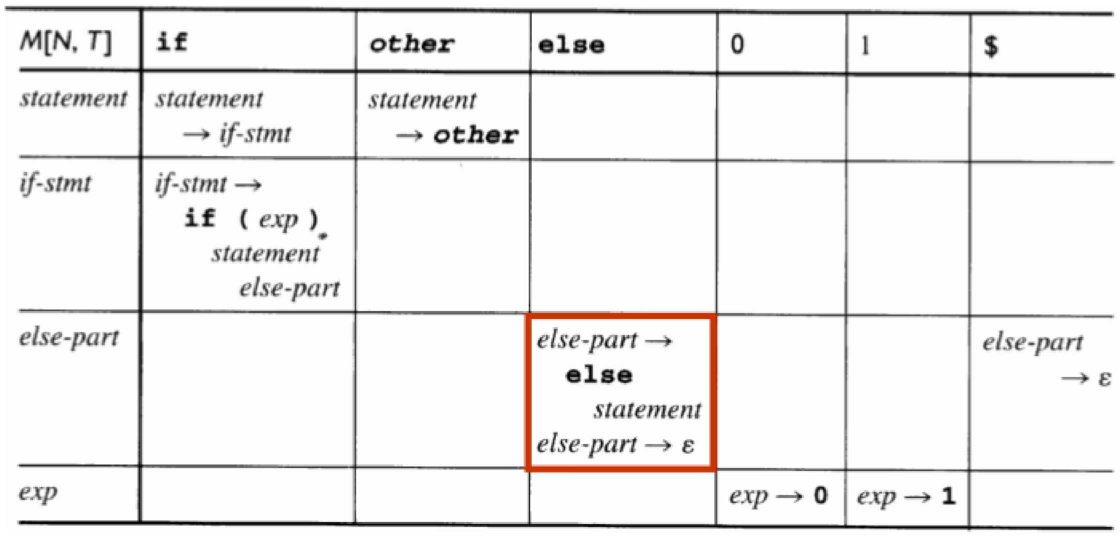
- grammars is left-factored and not left recursive

$$\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{if-stmt} \mid \textbf{other} \\
\textit{if-stmt} & \rightarrow & \textbf{if}\ (\ \textit{exp}\ )\ \textit{stmt}\ \textit{else-part} \\
\textit{else-part} & \rightarrow & \textbf{else}\ \textit{stmt} \mid \epsilon \\
\textit{exp} & \rightarrow & \textbf{0} \mid \textbf{1}
\end{array}$$

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

The slide 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.

Example: if statement: "LL(1) parse 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**	

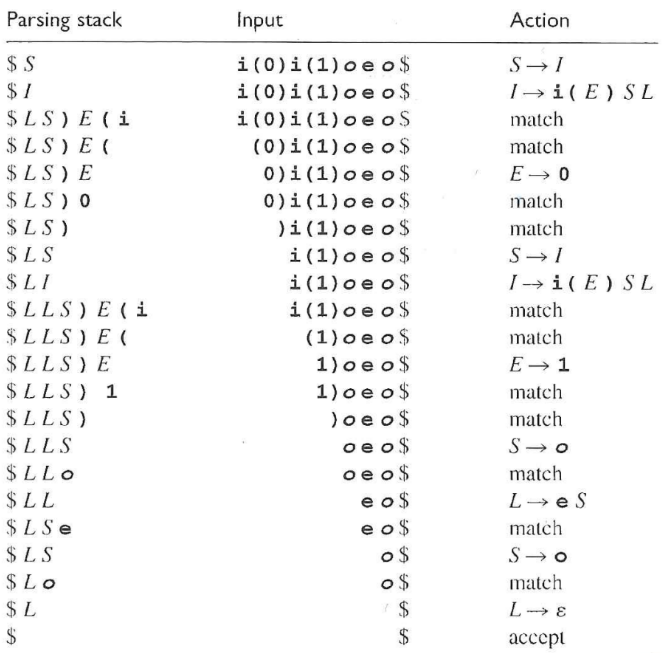
- 2 productions in the "red table entry"
- thus: it's technically *not* an LL(1) table (and it's not an LL(1) grammar)
- note: removing left-recursion and left-factoring did not help!

Saying that it's "not-an-LL(1)-table" is perhaps a bit nit-picking. The shape *is* according to the required format. It's only that in the slot marked red, there are two rules. That's a conflict and makes it at least not a legal LL(1) table. So, if in an exam question, 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). Similar remarks later for LR-parsers. Actually, for LR-parsers, tools like yacc build up a table (not an LL, but an LR-table) and, in case of double entries, making a choice which one to include. The user, in those cases, will reveive a warning about the grammar containing a corresponding *conflict*. So the user should be aware that the grammar is actually not parseable (because a parse would require backtracking, which is not done). 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 might be one. The given grammar “suffers” from the ambiguity called dangling-else problem. The left-factoring massage did not help there. 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.

LL(1) table-based algo

```
while  the top of the parsing stack $\neq$ $
  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''
  else  if   the top the parsing is non-terminal  $A$
          and   the next input token is  a  terminal or  $
          and   parsing table  $M[A,\mathbf{a}]$  contains
                production  $A \to X_1 X_2 \dots X_n$
          then  (* generate *)
                pop the parsing stack
                for  $i := n$  to  1  do
                push  $X_i$  onto the stack;
        else  error
  if    the top of the stack =  $
  then  accept
end
```

LL(1): illustration of a run of the algo



Parsing stack	Input	Action
$\$\, S$	$\mathtt{i(0)i(1)oeo}\$$	$S \to I$
$\$\, I$	$\mathtt{i(0)i(1)oeo}\$$	$I \to \mathtt{i}\,(\,E\,)\,S\,L$
$\$\, L\, S\,)\,E\,(\,\mathtt{i}$	$\mathtt{i(0)i(1)oeo}\$$	match
$\$\, L\, S\,)\,E\,($	$\mathtt{(0)i(1)oeo}\$$	match
$\$\, L\, S\,)\,E$	$\mathtt{0)i(1)oeo}\$$	$E \to \mathtt{0}$
$\$\, L\, S\,)\,\mathtt{0}$	$\mathtt{0)i(1)oeo}\$$	match
$\$\, L\, S\,)$	$\mathtt{)i(1)oeo}\$$	match
$\$\, L\, S$	$\mathtt{i(1)oeo}\$$	$S \to I$
$\$\, L\, I$	$\mathtt{i(1)oeo}\$$	$I \to \mathtt{i}\,(\,E\,)\,S\,L$
$\$\, L\, L\, S\,)\,E\,(\,\mathtt{i}$	$\mathtt{i(1)oeo}\$$	match
$\$\, L\, L\, S\,)\,E\,($	$\mathtt{(1)oeo}\$$	match
$\$\, L\, L\, S\,)\,E$	$\mathtt{1)oeo}\$$	$E \to \mathtt{1}$
$\$\, L\, L\, S\,)\,\mathtt{1}$	$\mathtt{1)oeo}\$$	match
$\$\, L\, L\, S\,)$	$\mathtt{)oeo}\$$	match
$\$\, L\, L\, S$	$\mathtt{oeo}\$$	$S \to \mathtt{o}$
$\$\, L\, L\, \mathtt{o}$	$\mathtt{oeo}\$$	match
$\$\, L\, L$	$\mathtt{eo}\$$	$L \to \mathtt{e}\, S$
$\$\, L\, S\, \mathtt{e}$	$\mathtt{eo}\$$	match
$\$\, L\, S$	$\mathtt{o}\$$	$S \to \mathtt{o}$
$\$\, L\, \mathtt{o}$	$\mathtt{o}\$$	match
$\$\, L$	$\$$	$L \to \varepsilon$
$\$$	$\$$	accept

The most interesting steps are of course those dealing with the dangling else, namely those with the non-terminal *else−part* at the top of the stack. That's where the LL(1) table is ambiguous. In principle, with *else−part* on top of the stack (in the picture it's just `L`), the parser table allows always to make the decision that the "current statement" resp "current conditional" is done.

Expressions

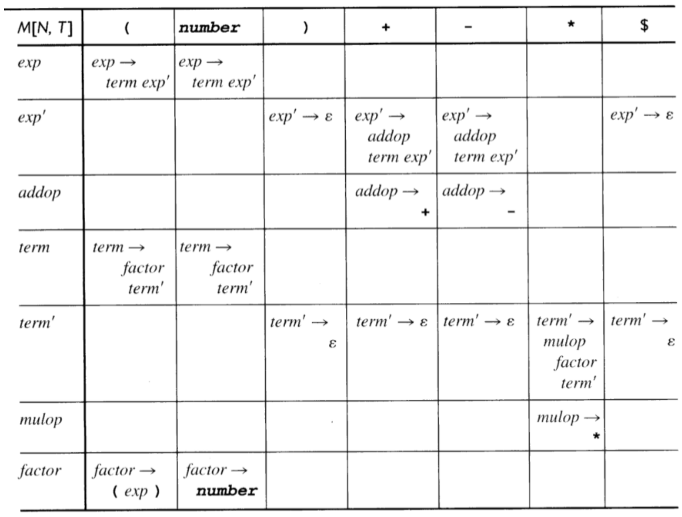
$$\begin{array}{rcl}
exp & \to & exp\ \ addop\ \ term \ \mid\ term \\
addop & \to & + \mid - \\
term & \to & term\ \ mulop\ \ factor \ \mid\ factor \\
mulop & \to & * \\
factor & \to & (\, exp\,) \ \mid\ \mathbf{number}
\end{array}$$

left-recursive ⇒ not LL(k)

$$\begin{array}{rcl}
exp & \to & term\ exp' \\
exp' & \to & addop\ term\ exp' \ \mid\ \boldsymbol{\epsilon} \\
addop & \to & + \mid - \\
term & \to & factor\ term' \\
term' & \to & mulop\ factor\ term' \ \mid\ \boldsymbol{\epsilon} \\
mulop & \to & * \\
factor & \to & (\, exp\,) \ \mid\ \mathbf{n}
\end{array}$$

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

Expressions: LL(1) parse table



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`					

4.6 Error handling

The error handling section is not part of the pensum (it never was), insofar it will not be asked in the written exam. That does not mean that, that we don't want some adequate error handling for the compiler in the oblig. The slides are not presented in detail in class. Parsers (and lexers) are built on some robust, established and well-understood theoretical foundations. That's less the case for how to deal with errors, where it's more of an art, and more pragmatics enter the pictures. It does not mean it's unimportant, it's just that the topic is less conceptually clarified. So, while certainly there is research, in compilers 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 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 maybe 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 encoutering an unexpected symbol, when currently looking at a field in the LL- or LR-table. That constitutes some "syntactic error" and should be reported, but it's not even clear what the "real cause" of an error is. Error localization as such cannot be formally solved, since one cannot properly define was the source of an error is in general. So, we focus here more on general "advice".

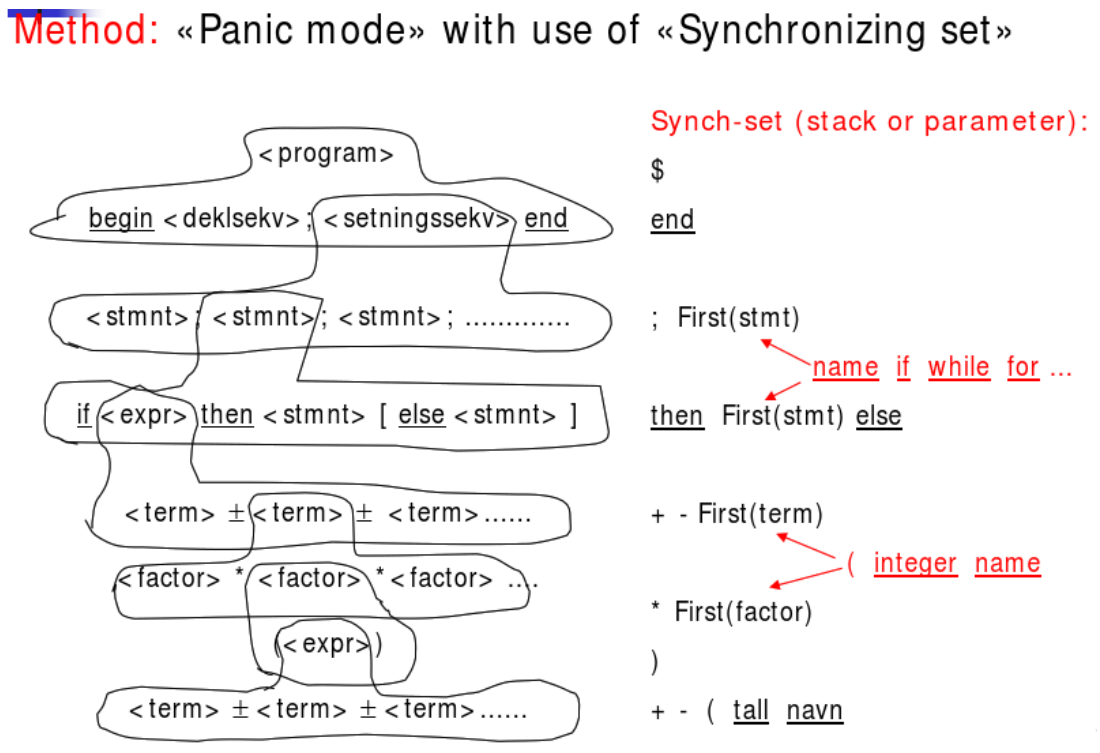
Error handling

- at the least: do an understandable error message
- give indication of line / character or region responsible for the error in the source file
- potentially *stop* the parsing
- some compilers do *error recovery*
  - give an understandable error message (as minimum)
  - continue reading, until it's plausible to resume parsing ⇒ find more errors
  - however: when finding at least 1 error: no code generation
  - observation: resuming after syntax error is not easy

Error messages

- important:
  - try to avoid error messages that only occur because of an already reported error!
  - report error as early as possible, if possible at the first point where the program cannot be extended to a correct program.
  - make sure that, after an error, one doesn't end up in a infinite loop without reading any input symbols.
- What's a good error message?
  - assume: that the method `factor()` chooses the alternative $(\,exp\,)$ but that it, when control returns from method `exp()`, does not find a $)$
  - one could report : `right paranthesis missing`
  - But this may often be confusing, e.g. if what the program text is: `( a + b c )`
  - here the `exp()` method will terminate after `( a + b`, as `c` cannot extend the expression). You should therefore rather give the message `error in expression or right paranthesis missing`.

Handling of syntax errors using recursive descent



Method: «Panic mode» with use of «Synchronizing set»

<program>
begin <deklsekv> ; <setningssekv> end
<stmnt> ; <stmnt> ; <stmnt> ; ....
if <expr> then <stmnt> [else <stmnt>]
<term> ± <term> ± <term> ......
<factor> * <factor> * <factor> ...
(<expr>)
<term> ± <term> ± <term> ......

Synch-set (stack or parameter):

$

end

; First(stmt)

name if while for ...

then First(stmt) else

+ - First(term)

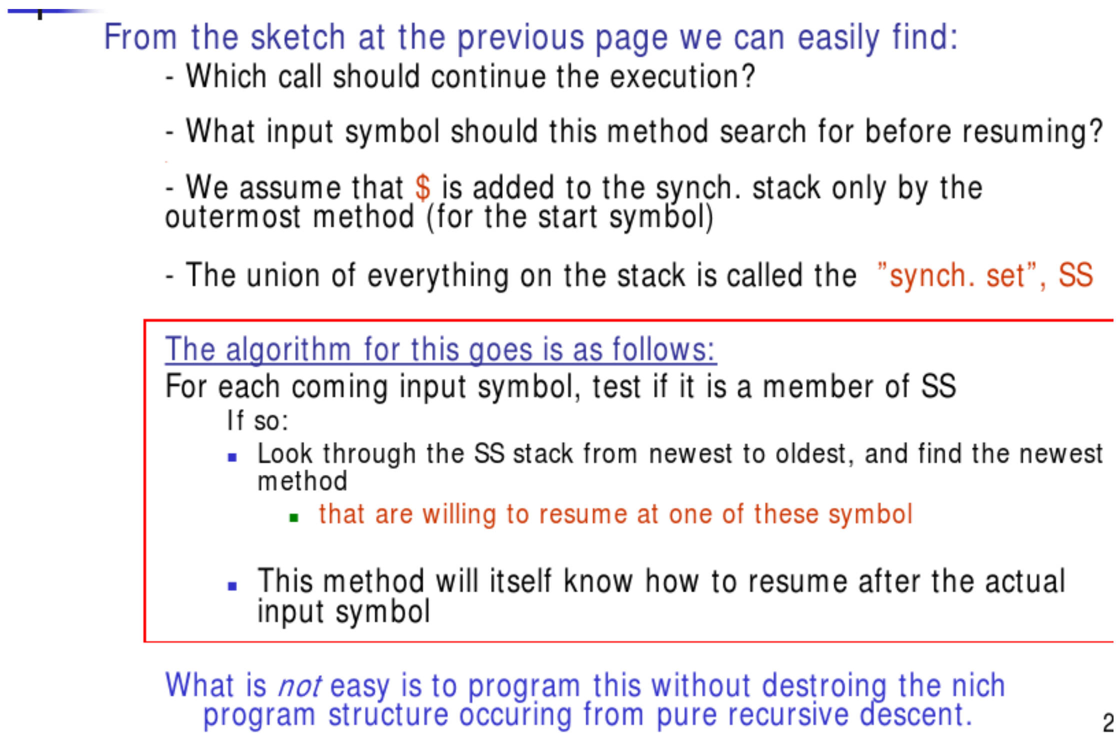
(integer name

* First(factor)

)

+ - (tall navn

Syntax errors with sync stack



From the sketch at the previous page we can easily find:

- Which call should continue the execution?
- What input symbol should this method search for before resuming?
- We assume that $ is added to the synch. stack only by the outermost method (for the start symbol)
- The union of everything on the stack is called the "synch. set", SS

The algorithm for this goes is as follows:

For each coming input symbol, test if it is a member of SS

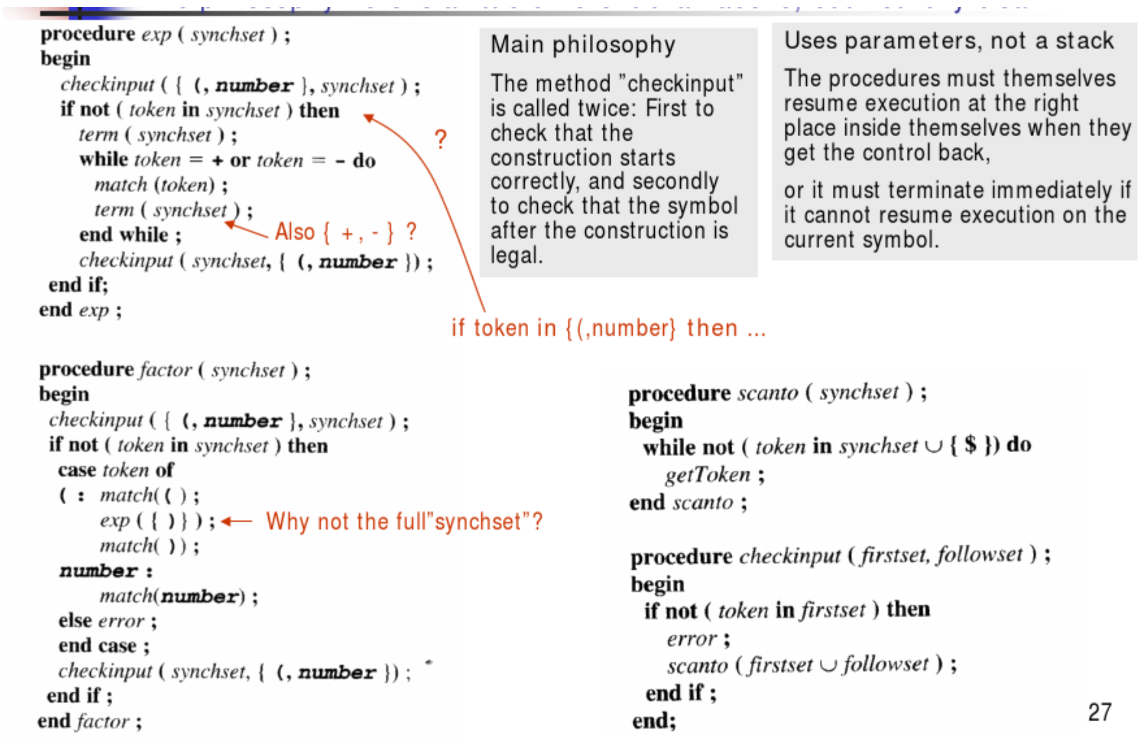
If so:

- Look through the SS stack from newest to oldest, and find the newest method
  - that are willing to resume at one of these symbol
- This method will itself know how to resume after the actual input symbol

What is *not* easy is to program this without destroing the nich program structure occuring from pure recursive descent.

2

Procedures for expression with "error recovery"



```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match (token) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number }) ;
  end if;
end exp ;
```

?

Also { +, - } ?

if token in {(,number} then ...

Main philosophy

The method "checkinput" is called twice: First to check that the construction starts correctly, and secondly to check that the symbol after the construction is legal.

Uses parameters, not a stack

The procedures must themselves resume execution at the right place inside themselves when they get the control back,

or it must terminate immediately if it cannot resume execution on the current symbol.

```
procedure factor ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    case token of
    ( : match( ( ) ;
        exp ( { ) } ) ;
        match( ) ) ;
    number :
        match(number) ;
    else error ;
    end case ;
    checkinput ( synchset, { (, number }) ;
  end if ;
end factor ;
```

Why not the full"synchset"?

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { $ }) do
    getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset ∪ followset ) ;
  end if ;
end;
```

27

4.7 Bottom-up parsing

Bottom-up parsing: intro

"R" stands for *right-most* derivation.

LR(0)
- only for very simple grammars
- approx. 300 states for standard programming languages
- only as warm-up for SLR(1) and LALR(1)

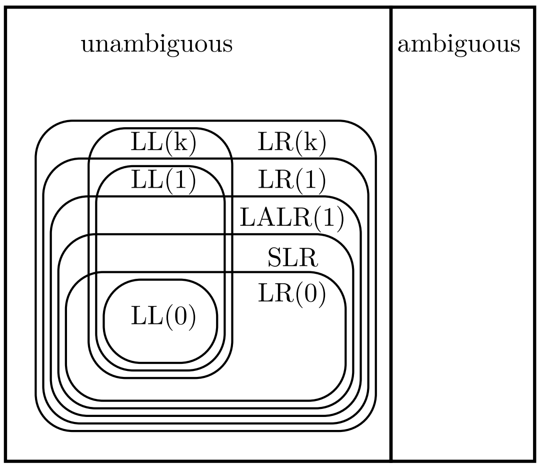
SLR(1)
- expressive enough for most grammars for standard PLs
- same number of states as LR(0)
- main focus here

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 parsed by looking at the next token

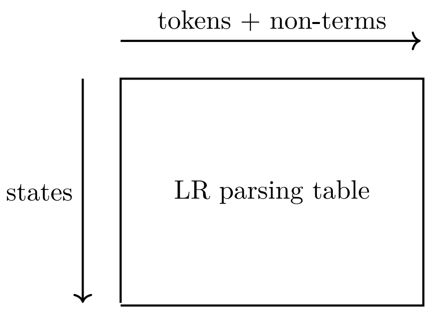
There might seem to be a contradiction in the explanation of LR(0): if LR(0) is so weak that it works only for unreasonably simple languages, why does the slides speaks about *standard* languages, and that LR(0) automata for those have 300 states, if one does not use LR(0)? The answer is, the other more expressive parsers (SLR(1) and LALR(1)) use the *same* construction of states, so that's why one can estimate the number of states, even if standard languages don't have an LR(0) parser; they may have an LALR(1)-parser, which has, it its core, LR(0)-states.

Grammar classes overview (again)



LR-parsing and its subclasses

- *right-most* derivation (but left-to-right parsing)
- in general: bottom-up: more powerful than top-down
- typically: tool-supported (unlike recursive descent, which may well be hand-coded)
- based on *parsing tables* + explicit *stack*
- thankfully: *left-recursion* no longer problematic
- typical tools: yacc and friends (like bison, CUP, etc.)
- another name: *shift-reduce* parser



Example grammar

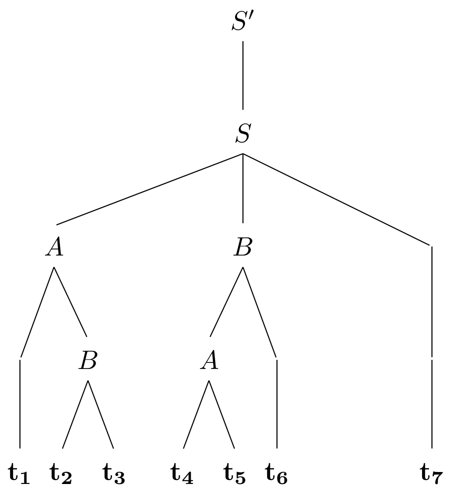
$$\begin{array}{rcl} S' & \to & S \\ S & \to & AB\mathbf{t_7} \mid \ldots \\ A & \to & \mathbf{t_4 t_5} \mid \mathbf{t_1} B \mid \ldots \\ B & \to & \mathbf{t_2 t_3} \mid A\mathbf{t_6} \mid \ldots \end{array}$$

- assume: grammar unambiguous
- assume word of terminals $\mathbf{t_1 t_2} \ldots \mathbf{t_7}$ and its (unique) parse-tree

- general agreement for bottom-up parsing:
  - start symbol *never* on the right-hand side of a production
  - **routinely add another "extra" start-symbol** (here S')

The fact that the start symbol *never* occurs on the right-hand side of a production will later be relied upon when constructing a DFA for “scanning” the stack, to control the reactions of the stack machine. This restriction leads to a unique, well-defined initial state. All goes just smoother (and the construction of the LR-automaton is slightly more straighforward) if one obeys that convention.

Parse tree for $\mathbf{t}_1 \ldots \mathbf{t}_7$



Remember: parse tree independent from left- or right-most-derivation

LR: left-to right scan, right-most derivation?

Potentially puzzling question at first sight:

what?: *right*-most derivation, when parsing *left*-to-right?

- short answer: parser builds the parse tree **bottom-up**
- derivation:
  - replacement of nonterminals by right-hand sides
  - *derivation*: builds (implicitly) a parse-tree *top-down*

- sentential form: word from Σ^* derivable from start-symbol

Right-sentential form: right-most derivation

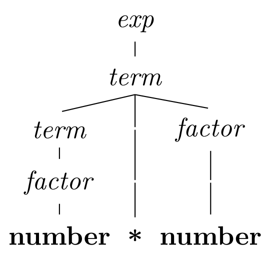
$$S \Rightarrow_r^* \alpha$$

Slighly longer answer

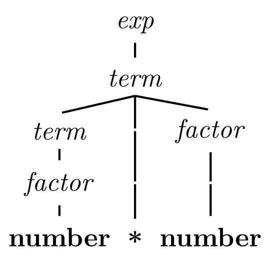
LR parser parses from left-to-right and builds the parse tree bottom-up. When doing the parse, the parser (implicitly) builds a *right-most* derivation **in reverse** (because of bottom-up).

Example expression grammar (from before)

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



Bottom-up parse: Growing the parse tree



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

The slides show in a series of overlays, how the parse-tree is growing, and at the same time, how the word **number** $*$ **number** is reduced step by step to the start symbol. That's the reverse direction compared to how one can use grammars to *derive* words and which corresponds to the direction of how top-down parsers work.

Reduction in reverse = right derivation

Reduction

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

Right derivation

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

- underlined part:
  - *different* in reduction vs. derivation
  - represents the "part being replaced"
    - for derivation: right-most non-terminal
    - for reduction: indicates the so-called **handle** (or part of it)
- consequently: all intermediate words are *right-sentential forms*

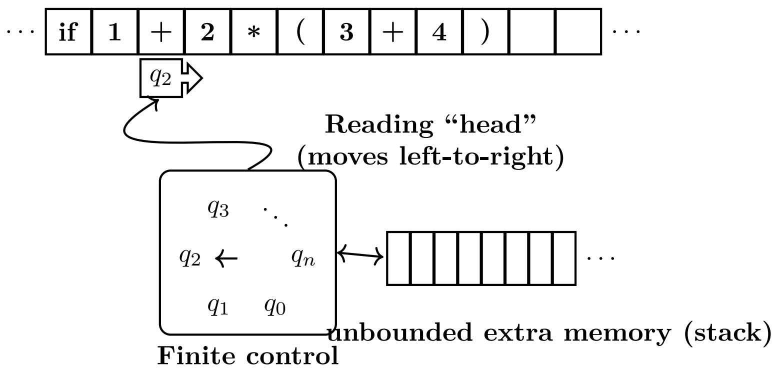
Handle

Definition 4.7.1 (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.

Note:

- w (right of a handle) contains only terminals
- w: corresponds to the future input still to be parsed!
- $\alpha\beta$ will correspond to the stack content (β the part touched by reduction step).
- the $\Rightarrow_r$ -derivation-step *in reverse*:
  - one **reduce**-step in the LR-parser-machine
  - adding (implicitly in the LR-machine) a new parent to children β (= **bottom-up**!)
- "handle"-part β can be *empty* (= $\boldsymbol{\epsilon}$)

Schematic picture of parser machine (again)



General LR "parser machine" configuration

- *stack*:
  - contains: terminals + non-terminals (+ **$**)
  - containing: what has been read already but not yet "processed"
- *position* on the "tape" (= token stream)
  - represented here as word of terminals *not yet read*
  - end of "rest of token stream": **$**, as usual
- *state* of the machine
  - in the following schematic illustrations: *not* yet part of the discussion
  - *later*: part of the parser table, currently we explain *without* referring to the state of the parser-engine
  - currently we assume: tree and rest of the input given
  - the trick ultimately will be: how do achieve the same *without that tree already given* (just parsing left-to-right)

Schematic run (reduction: from top to bottom)

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

2 basic steps: shift and reduce

- parsers reads input and uses stack as intermediate storage
- so far: no mention of look-ahead (i.e., action depending on the value of the next token(s)), but that may play a role, as well

Shift

Move the next input symbol (terminal) over to the top of the stack ("push")

Reduce

Remove the symbols of the *right-most* subtree from the stack and replace it by the non-terminal at the root of the subtree (replace = "pop + push").

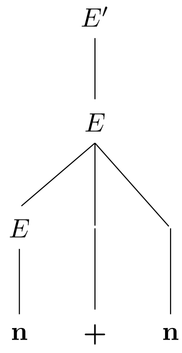
- decision *easy* to do **if one has the parse tree already**!
- *reduce* step: popped resp. pushed part = right- resp. left-hand side of handle

The remark that it's "easy to do" refers to something that is illustrated next: the question namely the decision-making process of the parser. should the parser do a shift or a reduce and if so, reduce with what rule. If one assumes the "target" parse-tree as already given (as we currently do in our presentation, for instance also in the following slides), then tree embodies those decisions. Ultimately, 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: LR parse for "+" (given the tree)

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array}$$

CST



Run

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

note: line 3 vs line 6!; both contain E on top of stack

(right) derivation: reduce-steps "in reverse"

$$\underline{E'} \Rightarrow \underline{E} \Rightarrow \underline{E} + \mathbf{n} \Rightarrow \mathbf{n} + \mathbf{n}$$

The example is supposed to shed light on how the machine can make decisions assuming that the tree is already given. For that, one should compare the situation in stage 3 and state 6. In both situations, the machine has *the same stack content* (containing only the end-marker and E on top of the stack). However, at stage 3, the machine does a shift, whereas in stage 6, it does a reduce.

Since the stack content (representing the "past" of the parse, i.e., the already processed input) is the identical in both cases, the parser machine is necessarily *in the same state* in both stages, which mean, 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 cannot be used to make the distinction, one takes the "future" input. Maybe not all of it, but part of it. That's a form of a look-ahead (that will *not* yet be done for LR(0), as that for is without look-ahead).

Example with ϵ-transitions: parentheses

$$\begin{array}{rcl} S' & \to & S \\ S & \to & (\,S\,)\,S \mid \epsilon \end{array}$$

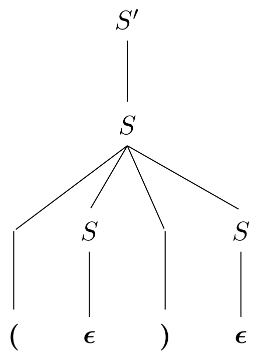
side remark: unlike previous grammar, here:

- production with *two* non-terminals on the right

⇒ difference between left-most and right-most derivations (and mixed ones)

Parentheses: run and right-most derivation

CST



Run

	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

Note: the 2 reduction steps for the ϵ productions

Right-most derivation and right-sentential forms

$$\underline{S'} \Rightarrow_r \underline{S} \Rightarrow_r (\, S \,)\, \underline{S} \Rightarrow_r (\, \underline{S} \,) \Rightarrow_r (\ )$$

Right-sentential forms & the stack

- sentential form: word from Σ^* derivable from start-symbol

Right-sentential form: right-most derivation

$$S \Rightarrow_r^* \alpha$$

- right-sentential forms:
  - part of the "run"
  - but: **split** between *stack* and *input*

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

$$\underline{E'} \Rightarrow_r \underline{E} \Rightarrow_r \underline{E + \mathbf{n}} \Rightarrow_r \mathbf{n} + \mathbf{n}$$

$$\underline{\mathbf{n}} + \mathbf{n} \hookrightarrow \underline{E + \mathbf{n}} \hookrightarrow \underline{E} \hookrightarrow E'$$

$$\underline{E'} \Rightarrow_r \underline{E} \Rightarrow_r \underline{E} + \mathbf{n} \mid \sim\ \underline{E} + \mid \mathbf{n} \sim \underline{E} \mid + \mathbf{n} \Rightarrow_r \mathbf{n} \mid\ + \mathbf{n} \sim\mid \mathbf{n} + \mathbf{n}$$

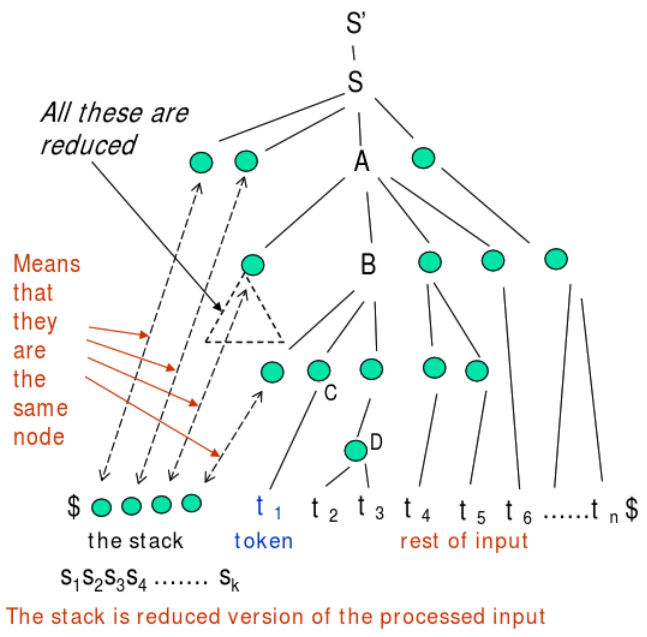
The $\mid$ here is introduced as "ad-hoc" notation to illustrate the separation between the parse stack on the left and the future input on the right.

Viable prefixes of right-sentential forms and handles

- right-sentential form: $E + \mathbf{n}$
- **viable prefixes** of RSF
  - prefixes of that RSF *on the stack*
  - here: 3 viable prefixes of that RSF: E, $E+$, $E + \mathbf{n}$
- *handle*: remember the definition earlier
- here: for instance in the sentential form $\mathbf{n} + \mathbf{n}$
  - handle is production $E \to \mathbf{n}$ on the *left* occurrence of $\mathbf{n}$ in $\mathbf{n} + \mathbf{n}$ (let's write $\mathbf{n}_1 + \mathbf{n}_2$ for now)
  - note: in the stack machine:
    * the left $\mathbf{n}_1$ on the stack
    * rest $+\,\mathbf{n}_2$ on the input (unread, because of LR(0))

- if the parser engine detects handle $\mathbf{n}_1$ on the stack, it does a *reduce*-step
- However (later): reaction depends on current *state* of the parser engine

A typical situation during LR-parsing


S'
S
A
B
C
D
All these are reduced
Means that they are the same node
$ t 1 t 2 t 3 t 4 t 5 t 6 ......t n $
the stack
token
rest of input
s1s2s3s4 .... sk
The stack is reduced version of the processed input

After a shift, the next reduction to be made is a reduction with the production:

C -> t1

Then, after two shifts, we will make a reduction with the production:

D -> t2 t3

Then, what's next?

General design for an LR-engine

- some ingredients clarified up-to now:
  - bottom-up tree building as reverse right-most derivation,
  - stack vs. input,
  - shift and reduce steps
- however: 1 ingredient missing: next step of the engine may depend on
  - top of the stack ("handle")
  - look ahead on the input (but not for LL(0))
  - and: current **state** of the machine (same stack-content, but different reactions at different stages of the parse)

But what are the states of an LR-parser?

General idea:

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

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

is **regular**!

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

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

LR(0) parsing as easy pre-stage

- LR(0): in practice *too simple*, but easy conceptual step towards LR(1), SLR(1) etc.
- LR(1): in practice good enough, LR(k) not used for $k > 1$
- to build the automaton: **LR(0)-items**

LR(0) parsing is introduced as easy pre-stage for the more expressive forms of bottom-up parsing later. In itself, it's not expressive enough to be practivally useful. But the construction underlies directly or at least conceptually the more complex parser constructions to come. In particular: for LR(0) parsing, the core of the construction is the so-called LR(0)-DFA, based on LR(0)-items. This construction is directly also used for SLR-parsing. For LR(1) and LALR(1), the construction of the corresponding DFA is not identical, but analogous to the construction of LR(0)-DFA.

LR(0) items

LR(0) item

production with specific "parser position" **.** in its right-hand side

- $.$: “meta-symbol” (not part of the production)

LR(0) item for a production $A \to \beta\gamma$

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

- item with dot at the beginning: *initial* item
- item with dot at the end: *complete* item

Example: items of LR-grammar

Next two examples. They should make the concept of items clear enough. The only point to keep in mind is the treatment of the ϵ symbol.

Grammar for parentheses: 3 productions

$$\begin{array}{rcl} S' & \to & S \\ S & \to & (\,S\,)\,S \mid \epsilon \end{array}$$

8 items

$$\begin{array}{rcl} S' & \to & .S \\ S' & \to & S. \\ S & \to & .(\,S\,)\,S \\ S & \to & (\,.S\,)\,S \\ S & \to & (\,S.\,)\,S \\ S & \to & (\,S\,).S \\ S & \to & (\,S\,)\,S. \\ S & \to & . \end{array}$$

- $S \to \epsilon$ gives $S \to .$ as item (not $S \to \epsilon.$ and $S \to .\epsilon$)

As a side remark for later: it will turn out: grammar is *not LR(0)*.

Another example: items for addition grammar

Grammar for addition: 3 productions

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array}$$

(coincidentally also:) 8 items

$$\begin{array}{lcl}
E' & \to & .E \\
E' & \to & E. \\
E & \to & .E+\mathbf{n} \\
E & \to & E.+\mathbf{n} \\
E & \to & E+.\mathbf{n} \\
E & \to & E+\mathbf{n}. \\
E & \to & .\mathbf{n} \\
E & \to & \mathbf{n}.
\end{array}$$

Also here, it will turn out: *not an LR(0)* grammar

Finite automata of items

- general set-up: *items* as **states in an automaton**
- automaton: "operates" *not* on the input, **but the stack**
- automaton either
  - first NFA, afterwards made deterministic (subset construction), or
  - directly DFA

States formed of sets of items

In a state marked by/containing item

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

- β on the *stack*
- γ: to be treated next (terminals on the input, but can contain also non-terminals(!))

The explanation of what the items as state of the automaton means is conceptual. One piece may be (at the current point) a bit mysterious, resp. does not quite fit: the fact that the γ can contain non-terminals. We come to that soon, and we will see later in examples, what happens.

State transitions of the NFA

- $X \in \Sigma$
- two kinds of transitions

Terminal or non-terminal

$$\begin{tikzcd}
\boxed{A \to \alpha.X\eta} \arrow[r, "X"] & \boxed{A \to \alpha X.\eta}
\end{tikzcd}$$

ϵ $(X \to \beta)$

$$\begin{tikzcd}\boxed{A \to \alpha.X\eta} \arrow[r, "\epsilon"] & \boxed{X \to .\beta}\end{tikzcd}$$

- In case $X = terminal$ (i.e. token) =
  - the left step corresponds to a **shift** step
- for non-terminals (see next slide):
  - interpretation more complex: non-terminals are officially never on the input
  - note: in that case, item $A \to \alpha.X\eta$ has two (kinds of) outgoing transitions

Explanations

We have explained *shift* steps so far as: parser eats one *terminal* (= input token) and pushes it on the stack.

Transitions for non-terminals and ϵ

- so far: we never pushed a non-terminal from the input to the stack, we **replace** in a **reduce**-step the right-hand side by a left-hand side
- but: replacement in a **reduce** steps can be seen as
  1. pop right-hand side off the stack,
  2. instead, "assume" corresponding non-terminal on input,
  3. eat the non-terminal an push it on the stack.
- two kinds of transitions
- assume production $X \to \beta$ and *initial* item $X \to .\beta$

Transitions (repeated)

Terminal or non-terminal

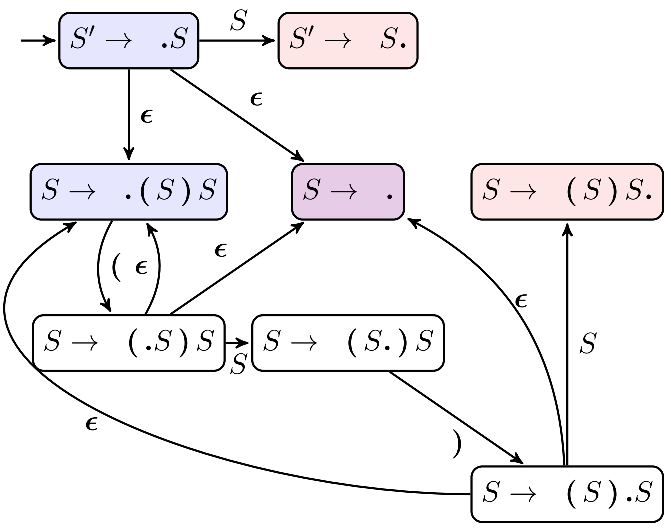
$$\begin{tikzcd}\boxed{A \to \alpha.X\eta} \arrow[r, "X"] & \boxed{A \to \alpha X.\eta}\end{tikzcd}$$

Epsilon (X: non-terminal here)

Given production $X \to \beta$:

$$\begin{tikzcd}\boxed{A \to \alpha.X\eta} \arrow[r, "\epsilon"] & \boxed{X \to .\beta}\end{tikzcd}$$

NFA: parentheses



In the figure, we use colors for illustration, only, i.e., they are not officially part of the construction. The colors are intended to represent the following:

- "reddish": complete items
- "blueish": init-item (less important)
- "violet'ish": both.

Furthermore, you may notice for the *initial items* and *complete items*:

- one initial item state per production of the grammar
- initial items is where the ϵ-transisitions go into, but *with exception* of the initial state (with S'-production)
- no outgoing edges from the complete items.

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.

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

Initial and final states

initial states:

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

⇒ initial item $S' \to .S$ as (only) **initial state**

final states:

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

- input must be empty
- stack must be empty except the (new) start symbol
- NFA has a word to say about acceptence
  - but *not* in form of being in an accepting state
  - so: no accepting *states*
  - but: accepting *action* (see later)

The NFA (or later DFA) has a specific task, it is used to "scan" the *stack* (at least conceptually), not the input. The automaton is not so much for *accepting* a stack and then stop, it's more like determining the state that corresponds to the current stack content. Therefore there are no accepting states in the sense of a FSA!

NFA: addition

$$\begin{tikzcd}
\arrow[r] & E' \to .E \arrow[r, "E"] \arrow[d, "\epsilon"] \arrow[rd, "\epsilon"] & E' \to E. & \\
& E \to .E+\mathbf{n} \arrow[loop left, "\epsilon"] \arrow[r, "\epsilon"] \arrow[d, "E"] & E \to .\mathbf{n} \arrow[r, "\mathbf{n}"] & E \to \mathbf{n}. \\
& E \to E.+\mathbf{n} \arrow[r, "+"'] & E \to E+.\mathbf{n} \arrow[r, "\mathbf{n}"'] & E \to E+\mathbf{n}.
\end{tikzcd}$$

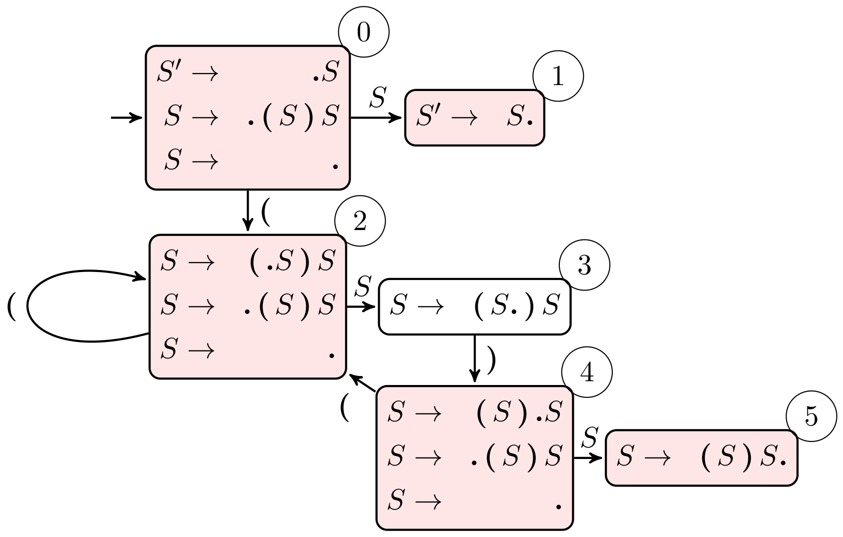
Determinizing: from NFA to DFA

- standard subset-construction[7]
- states then contain *sets* of items
- important: ϵ-closure
- also: *direct* construction of the DFA possible

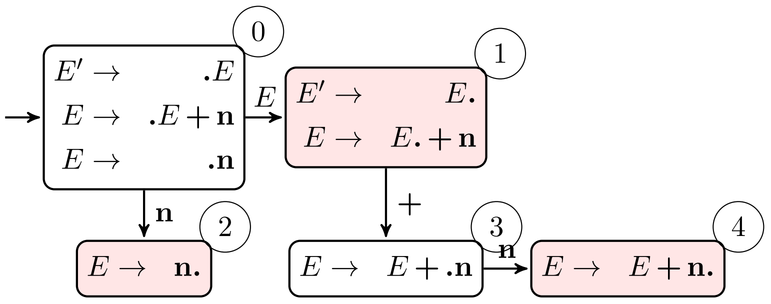
In the following two slides, we show the DFAs corresponding to the NFAs shown before. For the construction on how to determinize NFAs (and minimize them), we refer to the corresponding sections in the chapter about lexing. Anyway, we will afterwards also look at a *direct* construction of the DFA (without the detour over NFAs). That will result in the same automata anyway.

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

DFA: parentheses



DFA: addition



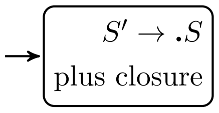
Direct construction of an LR(0)-DFA

- quite easy: just build in the closure directly…

ϵ-closure

- if $A \to \alpha.B\gamma$ is an item in a state where
- there are productions $B \to \beta_1 \mid \beta_2 \ldots$ then
- add items $B \to .\beta_1$, $B \to .\beta_2$ … to the state
- continue that process, until saturation

initial state



Direct DFA construction: transitions

$$\begin{tikzcd} \boxed{\begin{array}{ll} \ldots & \\ A_1 \to & \alpha_1.X\beta_1 \\ \ldots & \\ A_2 \to & \alpha_2.X\beta_2 \\ \ldots & \end{array}} \arrow[r, "X"] & \boxed{\begin{array}{lr} A_1 \to & \alpha_1X.\beta_1 \\ A_2 \to & \alpha_2X.\beta_2 \\ & \text{plus closure} \end{array}} \end{tikzcd}$$

- X: terminal or non-terminal, both treated uniformely
- *All* items of the form $A \to \alpha.X\beta$ must be included in the post-state
- and all others (indicated by "...") in the pre-state: not included

One can e-check the previous examples (first doing the NFA, then the DFA): the outcome is the same.

How does the DFA do the shift/reduce and the rest?

- we have seen: bottom-up parse tree generation
- we have seen: shift-reduce and the stack vs. input
- we have seen: the construction of the DFA

But: how does it hang together?

We need to interpret the "set-of-item-states" in the light of the stack content and figure out the **reaction** in terms of

- transitions in the automaton
- stack manipulations (shift/reduce)
- acceptance
- input (apart from shifting) not relevant when doing LR(0)

and the reaction better be uniquely determined ....

Stack contents and state of the automaton

- remember: at any config. of stack/input in a run
  1. stack contains words from Σ^*
  2. DFA operates deterministically on such words
- the stack contains "abstraction of the past":
- when feeding that "past" on the stack into the automaton
  - starting with the oldest symbol (not in a LIFO manner)
  - starting with the DFA's initial state

  $\Rightarrow$ stack content **determines** state of the DFA
- actually: each prefix also determines uniquely a state
- **top state**:

  - state after the complete stack content
  - corresponds to the **current** state of the stack-machine

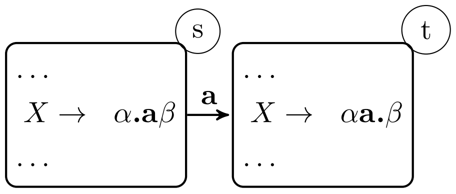
$\Rightarrow$ crucial when determining *reaction*

State transition allowing a shift

- assume: top-state (= current state) contains item

$$X \to \alpha.\mathbf{a}\beta$$

- construction thus has transition as follows


s
t
...
$X \to \quad \alpha.\mathbf{a}\beta$
...
a
...
$X \to \quad \alpha\mathbf{a}.\beta$
...

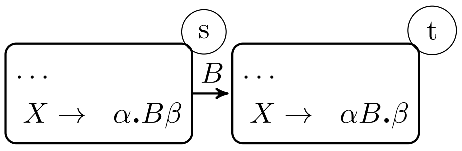
- shift is possible
- if shift is *the* correct operation and **a** is terminal symbol corresponding to the current token: state afterwards = t

State transition: analogous for non-term's

Production

$$X \to \alpha.B\beta$$

Transition



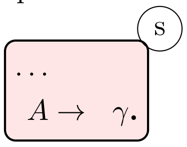
Rest

- "goto = shift for non-terms"
- intuition: "second half of a reduce step"
- same as before, now with non-terminal B

- note: we never read non-term from input
- not officially called a shift
- corresponds to the reaction **followed by** a *reduce* step, it's **not** the reduce step itself
- think of the reduce
  - not as: *replace* on top of the stack the handle (right-hand side) by non-term B,
  - but instead as:
    1. pop off the handle from the top of the stack
    2. put the non-term B "back onto the input" (corresponding to the above state s)
    3. eat the B and *"shift"* it to the stack
- later: a **goto** reaction in the parse table

State (not transition) where a reduce is possible

- remember: *complete items*
- assume **top state** s containing complete item $A \to \gamma.$



- a complete right-hand side ("handle") γ on the stack and thus done
- may be replaced by right-hand side A

⇒ reduce step
- builds up (implicitly) new parent node A in the bottom-up procedure
- **Note**: A on top of the stack instead of γ:
  - **new top state**!
  - remember the "goto-transition" (shift of a non-terminal)

A conceptual picture for the reduce step is as follows. As said, we remove the handle from the stack, and "pretend", as if the A is next on the input, and thus we "shift" it on top of the stack, doing the corresponding A-transition.

Remarks: states, transitions, and reduce steps

- ignoring the ϵ-transitions (for the NFA)
- there are 2 "kinds" of transitions in the DFA
  1. terminals: reals shifts
  2. non-terminals: "following a reduce step"

No edges to represent (all of) a reduce step!

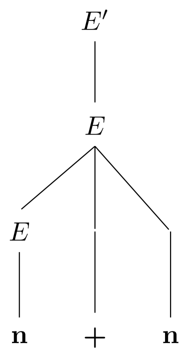
- if a reduce happens, parser engine *changes state*!
- however: this state change is **not** represented by a transition in the DFA (or NFA for that matter)
- especially *not* by outgoing errors of completed items

- if the (rhs of the) handle is *removed* from top stack ⇒
  - "go back to the (top) state before that handle had been added": *no edge for that*
- later: stack notation simply remembers the state as part of its configuration

Example: LR parsing for addition (given the tree)

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array}$$

CST



Run

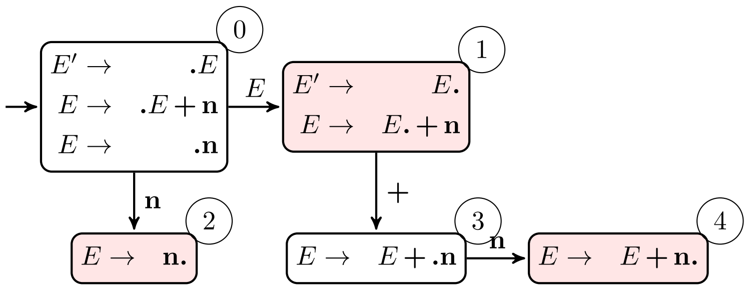
	parse stack	input	action
1	$\$$	$\mathbf{n} + \mathbf{n}\,\$$	shift
2	$\$\,\mathbf{n}$	$+\,\mathbf{n}\,\$$	red:. $E \to \mathbf{n}$
3	$\$\,E$	$+\,\mathbf{n}\,\$$	shift
4	$\$\,E +$	$\mathbf{n}\,\$$	shift
5	$\$\,E + \mathbf{n}$	$\$$	reduce $E \to E + \mathbf{n}$
6	$\$\,E$	$\$$	red.: $E' \to E$
7	$\$\,E'$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

note: line 3 vs line 6!; both contain E on top of stack

This is a revisit of an example resp. slide from earlier, when we discussed how a parser can do decisions, resp. that it would be easy to do decisions for the parser machine if it had the tree already. Unfortunately it has the tree not available, the only thing it has is "the past" which is represented (partially) by the stack content. As discussed earlier, interesting in the run are stage 3 and state 6, which have the same stack content, which also means, the parser is in the same state of its LR(0)-DFA. With the automaton constructed as before, that's state 1. The state 1 is important, as it illustrates a shift/reduce conflict. Remember: reduce-steps are *not* represented in the LR(0)-automaton via *transitions*. They are only implicitly represented by *complete items*. Thus, as shift-reduce conflict is not characterized by 2 outgoing edges. It's one **outgoing edge from a state containing a complete item**.

Earlier we hinted at that an automaton could make decisions based on a look-head. That is not yet done: the LR(0), in state 1 especially, can do a reduce step or a shift step, which constitutes the conflict. Later, we will see under which circumstances, looking at the "next symbol" can help to make the decision. That leads to SLR parsing (or even later to LR(1)/LALR(1)). In the particular situation of state 1 in the example, the next possible symbol would be $+$ or else **$**

DFA of addition example



- note line 3 vs. line 6
- both stacks $= E \Rightarrow$ same (top) state in the DFA (state 1)

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

LR(0) grammars

LR(0) grammar

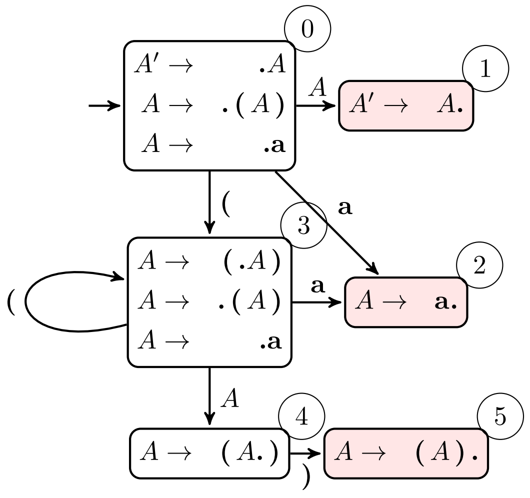
The top-state alone determines the next step.

- especially: no shift/reduce conflicts in the form shown
- thus: previous addition-grammar is *not LR(0)*

Simple parentheses

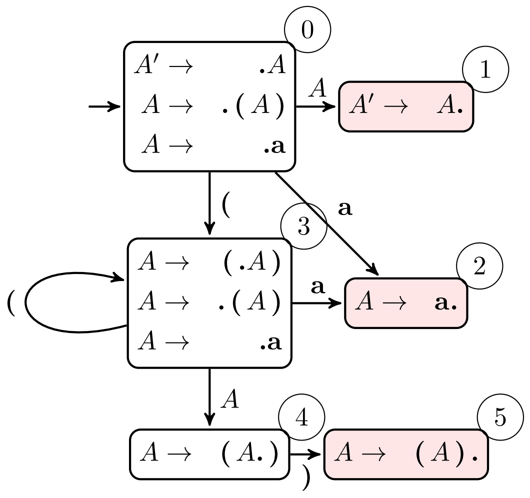
$$A \to (\,A\,) \mid \mathbf{a}$$

DFA



Simple parentheses is LR(0)

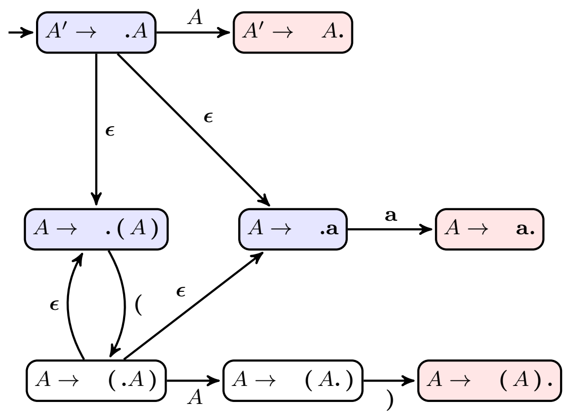
DFA



Remarks

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

NFA for simple parentheses (bonus slide)



For completeness sake: that's the NFA for the "simple parentheses".

Parsing table for an LR(0) grammar

- table structure: slightly different for SLR(1), LALR(1), and LR(1) (see later)
- note: the "goto" part: "shift" on non-terminals (only 1 non-terminal A here)
- corresponding to the A-labelled transitions

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	$A \to (A)$				

Parsing of ((a))

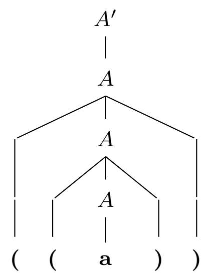
stage	parsing stack	input	action
1	$\$_0$	$((\mathbf{a}))\$$	shift
2	$\$_0(_3$	$(\mathbf{a}))\$$	shift
3	$\$_0(_3(_3$	$\mathbf{a}))\$$	shift
4	$\$_0(_3(_3\mathbf{a}_2$	$))\$$	reduce $A \to \mathbf{a}$
5	$\$_0(_3(_3A_4$	$))\$$	shift
6	$\$_0(_3(_3A_4)_5$	$)\$$	reduce $A \to (A)$
7	$\$_0(_3A_4$	$)\$$	shift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

- note: stack on the left
  - contains top *state* information
  - in particular: overall **top** state on the right-most end
- note also: **accept** action
  - reduce wrt. to $A' \to A$ and
  - *empty stack* (apart from **\$**, A, and the state annotation)

  $\Rightarrow$ accept

The left-most column is just line numbers ("stage" of the computation), it's not the *state*.

Parse tree of the parse



- As said:
  - the reduction "contains" the parse-tree
  - reduction: builds it bottom up
  - reduction in reverse: contains a *right-most* derivation (which is "top-down")
- accept action: corresponds to the parent-child edge $A' \to A$ of the tree

Parsing of erroneous input

- empty slots it the table: "errors"

stage	parsing stack	input	action
1	$\$_0$	$((\mathbf{a})\$$	shift
2	$\$_0(_3$	$(\mathbf{a})\$$	shift
3	$\$_0(_3(_3$	$\mathbf{a})\$$	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	$\$_0(_3A_4$	$\$$	????

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

Invariant

important general invariant for LR-parsing: never shift something "illegal" onto the stack

LR(0) parsing algo, given DFA

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:

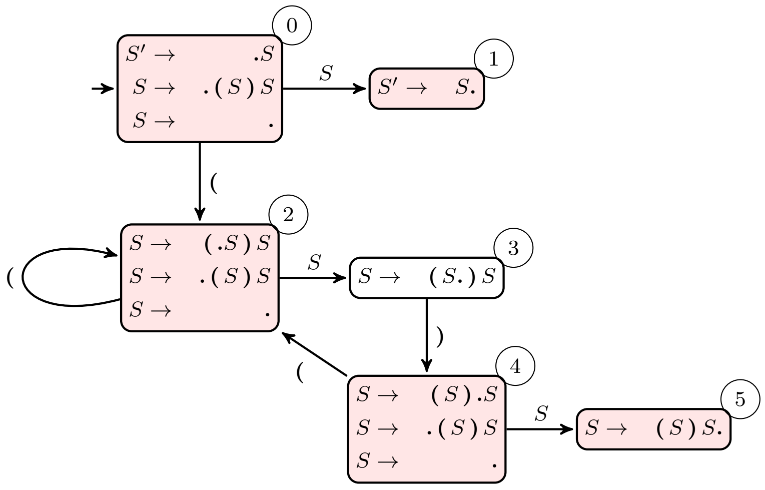
     pop: remove γ (including "its" states from the stack)

     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)

DFA parentheses again: LR(0)?

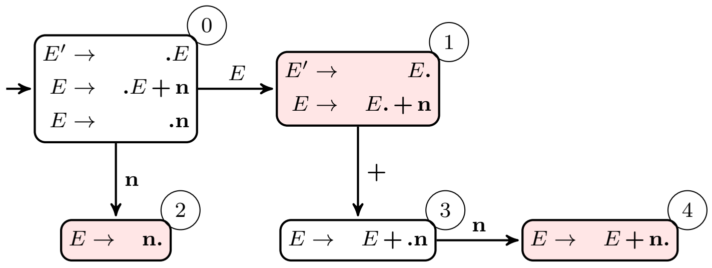
$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & (\,S\,)\,S \mid \epsilon \end{array}$$



Look at states 0, 2, and 4

DFA addition again: LR(0)?

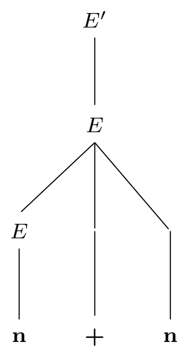
$$\begin{array}{rcl} E' & \rightarrow & E \\ E & \rightarrow & E + \mathbf{n} \mid \mathbf{n} \end{array}$$



How to make a decision in state 1?

Decision? If only we knew the ultimate tree already (expecially the parts still to come)...

CST



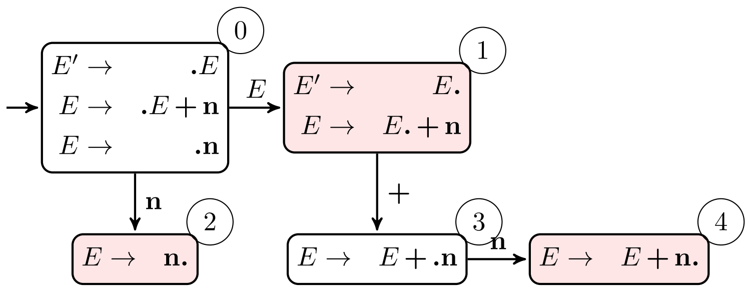
Run

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

- current stack: represents already known part of the parse tree
- since we don't have the future parts of the tree yet:

⇒ **look-ahead** on the input (without building the tree yet)
- LR(1) and its variants: *look-ahead of 1* (= look at the current type of the token)

Addition grammar (again)



- *How to make a decision in state* 1*?* (here: shift vs. reduce)

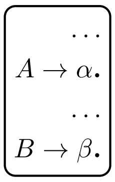
⇒ look at the next input symbol (in the token)

One look-ahead

- LR(0), not useful, too weak
- add look-ahead, here of *1 input symbol* (= token)
- different variations of that idea (with slight difference in expresiveness)
- tables slightly changed (compared to LR(0))
- but: *still* can use the LR(0)-DFAs

Resolving LR(0) reduce/reduce conflicts

LR(0) reduce/reduce conflict:



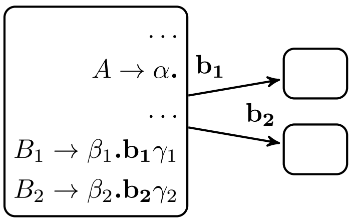
SLR(1) solution: use follow sets of non-terms

- If $\mathit{Follow}(A) \cap \mathit{Follow}(B) = \emptyset$

⇒ next symbol (in `token`) decides!
  - if `token` $\in \mathit{Follow}(\alpha)$ then reduce using $A \to \alpha$
  - if `token` $\in \mathit{Follow}(\beta)$ then reduce using $B \to \beta$
  - ...

Resolving LR(0) shift/reduce conflicts

LR(0) shift/reduce conflict:


...
$A \to \alpha.$
...
$B_1 \to \beta_1.\mathbf{b_1}\gamma_1$
$B_2 \to \beta_2.\mathbf{b_2}\gamma_2$
$\mathbf{b_1}$
$\mathbf{b_2}$

SLR(1) solution: again: use follow sets of non-terms

- If $\mathit{Follow}(A) \cap \{\mathbf{b_1}, \mathbf{b_2}, \ldots\} = \emptyset$

⇒ next symbol (in `token`) decides!
  - if `token` $\in \mathit{Follow}(A)$ then *reduce* using $A \to \alpha$, non-terminal A determines new top state
  - if `token` $\in \{\mathbf{b_1}, \mathbf{b_2}, \ldots\}$ then *shift*. Input symbol $\mathbf{b_i}$ determines new top state
  - ...

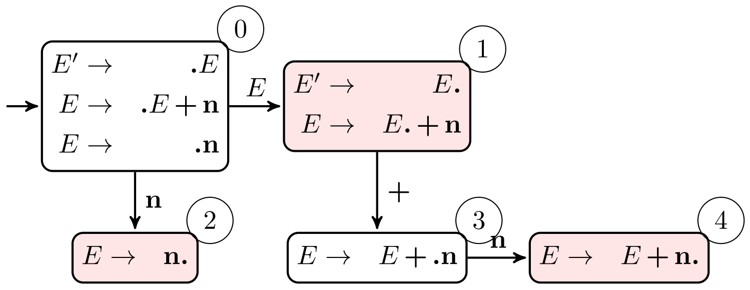
SLR(1) requirement on states (as in the book)

- formulated as conditions on the states (of LR(0)-items)
- given the LR(0)-item DFA as defined

SLR(1) condition, on all states s

1. For any item $A \to \alpha.X\beta$ in s with X a *terminal*, there is no **complete** item $B \to \gamma.$ in s with $X \in \mathit{Follow}(B)$.
2. For any **two complete** items $A \to \alpha.$ and $B \to \beta.$ in s, $\mathit{Follow}(\alpha) \cap \mathit{Follow}(\beta) = \emptyset$

Revisit addition one more time



- $\mathit{Follow}(E') = \{\$\}$

$\Rightarrow$
  - shift for $+$
  - reduce with $E' \to E$ for $\$$ (which corresponds to accept, in case the input is empty)

SLR(1) algo

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 $s \xrightarrow{X} t$[8]
2. s contains a *complete* item (say $A \to \gamma.$) **and the next token in the input is in** $\mathit{Follow}(A)$: *reduce* by rule $A \to \gamma$:
   - A reduction by $S' \to S$: *accept*, if input is empty[9]
   - else:

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

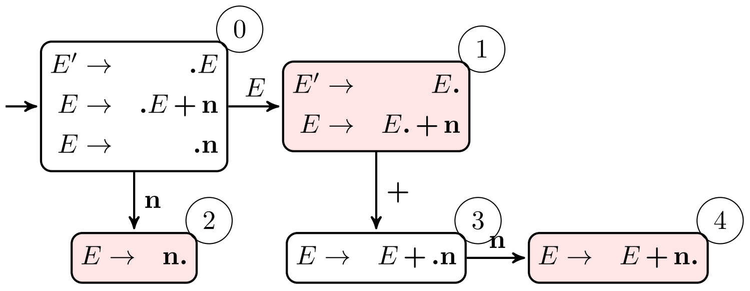
     back up: assume to be in state u which is *now* head state

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

[8] Cf. to the LR(0) algo: since we checked the existence of the transition before, the else-part is missing now.

[9] Cf. to the LR(0) algo: This happens *now* only if next token is $\$$. Note that the follow set of S' in the *augmented* grammar is always only $\$$

Parsing table for SLR(1)



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

for state 2 and 4: $\mathbf{n} \notin \mathit{Follow}(E)$

Parsing table: remarks

- SLR(1) parsing table: rather similar-looking to the LR(0) one
- differences: reflect the differences in: LR(0)-algo vs. SLR(1)-algo
- same number of rows in the table (= same number of states in the DFA)
- only: colums "arranged" differently
  - LR(0): each state **uniformely**: either shift or else reduce (with given rule)
  - now: non-uniform, **dependent** on the input. But that does not apply to the previous example. We'll see that in the next, then.
- it should be obvious:
  - SLR(1) may resolve LR(0) conflicts
  - but: if the follow-set conditions are not met: SLR(1) *shift-shift* and/or SLR(1) *shift-reduce* conflicts
  - would result in non-unique entries in SLR(1)-table[10]

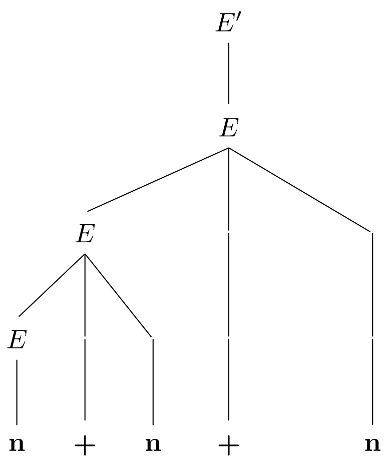
SLR(1) parser run (= "reduction")

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

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

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

Corresponding parse tree



Revisit the parentheses again: SLR(1)?

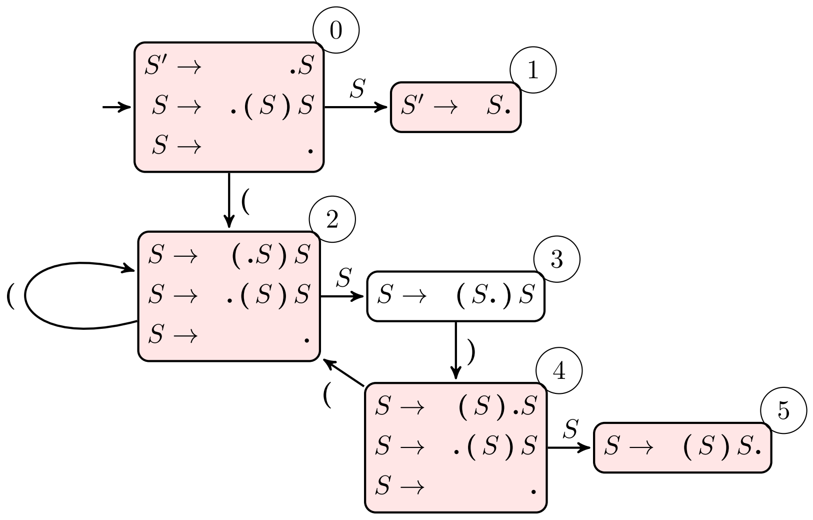
Grammar: parentheses

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

Follow set

$Follow(S) = \{), \$\}$

DFA for parentheses



SLR(1) parse table

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		$r: S \to (S)S$	$r: S \to (S)S$	

Parentheses: SLR(1) parser run (= "reduction")

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		$r: S \to (S)S$	$r: S \to (S)S$	

stage	parsing stack	input	action
1	$\$_0$	$(\,)\,(\,)\,\$$	shift: 2
2	$\$_0(_2$	$)\,(\,)\,\$$	reduce: $S \to \epsilon$
3	$\$_0(_2S_3$	$)\,(\,)\,\$$	shift: 4
4	$\$_0(_2S_3)_4$	$(\,)\,\$$	shift: 2
5	$\$_0(_2S_3)_4(_2$	$)\,\$$	reduce: $S \to \epsilon$
6	$\$_0(_2S_3)_4(_2S_3$	$)\,\$$	shift: 4
7	$\$_0(_2S_3)_4(_2S_3)_4$	$\$$	reduce: $S \to \epsilon$
8	$\$_0(_2S_3)_4(_2S_3)_4S_5$	$\$$	reduce: $S \to (\,S\,)\,S$
9	$\$_0(_2S_3)_4S_5$	$\$$	reduce: $S \to (\,S\,)\,S$
10	$\$_0S_1$	$\$$	accept

Remarks

Note how the stack grows, and would continue to grow if the sequence of ( ) would continue. That's characteristic for a right-recursive formulation of rules, and may constitute a problem for LR-parsing (stack-overflow).

Ambiguity & LR-parsing

- LR(k) (and LL(k)) grammars: *unambiguous*
- definition/construction: free of shift/reduce and reduce/reduce conflict (given the chosen level of look-ahead)
- However: ambiguous grammar tolerable, if (remaining) conflicts can be solved "meaningfully" otherwise:

Additional means of disambiguation:

1. by specifying associativity / precedence "externally"
2. by "living with the fact" that LR parser (commonly) *prioritizes shifts over reduces*

- for the second point ("let the parser decide according to its preferences"):
  - use sparingly and cautiously
  - typical example: *dangling-else*
  - even if parsers makes a decision, programmar may or may not "understand intuitively" the resulting parse tree (and thus AST)
  - grammar with many S/R-conflicts: go back to the drawing board

Example of an ambiguous grammar

$$\begin{array}{rcl} \textit{stmt} & \to & \textit{if-stmt} \mid \textbf{other} \\ \textit{if-stmt} & \to & \textbf{if}\ (\ \textit{exp}\ )\ \textit{stmt} \\ & \mid & \textbf{if}\ (\ \textit{exp}\ )\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\ \textit{exp} & \to & \textbf{0} \mid \textbf{1} \end{array}$$

In the following, E for *exp*, etc.

Simplified conditionals

Simplified "schematic" if-then-else

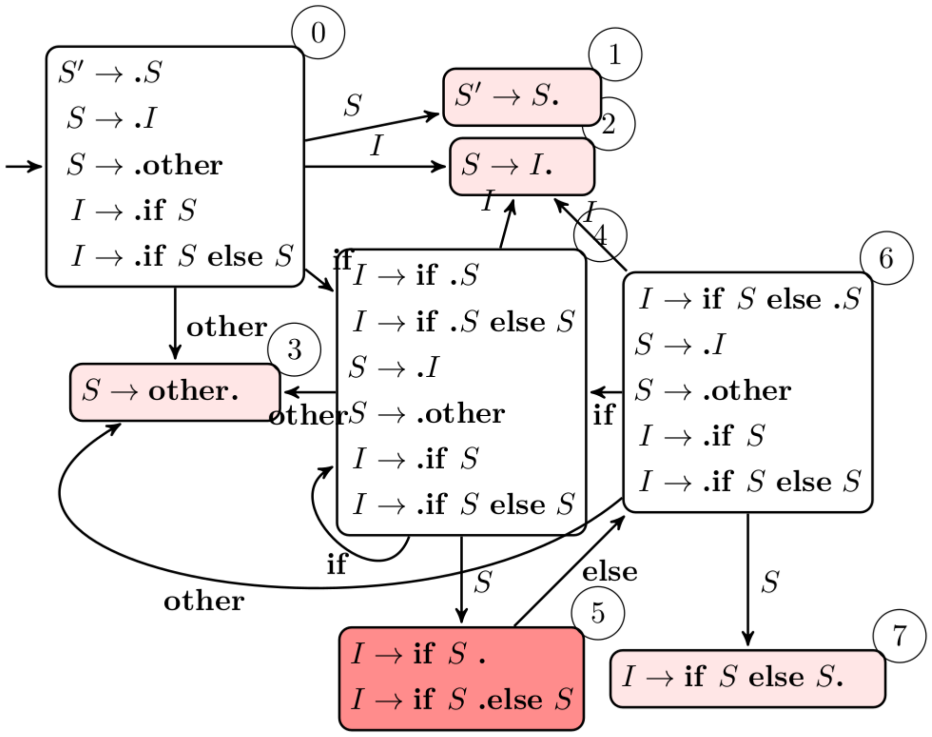
$$\begin{array}{lcl} S & \to & I \mid \textbf{other} \\ I & \to & \textbf{if}\ S \mid \textbf{if}\ S\ \textbf{else}\ S \end{array}$$

Follow-sets

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

- since ambiguous: at least one conflict must be somewhere

DFA of LR(0) items


0
$S' \to .S$
$S \to .I$
$S \to .\textbf{other}$
$I \to .\textbf{if}\ S$
$I \to .\textbf{if}\ S\ \textbf{else}\ S$
1
$S' \to S.$
2
$S \to I.$
3
$S \to \textbf{other}.$
4
$I \to \textbf{if}\ .S$
$I \to \textbf{if}\ .S\ \textbf{else}\ S$
$S \to .I$
$S \to .\textbf{other}$
$I \to .\textbf{if}\ S$
$I \to .\textbf{if}\ S\ \textbf{else}\ S$
5
$I \to \textbf{if}\ S\ .$
$I \to \textbf{if}\ S\ .\textbf{else}\ S$
6
$I \to \textbf{if}\ S\ \textbf{else}\ .S$
$S \to .I$
$S \to .\textbf{other}$
$I \to .\textbf{if}\ S$
$I \to .\textbf{if}\ S\ \textbf{else}\ S$
7
$I \to \textbf{if}\ S\ \textbf{else}\ S.$
S
I
if
other
else

Checking the previously shown conditions for SLR(1) parsing, one sees that there is a SLR(1) conflict in state 5: the follow-set of I contains **else**. In the following tables, *only* the shift-reaction is added in the corresponding slot (not both shift and reduce action), since that is default reaction of a parser tool, when facing a shift-reduce conflict.

Simple conditionals: parse table

Grammar

$$
\begin{array}{rcll}
S & \rightarrow & I & (1) \\
  & \mid & \textbf{other} & (2) \\
I & \rightarrow & \textbf{if}\ S & (3) \\
  & \mid & \textbf{if} S\ \textbf{else}\ S & (4)
\end{array}
$$

SLR(1)-parse-table, conflict "resolved"

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		$r:4$		$r:4$		

- *shift-reduce conflict* in state 5: reduce with *rule 3* vs. shift (to state 6)
- conflict there: **resolved** in favor of *shift* to 6
- note: extra start state left out from the table

Parser run (= reduction)

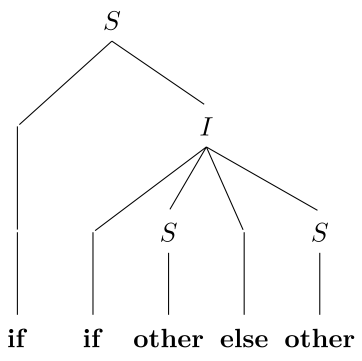
Parser run, different choice

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		$r:4$		$r:4$		

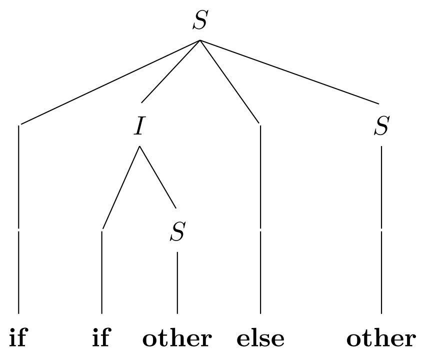
stage	parsing stack	input	action
1	$\$_0$	**if if other else other $**	shift: 4
2	$\$_0$**if**$_4$	**if other else other $**	shift: 4
3	$\$_0$**if**$_4$**if**$_4$	**other else other $**	shift: 3
4	$\$_0$**if**$_4$**if**$_4$**other**$_3$	**else other $**	reduce: 2
5	$\$_0$**if**$_4$**if**$_4 S_5$	**else 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

Parse trees for the "simple conditions"

shift-precedence: conventional



"wrong" tree



standard "dangling else" convention

"an **else** belongs to the last previous, still open (= dangling) if-clause"

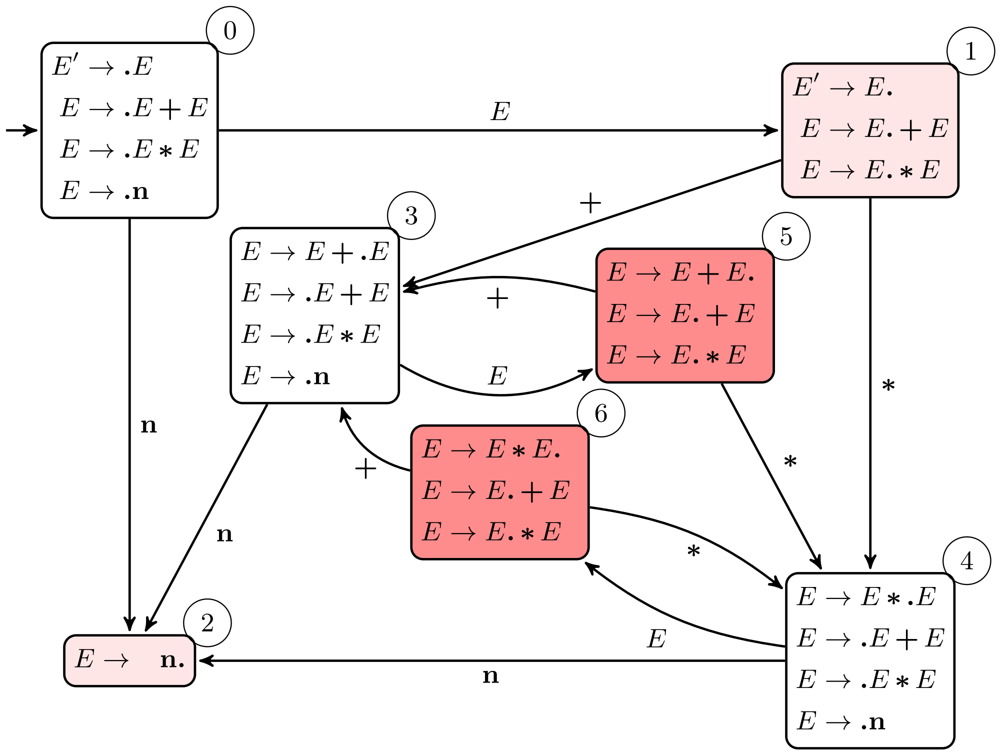
The example serves two purposes: for once shed a light on how the dangling else problem can be "solved" by preferring as shift over a reduce reaction. More generally, it should give (using that standard situation) give a feeling how generally a shift-vs-reduce changes the structure of the parse-tree (and indirectly most probably thereby also the AST). It's an issue of associativity and precedence (at least when dealing with binary operators), and we will see that in the following standard setting of expressions.

Use of ambiguous grammars

- advantage of ambiguous grammars: often simpler
- if ambiguous: grammar guaranteed to have conflicts
- can be (often) resolved by specifying *precedence* and *associativity*
- supported by tools like `yacc` and `CUP` ...

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

DFA for $+$ and $\times$



States with conflicts

- state 5
  - stack contains ...$E + E$
  - for input $\$$: reduce, since shift not allowed form $\$$
  - for input $+$; reduce, as $+$ is *left-associative*
  - for input $*$: shift, as $*$ has *precedence* over $+$
- state 6:
  - stack contains ...$E * E$

  - for input **\$**: reduce, since shift not allowed form **\$**
  - for input $+$; reduce, a $*$ has *precedence* over $+$
  - for input $*$: reduce, as $*$ is *left-associative*
- see also the table on the next slide

Parse table $+$ and $\times$

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		$r:E\to E*E$	$r:E\to E*E$	$r:E\to E*E$	

How about exponentiation (written $\uparrow$ or $**$)?

Defined as *right-associative*. See exercise

An interesting line is the one for state 5, and the difference in reaction when ecncountering a addition vs. a multiplication sign. Basically, the shift for multiplication realizes the fact that multiplication has a higher precedence than addition

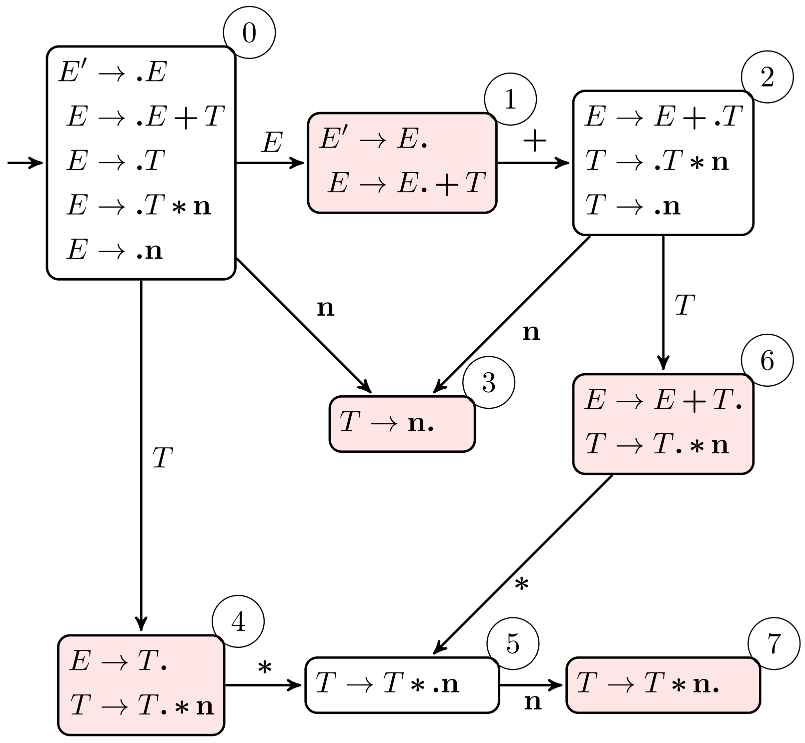
Compare: unambiguous grammar for $+$ and $*$

Unambiguous grammar: precedence and left-assoc built in

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E+T \mid T \\ T & \to & T*\mathbf{n} \mid \mathbf{n} \end{array}$$

	Follow	
E'	$\{\$\}$	(as always for start symbol)
E	$\{\$,+\}$	
T	$\{\$,+,*\}$	

DFA for unambiguous $+$ and $\times$



DFA remarks

- the DFA now is SLR(1)
  - check states with *complete* items

    state 1: $\mathit{Follow}(E') = \{\$\}$

    state 4: $\mathit{Follow}(E) = \{\$, +\}$

    state 6: $\mathit{Follow}(E) = \{\$, +\}$

    state 3/7: $\mathit{Follow}(T) = \{\$, +, *\}$
  - in no case there's a shift/reduce conflict (check the outgoing edges vs. the follow set)
  - there's not reduce/reduce conflict either

LR(1) parsing

- most general from of LR(1) parsing
- aka: *canonical* LR(1) parsing
- usually: considered as unecessarily "complex" (i.e. LALR(1) or similar is good enough)
- "stepping stone" towards LALR(1)

Basic restriction of SLR(1)

Uses *look-ahead*, yes, but only *after* it has built a non-look-ahead DFA (based on **LR(0)**-items)

A help to remember

SLR(1) “improved” LR(0) parsing LALR(1) is “crippled” LR(1) parsing.

Limits of SLR(1) grammars

Assignment grammar fragment[11]

$$\begin{array}{rcl}
\mathit{stmt} & \to & \mathit{call\text{-}stmt} \mid \mathit{assign\text{-}stmt} \\
\mathit{call\text{-}stmt} & \to & \textbf{identifier} \\
\mathit{assign\text{-}stmt} & \to & \mathit{var} \mathrel{\textbf{:=}} \mathit{exp} \\
\mathit{var} & \to & [\,\mathit{exp}\,] \mid \textbf{identifier} \\
\mathit{exp} & \to & \mathit{var} \mid \mathbf{n}
\end{array}$$

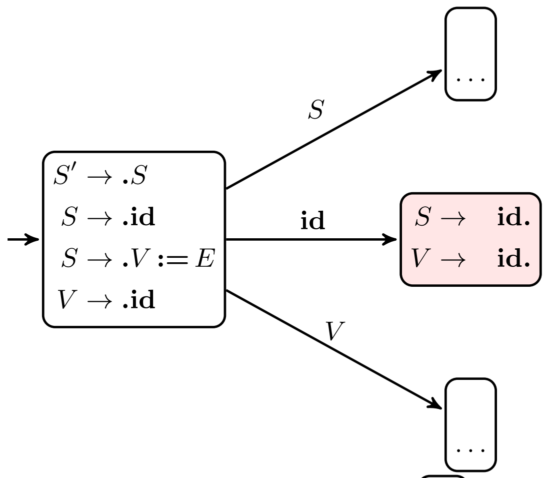
Assignment grammar fragment, simplified

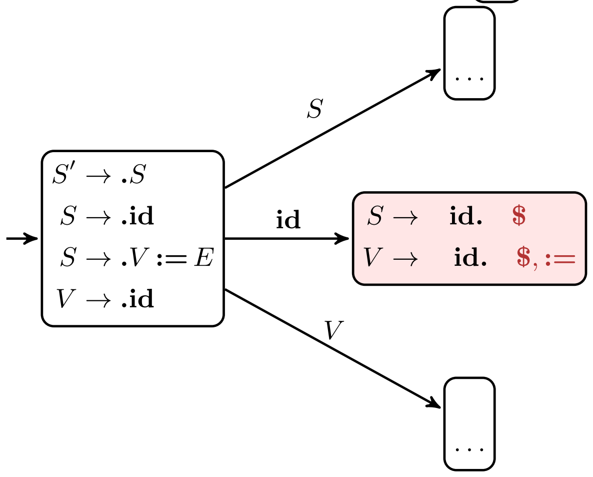
$$\begin{array}{rcl}
S & \to & \textbf{id} \mid V \mathrel{\textbf{:=}} E \\
V & \to & \textbf{id} \\
E & \to & V \mid \mathbf{n}
\end{array}$$

The problematic situation, as we will see on the next slide, concerns identifiers (resp. variables as left-hand side of an assignment or as a call expression).

[11]Inspired by Pascal, analogous problems in C . . .

non-SLR(1): Reduce/reduce conflict

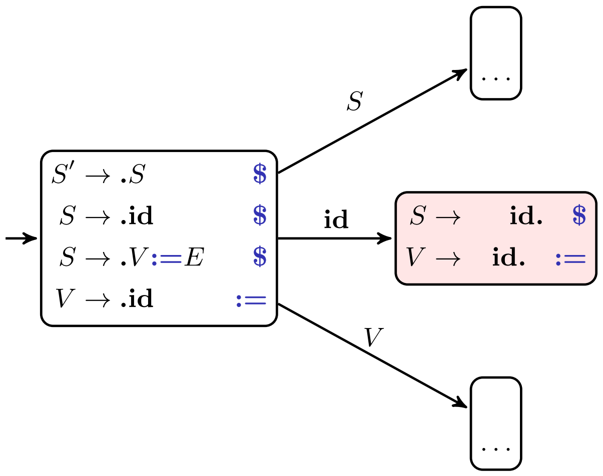




	First	*Follow*
S	$\mathbf{id}$	$\$$
V	$\mathbf{id}$	$\$, :=$
E	$\mathbf{id}, \mathbf{n}$	$\$$

Checking the previously shown conditions for SLR(1)-parsing shows (amongst others) a reduce/reduce conflict situation in the state on the right-hand side. The R/R conflict is on the symbol $\$$: the parser does not know which production to use in the reduce step. 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 on the next slides) builds in one additional look-ahead symbol officially as parts of the items and thus states.

Situation can be saved: more look-ahead



The (sketch of the) automaton here looks pretty similar to the previous one. However, we should think now of the non-terminals as officially part of the items. The interesting piece in this example is the transition from the initial state following the **id**-transition, to the state containing the items $\to$ **id.** and $V \to$ **id.**. That was the state on the previous slide with the reduce/reduce conflict (on the following symbol **$**). Now, without showing the construction in detail (later we give at least the rules for the construction of the NFA, not the DFA with the closure): the interesting situation is, in the first state, the item $S \to .V := E, \$$. With the **.** in front of the V, that's when we have to take the ϵ-closure into account, basically adding also the initial items (here one initial item) for the productions for V into account. Now, by adding that item $V \to .\mathbf{id}$, 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 an* **:=**. That leads (in this situation) to the item of the form $[V \to .\mathbf{id}, :=]$. This information is more specific than the knowledge about the general follow-set of V, which constains **:=** and **$**. Now, by recording that extra piece of information in the closure, the state remembers that the only thing at the current state that is allowed to follow the V is the **:=**. That will defuse the discussed conflict, namely as follows: if we follow the **id**-arrow, we end up in the state on the right-hand side. Such a transition does not touch the additional new look-ahead information (here the **$** resp the **:=** symbol). Thus, in the state at the right-hand side, the reduce-reduce conflict has disappeared!

LALR(1) (and LR(1)): Being more precise with the follow-sets

- LR(0)-items: too "indiscriminate" wrt. the follow sets
- remember the definition of SLR(1) conflicts
- LR(0)/SLR(1)-states:
  - sets of items[12] due to subset construction
  - the items are LR(0)-items
  - follow-sets as an *after-thought*

[12]That won't change in principle (but the items get more complex)

Add precision in the states of the automaton already

Instead of using LR(0)-items and, when the LR(0) DFA is done, try to add a little disambiguation with the help of the follow sets for states containing complete items, better **make more fine-grained items** from the very start:

- **LR(1) items**
- each *item* with "specific follow information": look-ahead

LR(1) items

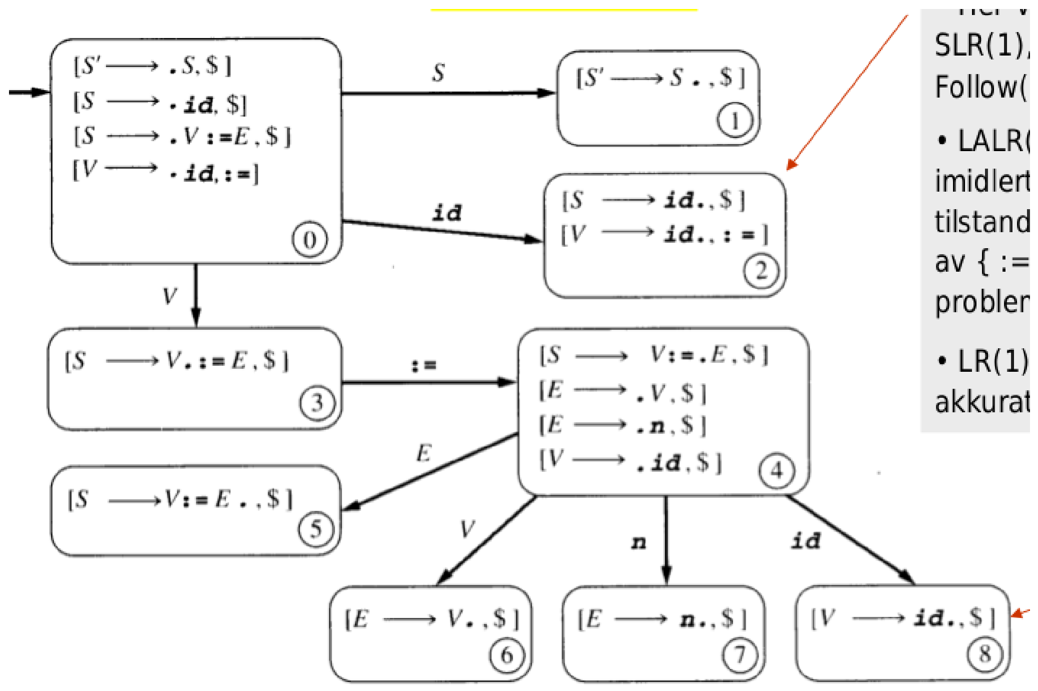
- main idea: simply make the look-ahead part of the item
- obviously: proliferation of states[13]

LR(1) items

$$[A \to \alpha.\beta, \mathbf{a}] \tag{4.9}$$

- **a**: terminal/token, including **$**

LALR(1)-DFA (or LR(1)-DFA)



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

Remarks on the DFA

- Cf. state *2* (seen before)
  - in SLR(1): problematic (reduce/reduce), as $\mathit{Follow}(V) = \{\mathbf{:=}, \mathbf{\$}\}$
  - now: diambiguation, by the added information
- LR(1) would give the same DFA

Full LR(1) parsing

- AKA: **canonical** LR(1) parsing
- the *best* you can do with 1 look-ahead
- unfortunately: big tables
- pre-stage to LALR(1)-parsing

SLR(1)

LR(0)-item-based parsing, with *afterwards* adding some extra "pre-compiled" info (about follow-sets) to increase expressivity

LALR(1)

LR(1)-item-based parsing, but *afterwards* throwing away precision by collapsing states, to save space

LR(1) transitions: arbitrary symbol

- transitions of the **NFA** (not DFA)

X-transition

$$\begin{tikzcd}
\boxed{[A \to \quad \alpha.X\beta, \mathbf{a}]} \arrow[r, "X"] & \boxed{[A \to \quad \alpha X.\beta, \mathbf{a}]}
\end{tikzcd}$$

LR(1) transitions: ϵ

ϵ-transition

for all

$$B \to \beta_1 \mid \beta_2 \ldots \quad \text{and all} \quad \mathbf{b} \in \mathit{First}(\gamma\mathbf{a})$$

$$\begin{tikzcd}
\boxed{[A \to \alpha.B\gamma \quad ,\mathbf{a}]} \arrow[r, "\epsilon"] & \boxed{[B \to .\beta \quad ,\mathbf{b}]}
\end{tikzcd}$$

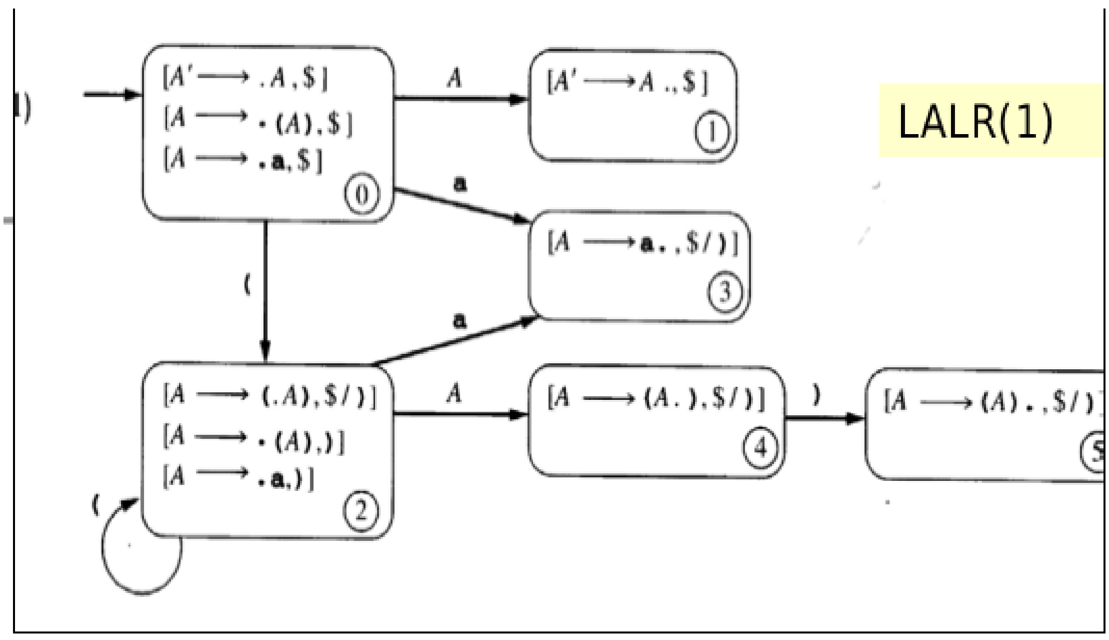
including special case ($\gamma = \epsilon$)

$$\text{for all } B \to \beta_1 \mid \beta_2 \dots$$

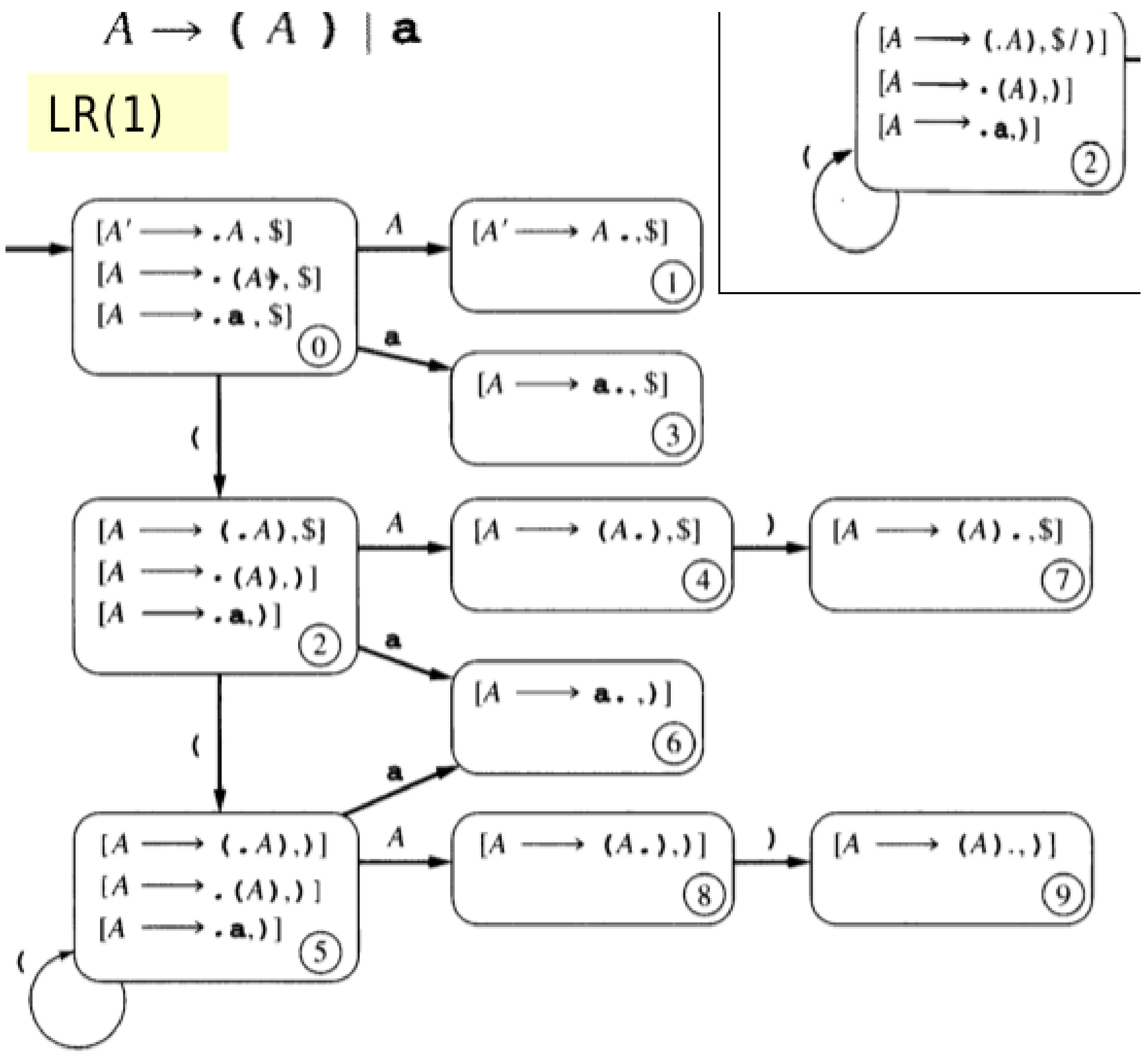
$$\begin{tikzcd}\boxed{[A \to \alpha.B \quad ,\mathbf{a}]} \arrow[r, "\epsilon"] & \boxed{[B \to .\beta \quad ,\mathbf{a}]}\end{tikzcd}$$

LALR(1) vs LR(1)

LALR(1)



LR(1)



Core of LR(1)-states

- actually: not done that way in practice
- main idea: *collapse* states with the same *core*

Core of an LR(1) state

= set of *LR(0)*-items (i.e., ignoring the look-ahead)

- observation: core of the LR(1) item = LR(0) item
- 2 LR(1) states with the same core have same outgoing edges, and those lead to states with the same core

LALR(1)-DFA by as collapse

- collapse all states with the same core
- based on above observations: edges are also consistent
- Result: almost like a LR(0)-DFA but additionally
  - still each individual item has still look ahead attached: the **union** of the "collapsed" items
  - especially for states with *complete* items $[A \to \alpha, \mathbf{a}, \mathbf{b}, \ldots]$ is **smaller** than the follow set of A
  - $\Rightarrow$ less unresolved conflicts compared to SLR(1)

Concluding remarks of LR / bottom up parsing

- all constructions (here) based on BNF (not EBNF)
- *conflicts* (for instance due to ambiguity) can be solved by
  - reformulate the grammar, but generarate the same language[14]
  - use *directives* in parser generator tools like `yacc`, `CUP`, `bison` (precedence, assoc.)
  - or (not yet discussed): solve them later via *semantical analysis*
  - NB: *not all* conflics are solvable, also not in LR(1) (remember ambiguous languages)

LR/bottom-up parsing overview

	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 often good 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 11M of entries), mostly LALR(1) preferred

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

Error handling

Minimal requirement

Upon "stumbling over" an error (= deviation from the grammar): give a *reasonable* & *understandable* error message, indicating also error *location*. Potentially stop parsing

[14] 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.

- for parse error *recovery*
  - one cannot really recover from the fact that the program has an error (an syntax error is a syntax error), but
  - after giving decent error message:
    - move on, potentially jump over some subsequent code,
    - until parser can *pick up* normal parsing again
    - so: meaningfull checking code even following a first error
  - avoid: reporting an avalanche of subsequent *spurious* errors (those just "caused" by the first error)
  - "pick up" again after semantic errors: easier than for syntactic errors

Error messages

- important:
  - avoid error messages that only occur because of an already reported error!
  - report error as early as possible, if possible at the *first point* where the program cannot be extended to a correct program.
  - make sure that, after an error, one doesn't end up in an *infinite loop* without reading any input symbols.
- What's a good error message?
  - assume: that the method `factor()` chooses the alternative **(** *exp* **)** but that it , when control returns from method `exp()`, does not find a **)**
  - one could report : `right parenthesis missing`
  - But this may often be confusing, e.g. if what the program text is: `( a + b c )`
  - here the `exp()` method will terminate after `( a + b`, as `c` cannot extend the expression). You should therefore rather give the message `error in expression or right parenthesis missing`.

Error recovery in bottom-up parsing

- *panic recovery* in LR-parsing
  - simple form
  - the only one we shortly look at
- upon error: recovery ⇒
  - pops parts of the stack
  - ignore parts of the input
- until "on track again"
- but: how to do that
- additional problem: *non-determinism*
  - table: constructed *conflict-free* **under normal operation**
  - upon error (and clearing parts of the stack + input): no guarantee it's clear how to continue

⇒ **heuristic** needed (like panic mode recovery)

Panic mode idea

- try a **fresh start**,
- promising "fresh start" is: a possible **goto** action
- thus: back off and take the *next* such goto-opportunity

Possible error situation

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

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

Panic mode recovery

Algo

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

Example again

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

- first pop, until in state 3
- then jump over input
  - until next input **g**
  - since **f** and **)** cannot be treated
- choose to goto v (shift in that state)

Panic mode may loop forever

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

Panicking and looping

	parse stack	input	action
1	$\$_0$	$(\ \mathbf{n}\ \mathbf{n}\ )\ \$$	
2	$\$_0(_6$	$\mathbf{n}\ \mathbf{n}\ )\ \$$	
3	$\$_0(_6\mathbf{n}_5$	$\mathbf{n}\ )\ \$$	
4	$\$_0(_6 \mathit{factor}_4$	$\mathbf{n}\ )\ \$$	
6	$\$_0(_6 \mathit{term}_3$	$\mathbf{n}\ )\ \$$	
7	$\$_0(_6 \mathit{exp}_{10}$	$\mathbf{n}\ )\ \$$	panic!
8	$\$_0(_6 \mathit{factor}_4$	$\mathbf{n}\ )\ \$$	been there before: stage 4!

- error raised in stage 7, no action possible
- panic:
  1. pop-off exp_{10}
  2. state 6: 3 goto's

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

  3. no shift, so we need to decide between the two reduces
  4. *factor*: less general, we take that one

How to deal with looping panic?

- make sure to detec loop (i.e. previous "configurations")
- if loop detected: doen't repeat but do something special, for instance
  - pop-off more from the stack, and try again
  - pop-off and *insist* that a shift is part of the options

Left out (from the book and the pensum)

- more info on error recovery
- expecially: more on `yacc` error recovery
- it's not pensum, and for the oblig: need to deal with CUP-specifics (not classic `yacc` specifics even if similar) anyhow, and error recovery is not part of the oblig (halfway decent error *handling* is).

Bibliography

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

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

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

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

Index