Course Script

INF 5110: Compiler construction

INF5110, spring 2018

Martin Steffen

Contents

Chapter 4
Parsing

What is it about?

Learning Targets of this Chapter

1. (context-free) grammars + BNF
2. ambiguity and other properties
3. terminology: tokens, lexemes,
4. different trees connected to grammars/parsing
5. derivations, sentential forms

The chapter corresponds to [5, Section 3.1–3.2] (or [6, Chapter 3]).

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*

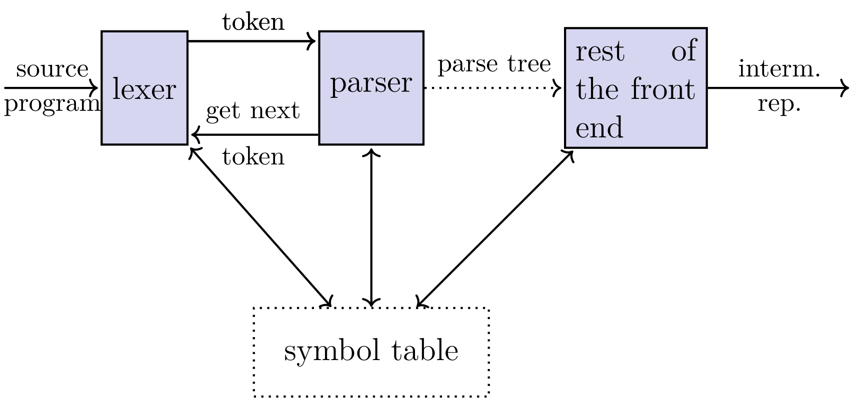
- the full "power" (i.e., expressiveness) of CFGs not used
- thus:
  - consider *restrictions* of CFGs, i.e., a specific subclass, and/or
  - *represented* in specific ways (no left-recursion, left-factored . . .)

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 location 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

- while parse tree mostly conceptual: parsing build up the concrete data structure of AST bottom-up vs. top-down.

Parsing restricted classes of CFGs

- parser: better be "efficient"
- full complexity of CFLs: not really needed in practice[1]
- 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:

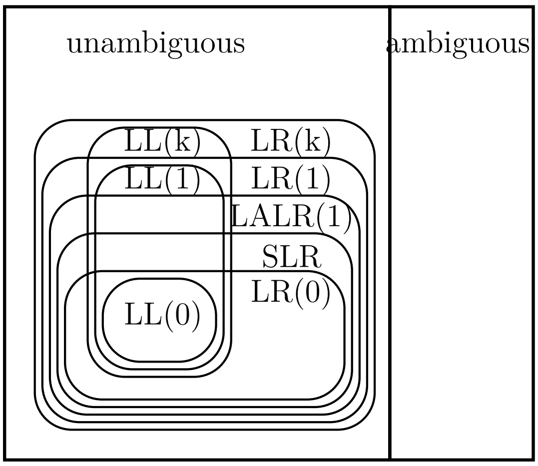
[1] Perhaps: if a parser has trouble to figure out if a program has a syntax error or not (perhaps using back-tracking), probably humans will have similar problems. So better keep it simple. And time in a compiler may be better spent elsewhere (optimization, semantical analysis).

- based on accepted notation for grammars: (BNF or some form of EBNF etc.)

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]

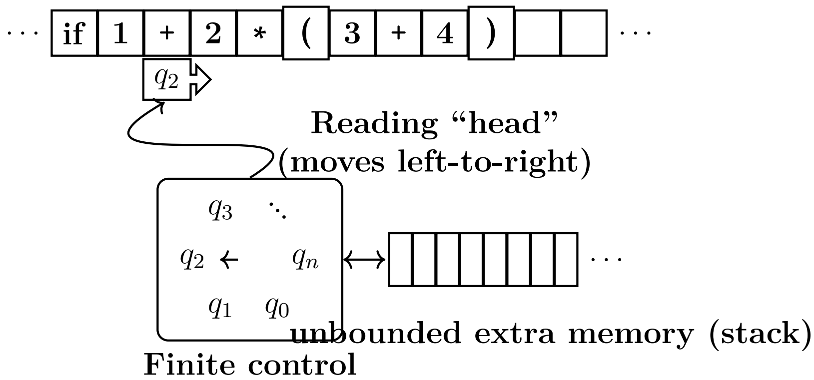
4.2 Top-down parsing

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 reprodiced here (resp. only a few slides later as some big array of steps).

factors and terms

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

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:

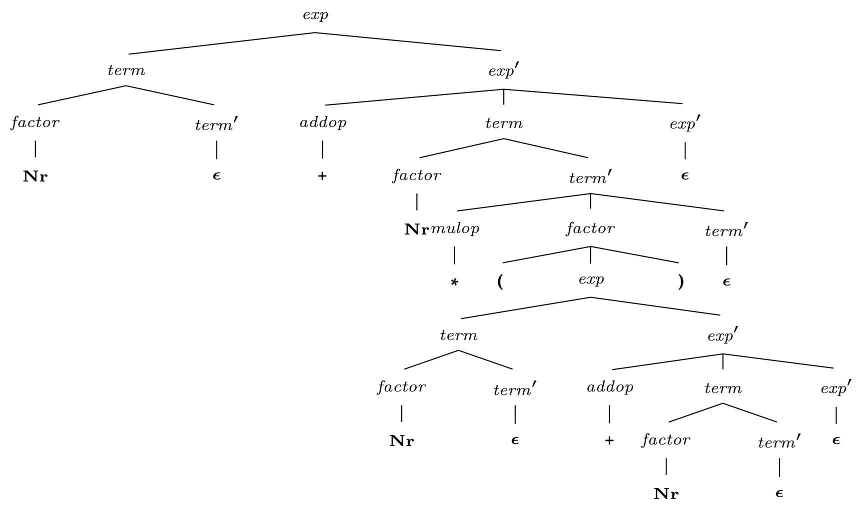
- underline: the *place* (occurrence of *non-terminal* where production is used)
- ~~*crossed out*~~:
  - *terminal* = *token* is considered treated
  - parser "moves on"
  - later implemented as `match` or `eat` procedure

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

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

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

Oracular derivation

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

$\underline{\mathit{exp}}$	$\Rightarrow_1$	$\downarrow 1 + 2 * 3$
$\underline{\mathit{exp}} + \mathit{term}$	$\Rightarrow_3$	$\downarrow 1 + 2 * 3$
$\underline{\mathit{term}} + \mathit{term}$	$\Rightarrow_5$	$\downarrow 1 + 2 * 3$
$\underline{\mathit{factor}} + \mathit{term}$	$\Rightarrow_7$	$\downarrow 1 + 2 * 3$
$\mathbf{number} + \mathit{term}$		$\downarrow 1 + 2 * 3$
$\mathbf{number} + \mathit{term}$		$1 \downarrow + 2 * 3$
$\mathbf{number} + \underline{\mathit{term}}$	$\Rightarrow_4$	$1 + \downarrow 2 * 3$
$\mathbf{number} + \underline{\mathit{term}} * \mathit{factor}$	$\Rightarrow_5$	$1 + \downarrow 2 * 3$
$\mathbf{number} + \underline{\mathit{factor}} * \mathit{factor}$	$\Rightarrow_7$	$1 + \downarrow 2 * 3$
$\mathbf{number} + \mathbf{number} * \mathit{factor}$		$1 + \downarrow 2 * 3$
$\mathbf{number} + \mathbf{number} * \mathit{factor}$		$1 + 2 \downarrow * 3$
$\mathbf{number} + \mathbf{number} * \underline{\mathit{factor}}$	$\Rightarrow_7$	$1 + 2 * \downarrow 3$
$\mathbf{number} + \mathbf{number} * \mathbf{number}$		$1 + 2 * \downarrow 3$
$\mathbf{number} + \mathbf{number} * \mathbf{number}$		$1 + 2 * 3 \downarrow$

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-head there is, namely until the end of the word. and it can *still* not 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

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

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), so that is not the problem.

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.

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, obvously, 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 as it is always done).

Two principle sources of non-determinism here

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]
2. **which production** to apply (for the chosen non-terminal).

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[3]
- 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 $A \to \beta$

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

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

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

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

[2]Note that α_1 and α_2 may contain non-terminals, including further occurrences of A.
[3]A CFG is ambiguous, if there exists a word (of terminals) with 2 different parse trees.

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

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

- 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.[4]

[4] 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

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 for derivations, but on *grammars* that can be used to check, 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 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*.

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 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* later be used when discussing bottom-up parsing.

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 $First_G(A)$ is defined as

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

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 set

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

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

- in many languages:

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

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

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

Remarks

- note: special treatment of the empty word ϵ
- 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*)

$\Rightarrow$ **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 $First(X)$, is defined as follows:

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

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

   a) $First(X)$ contains $First(X_1) \setminus \{\boldsymbol{\epsilon}\}$
   b) If, for some $i < n$, *all* $First(X_1), \dots, First(X_i)$ contain $\boldsymbol{\epsilon}$, then $First(X)$ contains $First(X_{i+1}) \setminus \{\boldsymbol{\epsilon}\}$.
   c) If all $First(X_1), \dots, First(X_n)$ contain $\boldsymbol{\epsilon}$, then $First(X)$ 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 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 get's 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 $First(X)$ is defined as follows:

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

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

$$First(X) \ \supseteq \ First(X_1).$$

Compared to the previous condition, I did the following 2 minor adaptations (apart from cleaning up the $\boldsymbol{\epsilon}$'s): In case (2), I replaced the English word "contains" with the superset relation symbol $\supseteq$. In case (1), I replaced the equality symbol = with the superset symbol ⊇, basically for consistency with the other case.

Now, with Definition 4.3.4 as a simplified version of the original definition being made slightly more explicit and internally consistent: 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 that definition is just "sloppy". Why does it use the word "contain" resp. the ⊇-relation, instead of requiring equality, i.e., =? While plausible at first sight, unfortunately, whether we use ⊇ or set equality = in Definition 4.3.4 does not change anything (and remember that the original Definition 4.3.3 "mixed up" the styles by requiring equality in the case of non-terminals and requiring "contains", i.e., ⊇ for non-terminals).

Anyhow, the core of the matter is not = vs. ⊇. 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 ⊇ as the recursive definition of a function leads immediately to an "infinite regress", the recurive 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

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

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

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

Concerning the definition of First

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

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

The second question has no “real” answer, it's a choice which is being made which could be made differently. What the definition from equation (4.3.1) in fact says is: “give the set of terminal symbols in the derivable word **and** indicate whether or not the start symbol is *nullable*. The information might as well be interpreted as a *pair* consisting of a set of terminals *and* a boolean (indicating nullability). The fact that the definition of *First* as presented here uses ϵ to indicate that additional information is a particular choice of representation (probably due to historical reasons: “they always did it like that ...”). For instance, the influential “Dragon book” [1, Section 4.4.2] uses the ϵ-based definition. The texbooks [2] (and its variants) don't use ϵ as indication for nullability.

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

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

As a further point concerning the first sets: 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 allX \in A ∪ {ϵ} 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 → X_1 ... X_n do
    k := 1;
    continue := true
    while continue = true and k ≤ n do
      First[A] := First[A] ∪ First[X_k] ∖ {ϵ}
      if ϵ ∉ First[X_k] then continue := false
      k := k + 1
    end;
    if  continue = true
    then First[A] := First[A] ∪ {ϵ}
  end;
end
```

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

for grammar without ϵ-*productions*.[5]

```
for all non-terminals A do
  First[A] := {}        // counts as change
end
while there are changes to any First[A] do
```

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

```
  for  each production A → X_1 ... X_n  do
       First [A]  :=  First [A]  ∪  First [X_1]
  end;
end
```

This simplification is added for illustration, only. 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)

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

Example expression grammar (expanded)

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

"Run" of the algo

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

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 $A \rightarrow X_1 \dots X_n$ 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	$\{\mathbf{(}, \mathbf{n}\}$
$addop$	$\{+, -\}$
$term$	$\{\mathbf{(}, \mathbf{n}\}$
$mulop$	$\{*\}$
$factor$	$\{\mathbf{(}, \mathbf{n}\}$

Work-list formulation

```
for  all non-terminals A  do
  First[A]  := {}
  WL        := P    // all productions
end
while WL ≠ ∅ do
  remove one (A → X1 ... Xn) from WL
  if     First[A] ≠ First[A] ∪ First[X1]
  then   First[A] := First[A] ∪ First[X1]
     add all productions (A → X'1 ... X'm) to WL
  else  skip
end
```

- worklist here: "collection" of productions
- alternatively, with slight reformulation: "collection" of non-terminals instead also possible

Follow sets

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

The *follow-set* of A, written $Follow_G(A)$, is

$$Follow_G(A) = \{a \mid S \Rightarrow_G^* \alpha_1 A a \alpha_2, \quad a \in \Sigma_T\} . \tag{4.5}$$

- More generally: **\$** as special end-marker

$$S\,\mathbf{\$} \Rightarrow_G^* \alpha_1 A a \alpha_2, \quad a \in \Sigma_T + \{\,\mathbf{\$}\,\} .$$

- 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 on 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 has 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 enty 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.

Special symbol $

It seems that [3] does not use the special symbol in his treatment of the follow set, but the dragon book uses it. It is used to represent the symbol (not otherwise used) "right of the start symbol", resp., the symbol right of a non-terminal which is at the right end of a derived word.

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 $\mathit{First}(\beta) \setminus \{\boldsymbol{\epsilon}\}$.

3. If there is a production $B \to \alpha A \beta$ such that $\epsilon \in First(\beta)$, then $Follow(A)$ contains $Follow(B)$.

- **$**: "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 ≠ S do
  Follow[A] := {}
end
while there are changes to any Follow-set do
  for each production A → X_1 ... X_n do
    for each X_i which is a non-terminal do
      Follow[X_i] := Follow[X_i] ∪ (First(X_{i+1} ... X_n) ∖ {ε})
      if ε ∈ First(X_{i+1} X_{i+2} ... X_n)
      then Follow[X_i] := Follow[X_i] ∪ Follow[A]
    end
  end
end
```

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

Expression grammar once more

"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 $)$		

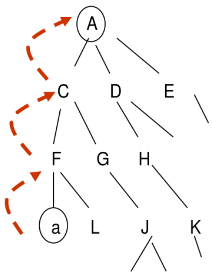
normalsize

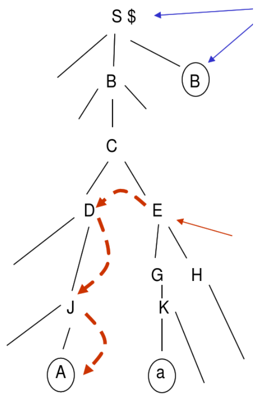
Recursion vs. iteration

"Run" of the algo

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

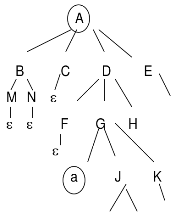
In case of the tree on the left (for the first sets): A is the root and must therefore be the start symbol. Since the root A has three children C, D, and E, there must be a production $A \rightarrow C\ D\ E$. etc.

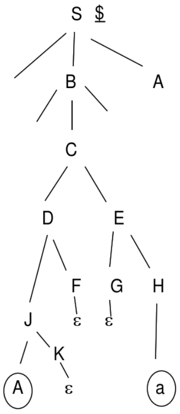
The first-set definition would "immediately" detect that F has **a** in its first-set, i.e., all words derivable starting from F start with an a (and actually with no other terminal, as F is mentioned only once in that sketch of a tree). At any rate, only *after* determining that **a** is in the first-set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

It should be stressed ...

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 ϵ. There will also be exercises about that.

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. 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 amout 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 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 the *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

$$exp \rightarrow exp + term$$

- classical example for common left factor: rules for conditionals

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

Transforming the expression grammar

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

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

After removing left recursion

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

- 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

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

space

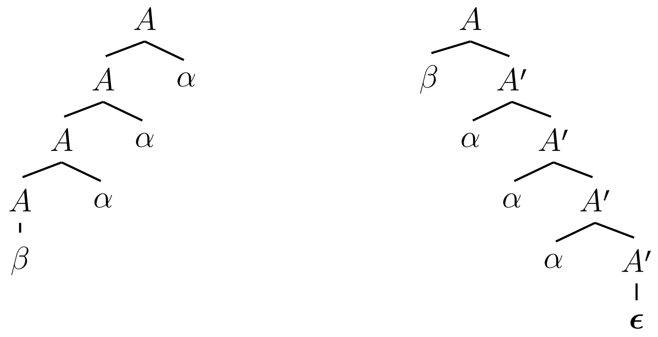
After

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

Schematic representation

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

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



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

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

space

After

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

EBNF

Note: can be written in *EBNF* as:

$$A \rightarrow (\beta_1 \mid \ldots \mid \beta_m)(\alpha_1 \mid \ldots \mid \alpha_n)^*$$

Removal of: general left recursion

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

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

"current" = rule in the current stage of algo

Example (for the general case)

let $A = A_1$, $B = A_2$

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

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

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

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

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

1. before

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

2. after

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

Example: sequence of statements

sequences of statements

1. Before

$$\begin{array}{lcl} \textit{stmt-seq} & \rightarrow & \textit{stmt}\,\mathbf{;}\,\textit{stmt-seq} \\ & \mid & \textit{stmt} \end{array}$$

2. After

$$\begin{array}{lcl} \textit{stmt-seq} & \rightarrow & \textit{stmt}\ \textit{stmt-seq}' \\ \textit{stmt-seq}' & \rightarrow & \mathbf{;}\,\textit{stmt-seq} \mid \boldsymbol{\epsilon} \end{array}$$

Example: conditionals

1. Before

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

2. After

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

Example: conditionals (without else)

1. Before

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

2. After

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

Not all factorization doable in "one step"

1. Starting point

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

2. After 1 step

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

3. After 2 steps

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

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

4.4 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

1. looking **1 symbol ahead**.

Explanation

- 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 back-tracking. 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 expr grammar again

factors and terms

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

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[6]

Example: 2 productions for non-terminal *factor*

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

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

Remark

that situation is *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 right now): when doing the above successfully, build the *AST* for the accepted nonterminal.

Recursive descent

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);
     }
}
```

Recursive descent

```
qtype 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[7]

First

⇒ definition of the *First* **set**

Lemma 4.4.1 (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

$$\begin{array}{lcl} \textit{if-stmt} & \to & \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt} \\ & \mid & \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \end{array}$$

- requires a look-ahead of (at least) 2
- ⇒ try to rearrange the grammar
  1. *Extended* BNF ([6] suggests that)

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

  1. *left-factoring*:

[7] 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.

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

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

$$
\begin{array}{rcl}
\textit{exp} & \rightarrow & \textit{exp}\ \textit{addop}\ \textit{term} \mid \textit{term} \\
\textit{addop} & \rightarrow & \texttt{+} \mid \texttt{-} \\
\textit{term} & \rightarrow & \textit{term}\ \textit{mulop}\ \textit{factor} \mid \textit{factor} \\
\textit{mulop} & \rightarrow & \texttt{*} \\
\textit{factor} & \rightarrow & \textbf{(}\ \textit{exp}\ \textbf{)} \mid \textbf{number}
\end{array}
\tag{4.7}
$$

Left recursion explanation

- consider treatment of *exp*: $\mathit{First}(\mathit{exp})$?
  - whatever is in $\mathit{First}(\mathit{term})$, is in $\mathit{First}(\mathit{exp})$[8]
  - even if only *one* (left-recursive) production $\Rightarrow$ *infinite* recursion.

Left-recursion

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

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

Removing left recursion may help

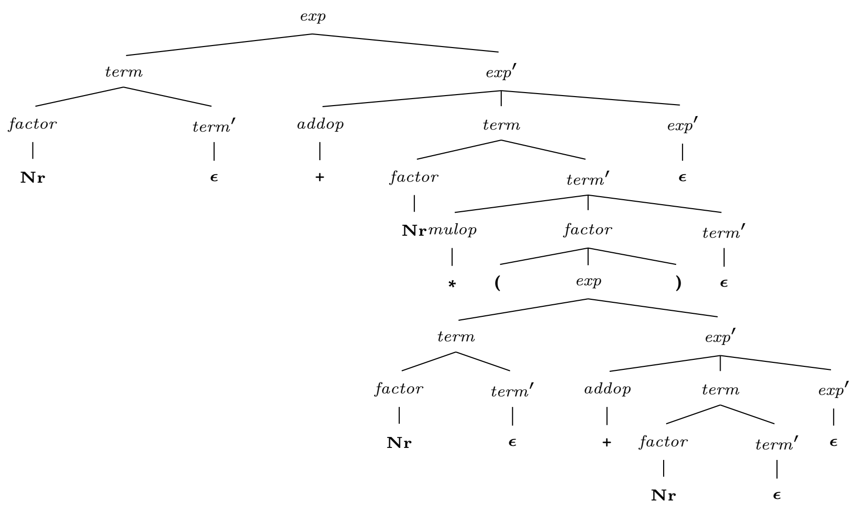
Pseudo code

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

```
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?

The two expression grammars again

factors and terms

1. Precedence & assoc.

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

2. explanation
   - clean and straightforward rules
   - left-recursive

no left recursion

1. no left-rec.

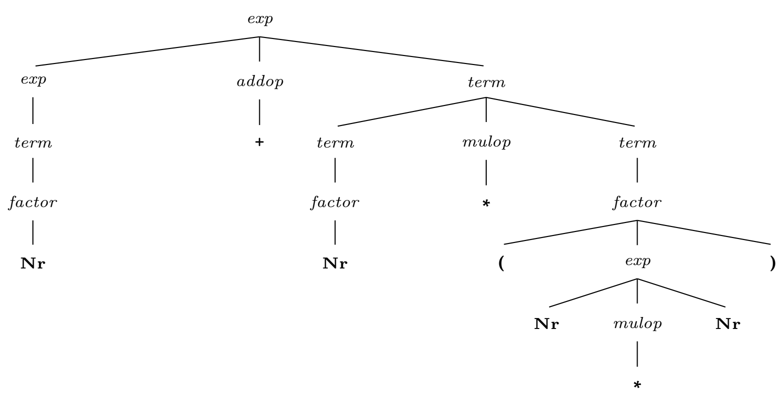
$$\begin{array}{rcl} exp & \rightarrow & term\ exp' \\ exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\ addop & \rightarrow & + \mid - \\ term & \rightarrow & factor\ term' \\ term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\ mulop & \rightarrow & * \\ factor & \rightarrow & (\, exp\,)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

2. Explanation
   - no left-recursion

- assoc. / precedence ok
- rec. descent parsing ok
- but: just "unnatural"
- non-straightforward parse-trees

Left-recursive grammar with nicer parse trees

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



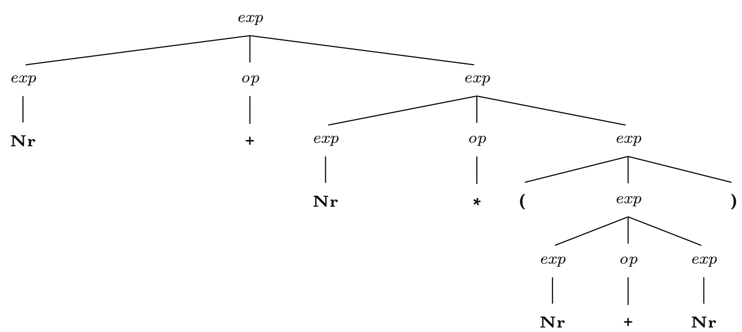
The simple "original" expression grammar (even nicer)

Flat expression grammar

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

Nice tree

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



Associtivity problematic

Precedence & assoc.

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

Example plus and minus

1. Formula

$$3 + 4 + 5$$

parsed "as"

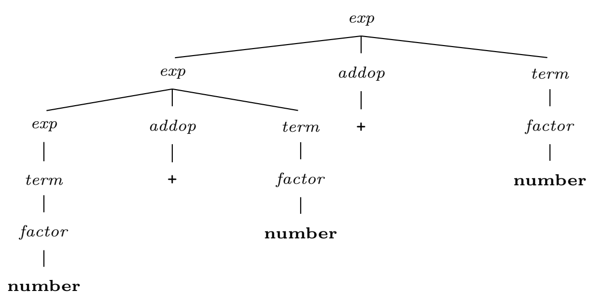
$$(3 + 4) + 5$$

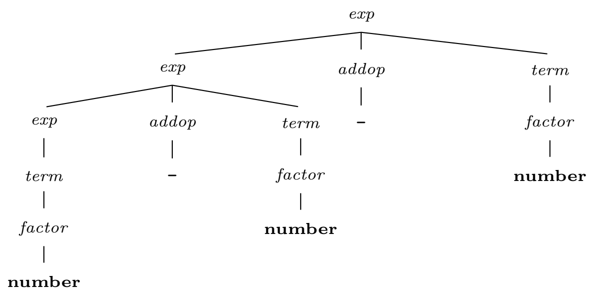
$$3 - 4 - 5$$

parsed "as"

$$(3 - 4) - 5$$

2. Tree





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

No left-rec.

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

Example minus

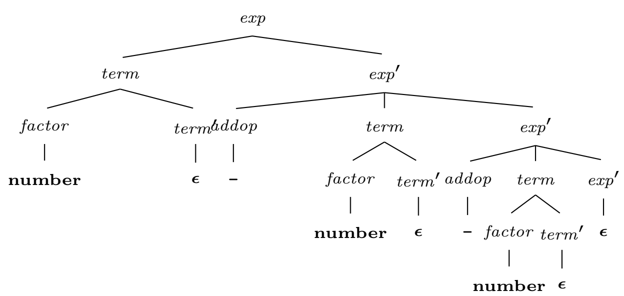
1. Formula

$$3 - 4 - 5$$

parsed "as"

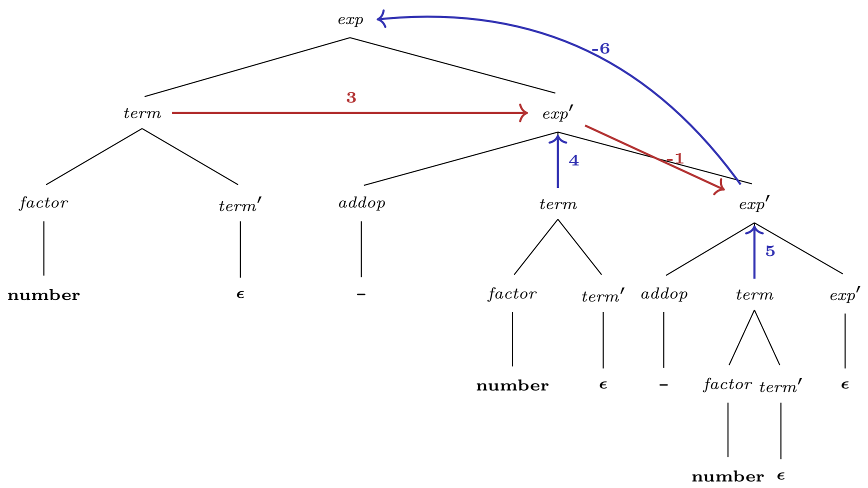
$$3 - (4 - 5)$$

2. 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`

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

- trade offs:
  1. starting from: design of the language, how much of the syntax is left "implicit"[9]

[9]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 ...

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

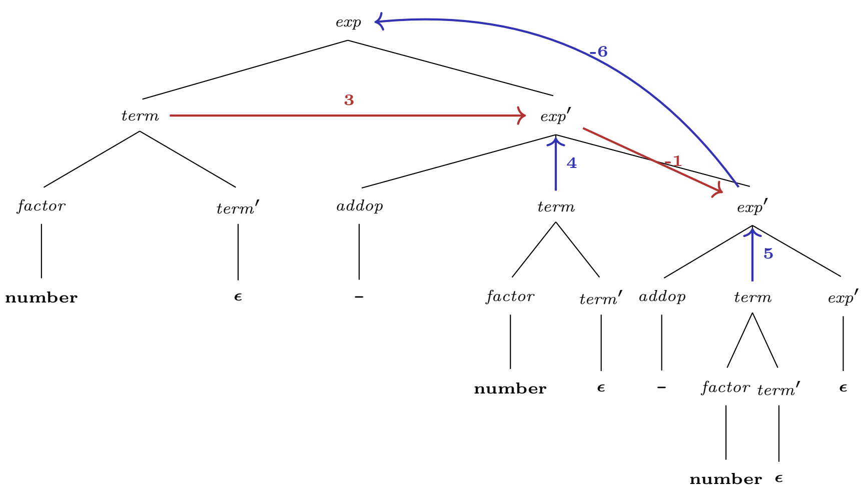
- once steps 1.–3. are fixed: *parse-trees* fixed!
- parse-trees = *essence* of grammatical derivation process
- often: parse trees only "conceptually" present in a parser
- AST:
  - *abstractions* of the parse trees
  - *essence* of the parse tree
  - actual tree data structure, as output of the parser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

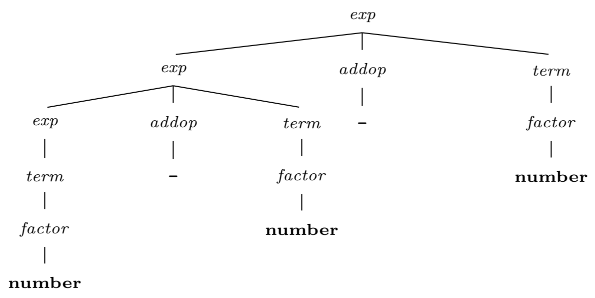
Parser "**builds**" the AST data structure while "**doing**" the parse tree

AST: How "far away" from the CST?

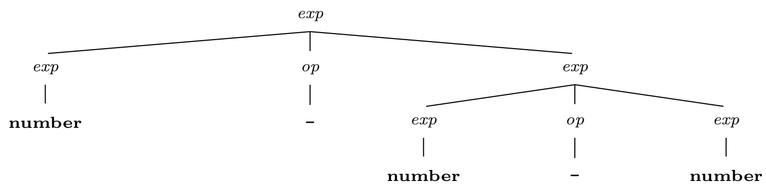
- AST: only thing relevant for later phases ⇒ better be *clean* ...
- AST "=" CST?
  - building AST becomes straightforward
  - possible choice, **if** the grammar is not designed "weirdly",



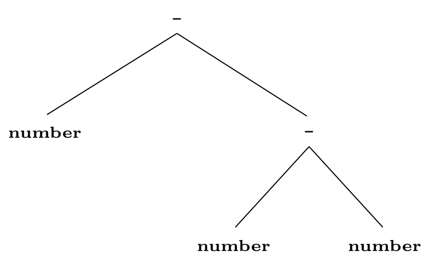
parse-trees like that better be cleaned up as AST

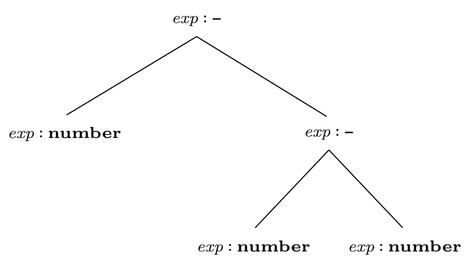


slightly more reasonable looking as AST (but underlying grammar not directly useful for recursive descent)



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

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

Explanation

- 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) **sub-class** 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

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

```java
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 -> "*"
}
```

$3 - (4 - 5)$

```
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

$\Rightarrow$ that class is *not* needed

- some might prefer an implementation of

$$op \to \texttt{+} \mid \texttt{-} \mid \texttt{*}$$

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 (at least with pros who (of course) can read BNF) what the syntax is. 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)

Louden

- a word on [6]: His C-based representation of the AST is a bit on the "bit-squeezing" side of things . . .

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

$$\begin{aligned} exp &\rightarrow term\{\, addop\, term \,\} \\ term &\rightarrow factor\{\, mulop\, factor \,\} \end{aligned}$$

Explanation

but remember:

- EBNF just a notation, just because we do not see (left or right) recursion in $\{\ldots\}$, does not mean there is no recursion.
- not all parser generators support EBNF
- however: often easy to translate into loops- [10]
- 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
```

How to produce "something" during RD parsing?

Recursive descent

So far: RD = top-down (parse-)tree traversal via recursive procedure.[11] Possible outcome: termination or failure.

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

[11] 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.

Rest

- 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
```

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

Building an AST: factor

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

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

Building an AST: conditionals

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

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

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

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

LL(1) parsing principle

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

Further remarks

- 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 2 dimensional array)[12]
    $$M : \Sigma_N \times \Sigma_T \to ((\Sigma_N \times \Sigma^*) + \texttt{error})$$
  - $M[A, a] = w$
- we assume: pure BNF

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

Example: if-statements

- grammars is left-factored and not left recursive

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

[12] Often, 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 $\Sigma_N \times \Sigma_T \to (\Sigma^* + \texttt{error})$. We follow the convention of this book.

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

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!

LL(1) table based algo

```
while  the top of the parsing stack ≠ $
  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, a]  contains
                production  A → X1X2...Xn
          then  (*  generate  *)
                pop the parsing stack
                for  i := n  to  1  do
```

```
                  push X onto the stack;
          else error
    if    the top of the stack = $
    then  accept
end
```

LL(1): illustration of run of the algo

Table 4.3
LL(1) parsing actions for if-statements using the most closely nested disambiguating rule

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

*

Remark

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

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

left-recursive ⇒ not LL(k)

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

	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					

Error handling

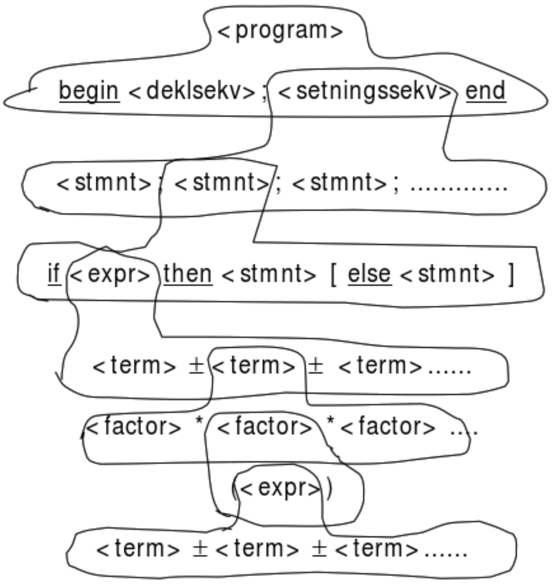
- at the least: do an understandable error message
- give indication of line / character or region responsible for the error in the source file
- potentially *stop* the parsing
- some compilers do *error recovery*
  - give an understandable error message (as minimum)
  - continue reading, until it's 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 : `left 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 left paranthesis missing`.

Handling of syntax errors using recursive descent

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



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.

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

4.5 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 intro to 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

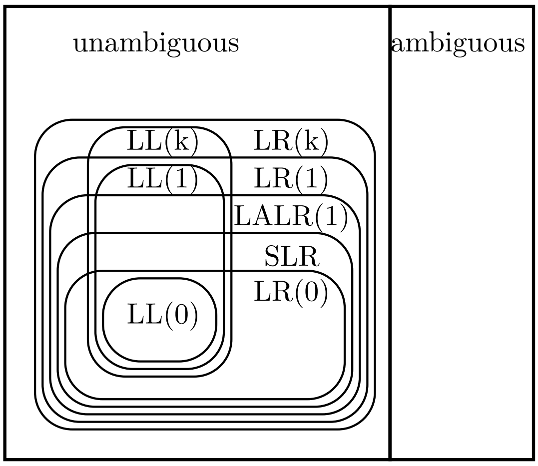
LALR(1)
- slightly more expressive than SLR(1)
- same number of states as LR(0)
- we look at ideas behind that method as well

LR(1) covers all grammars, which can in principle be parsed by looking at the next token

Remarks

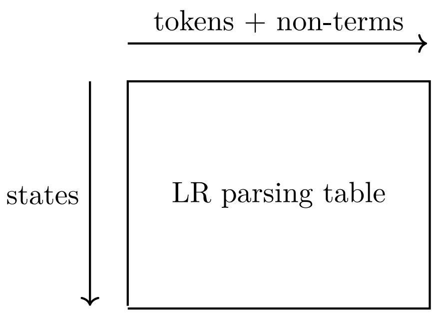
There seems to be a contradiction in the explanation of LR(0): if LR(0) is so weak that it works only for unreasonably simple language, how can one speak about that standard languages have 300 states? The answer is, the other more expressive parsers (SLR(1) and LALR(1)) use the *same* construction of states, so that's why one can estimate the number of states, even if standard languages don't have a 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 parsing 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 its descendants (like bison, CUP, etc)
- another name: *shift-reduce* parser



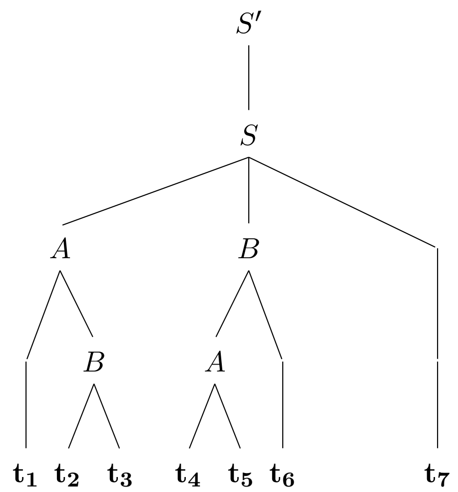
Example grammar

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

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

- general agreement for bottom-up parsing:
  - start symbol *never* on the right-hand side or a production
  - **routinely add another "extra" start-symbol** (here S')[13]

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



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

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

Potentially puzzling question at first sight:

How does the parser *right*-most derivation, when parsing *left*-to-right?

[13]That 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.

Discussion

- 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

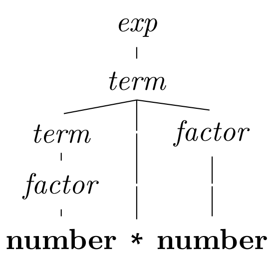
$$S \Rightarrow_r^* \alpha$$

Slighly longer answer

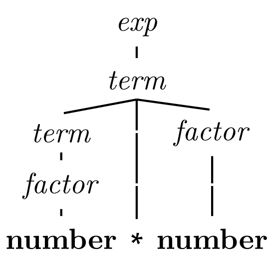
LR parser parses from left-to-right and builds the parse tree bottom-up. When doing the parse, the parser (implicitly) builds a *right-most* derivation **in reverse** (because of bottom-up).

Example expression grammar (from before)

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



Bottom-up parse: Growing the parse tree



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

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 entity

- 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*

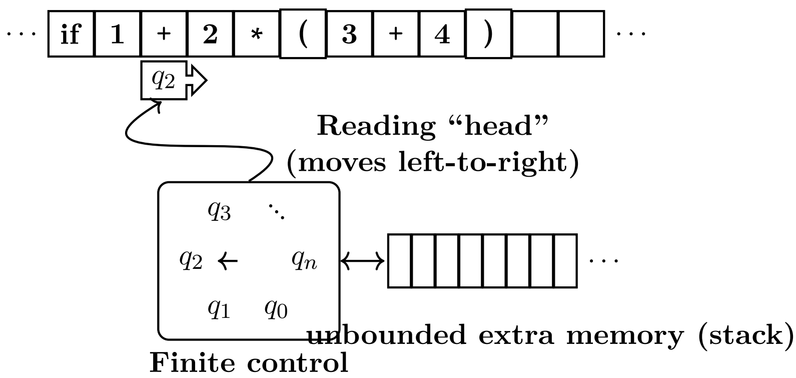
Handle

Definition 4.5.1 (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* (= ϵ)

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)

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

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").

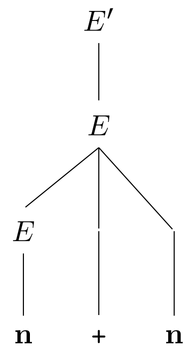
Remarks

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

Example: LR parsing for addition (given the tree)

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

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"

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

Example with ϵ-transitions: parentheses

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

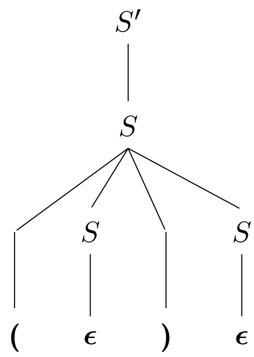
side remark: unlike previous grammar, here:

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

$\Rightarrow$ difference between left-most and right-most derivations (and mixed ones)

Parentheses: tree, 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

$$S \Rightarrow_r^* \alpha$$

Explanation

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

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

Derivation and split

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

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

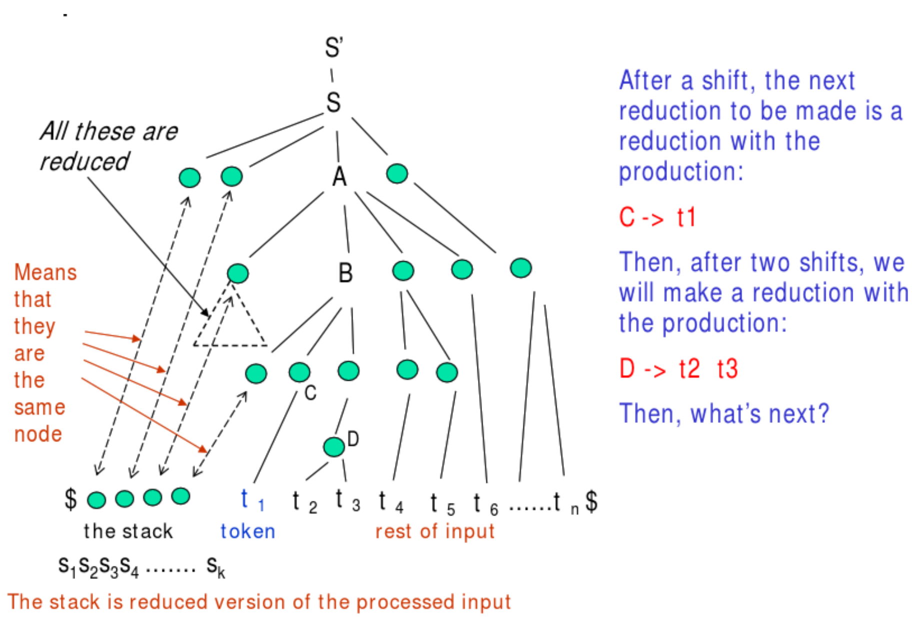
Rest

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

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



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**!

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 $k > 1$

LR(0) item

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

Rest

- **.** is, of course, a "meta-symbol" (not part of the production)
- For instance: production $A \to \alpha$, where $\alpha = \beta\gamma$, then

LR(0) item

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

complete and initial items

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

Example: items of LR-grammar

Grammar for parentheses: 3 productions

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

8 items

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

Remarks

- note: $S \to \epsilon$ gives $S \to .$ as item (not $S \to \epsilon.$ and $S \to .\epsilon$)
- side remark: see later, it will turn out: grammar *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

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

Remarks: no LR(0)

- also here: it will turn out: *not 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)

State transitions of the NFA

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

Terminal or non-terminal

$$\boxed{A \to \alpha.X\eta} \xrightarrow{X} \boxed{A \to \alpha X.\eta}$$

Epsilon (X: non-terminal here)

$$\boxed{A \to \alpha.X\eta} \xrightarrow{\epsilon} \boxed{X \to .\beta}$$

Explanation

- In case X = *terminal* (i.e. token) =
  - the left step corresponds to a **shift** step[14]
- for non-terminals (see next slide):

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

  - 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

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
- however: the 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 kind of transitions
  1. the $\boldsymbol{\epsilon}$-transition correspond to the "pop" half
  2. that X transition (for non-terminals) corresponds to that "eat-and-push" part
- assume production $X \to \beta$ and *initial* item $X \to .\beta$

Terminal or non-terminal

$$\boxed{A \to \alpha.X\eta} \xrightarrow{X} \boxed{A \to \alpha X.\eta}$$

Epsilon (X: non-terminal here)

Given production $X \to \beta$:

$$\boxed{A \to \alpha.X\eta} \xrightarrow{\epsilon} \boxed{X \to .\beta}$$

Initial and final states

initial states:

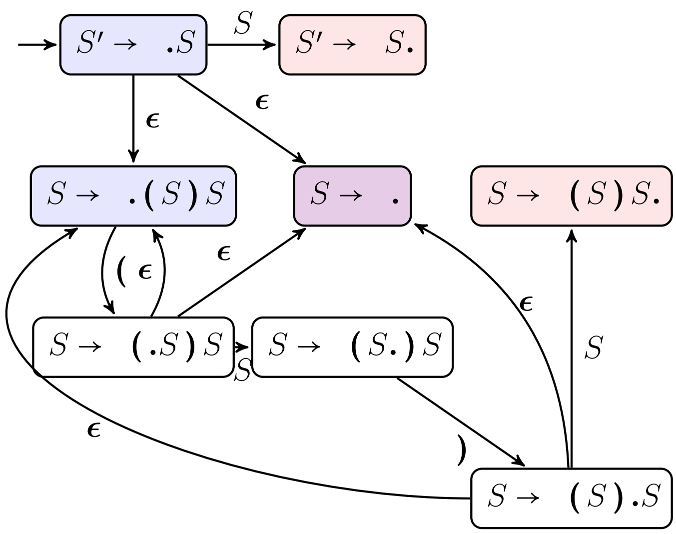
- we make our lives *easier*
- we assume (as said): one *extra* start symbol say S' (augmented grammar)

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

final states:

- NFA has a specific task, "scanning" the stack, not scanning the input
- 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)

NFA: parentheses

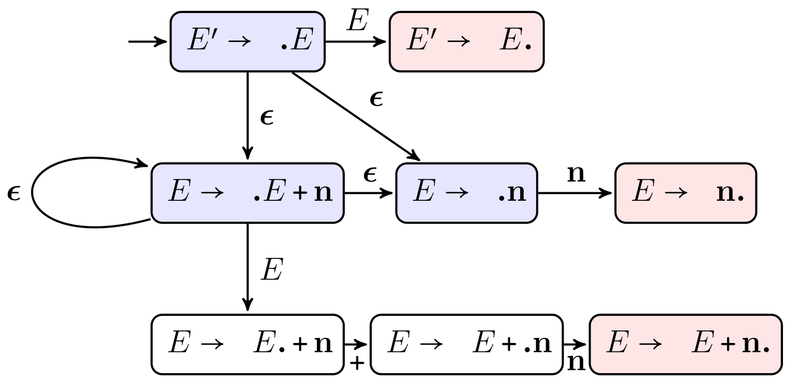


Remarks on the NFA

- colors for illustration
  - "reddish": complete items
  - "blueish": init-item (less important)
  - "violet'tish": both
- init-items
  - one per production of the grammar
  - that's where the ϵ-transistions go into, but
  - *with exception* of the initial state (with S'-production)

  no outgoing edges from the complete items

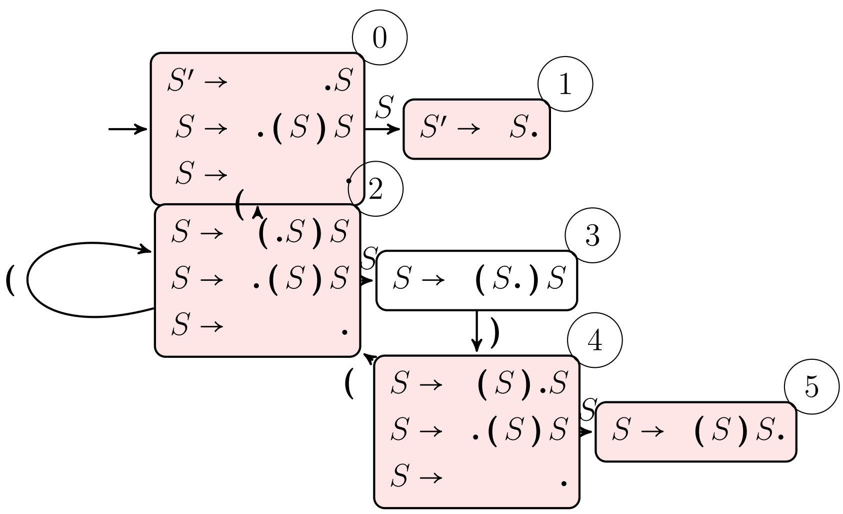
NFA: addition



Determinizing: from NFA to DFA

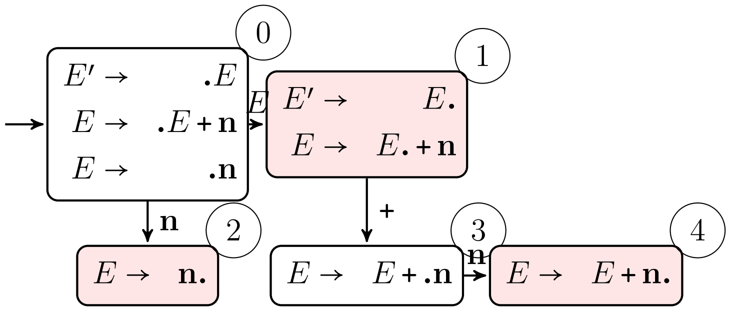
- standard subset-construction[15]
- states then contains *sets* of items
- especially important: ϵ-closure
- also: *direct* construction of the DFA possible

DFA: parentheses



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

DFA: addition



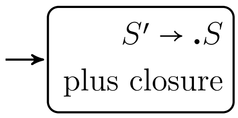
Direct construction of an LR(0)-DFA

- quite easy: simply build in the closure already

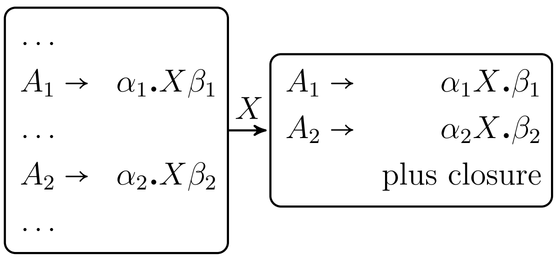
ϵ-closure

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

initial state



Direct DFA construction: transitions



- 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
- re-check the previous examples: 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)

Determinism

and the reaction better be uniquely determined . . . .

Stack contents and state of the automaton

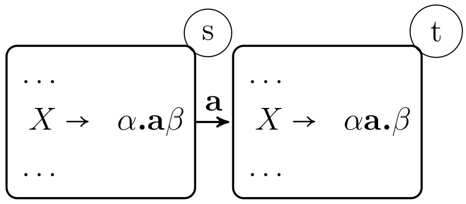
- remember: at any given intermediate configuration of stack/input in a run
  1. stack contains words from Σ^*
  2. DFA operates deterministically on such words
- the stack contains the "past": read input (potentially partially reduced)
- when feeding that "past" on the stack into the automaton
  – starting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



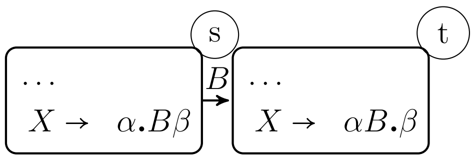
- shift is possible
- if shift is *the* correct operation and **a** is terminal symbol corresponding to the current token: state afterwards = t

State transition: analogous for non-terminals

Production

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

Transition



Explanation

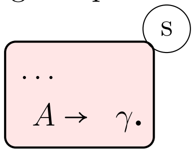
- same as before, now with non-terminal B
- note: we never read non-term from input
- not officially called a shift
- corresponds to the reaction **followed by** a *reduce* step, it's **not** the reduce step itself
- think of it as folllows: reduce and subsequent step
  - not as: *replace* on top of 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* (those with a dot **.** at the end)
- 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:[16]
  - **new top state**!
  - remember the "goto-transition" (shift of a non-terminal)

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

[16]Indirectly only: 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.

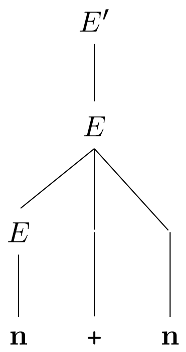
Rest

- if the (rhs of the) handle is *removed* from top stack: $\Rightarrow$
  - "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

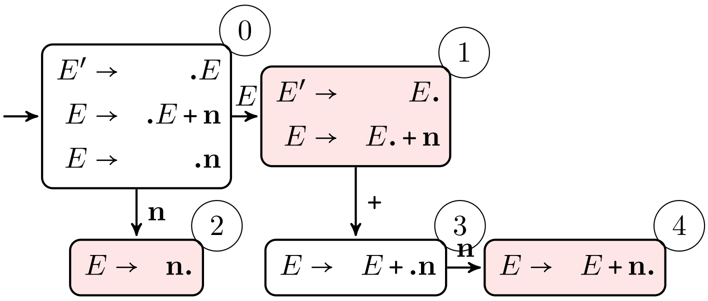


Run

	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

DFA of addition example



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

LR(0) grammars

LR(0) grammar

The top-state alone determines the next step.

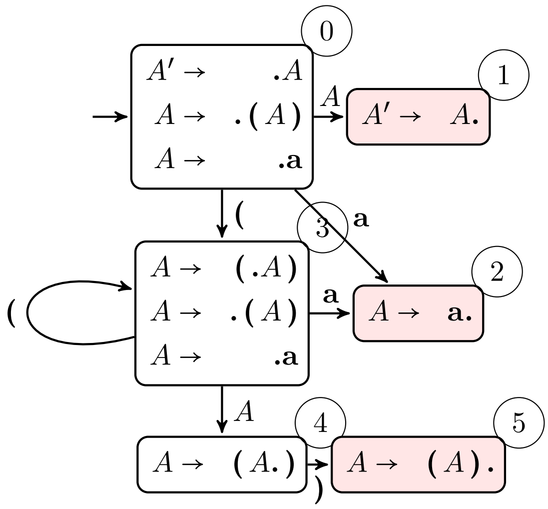
No LR(0) here

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

Simple parentheses

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

DFA

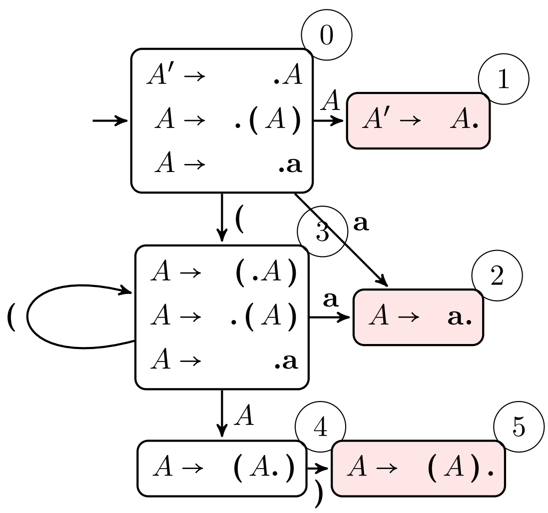


Remaks

- for *shift*:
  - **many** shift transitions in 1 state allowed
  - shift counts as *one* action (including "shifts" on non-terms)
- but for reduction: also the *production* must be clear

Simple parentheses is LR(0)

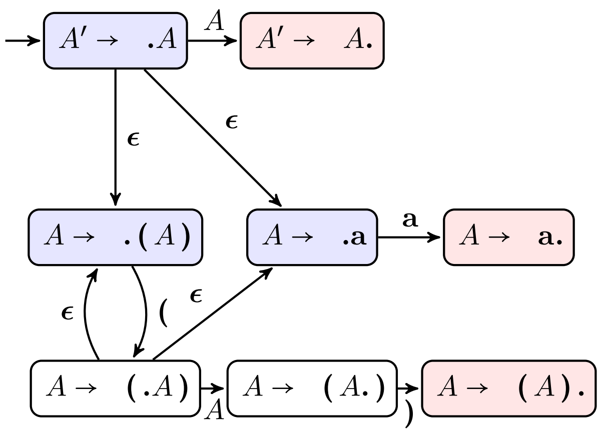
DFA



Remaks

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

NFA for simple parentheses (bonus slide)



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
- see the parser run on the next slide

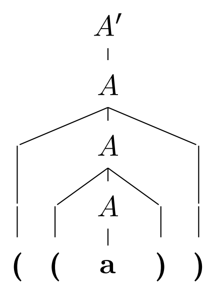
state	action	rule	input			goto
			(	**a**	**)**	A
0	shift		3	2		1
1	reduce	$A' \to A$				
2	reduce	$A \to \mathbf{a}$				
3	shift		3	2		4
4	shift				5	
5	reduce	$A \to \mathbf{(}\, A \,\mathbf{)}$				

Parsing of ((a))

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

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	$\$_0\mathbf{(}_3$	$\mathbf{(\,a\,)\,\$}$	shift
3	$\$_0\mathbf{(}_3\mathbf{(}_3$	$\mathbf{a\,)\,\$}$	shift
4	$\$_0\mathbf{(}_3\mathbf{(}_3\mathbf{a}_2$	$\mathbf{)\,\$}$	reduce $A \to \mathbf{a}$
5	$\$_0\mathbf{(}_3\mathbf{(}_3A_4$	$\mathbf{)\,\$}$	shift
6	$\$_0\mathbf{(}_3\mathbf{(}_3A_4\mathbf{)}_5$	$\$$	reduce $A \to \mathbf{(}\,A\,\mathbf{)}$
7	$\$_0\mathbf{(}_3A_4$	$\$$	????

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

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)

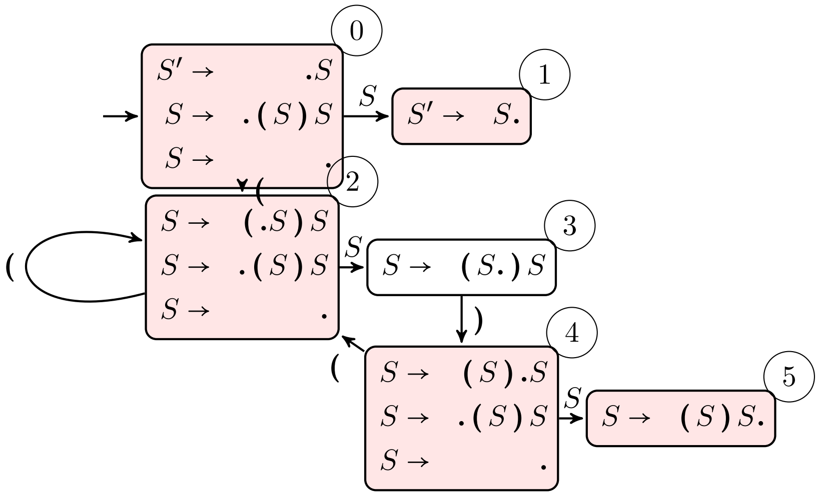
LR(0) parsing algo remarks

- in [6]: slightly differently formulated
- instead of requiring (in the first case):
  - push state t were $s \xrightarrow{X} t$ or similar, book formulates
  - push *state containing item* $A \to \alpha.X\beta$
- analogous in the second case
- algo (= deterministic) only if LR(0) grammar

- in particular: cannot have states with *complete item* and item of form $A\alpha.X\beta$ (otherwise **shift-reduce** conflict)
- cannot have states with two X-successors (known as **reduce-reduce** conflict)

DFA parentheses again: LR(0)?

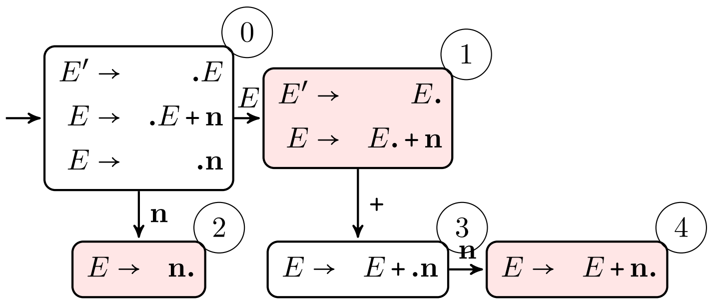
$$
\begin{array}{rcl}
S' & \to & S \\
S & \to & (\,S\,)\,S \mid \epsilon
\end{array}
$$



Look at states 0, 2, and 4

DFA addition again: LR(0)?

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

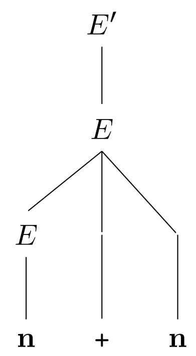


How to make a decision in state 1*?*

Decision? If only we knew the ultimate tree already . . .

. . . especially 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

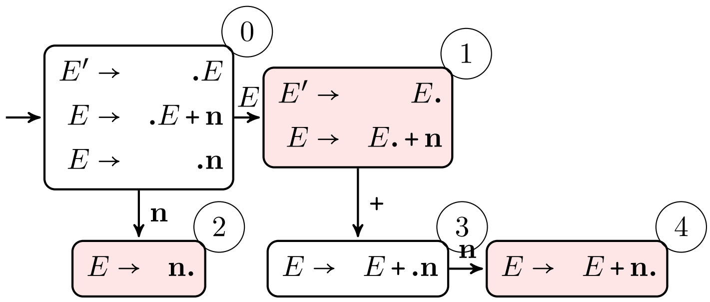
Explanation

- 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 as 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:

$$\boxed{\begin{array}{r} \ldots \\ A \to \alpha. \\ \ldots \\ B \to \beta. \end{array}}$$

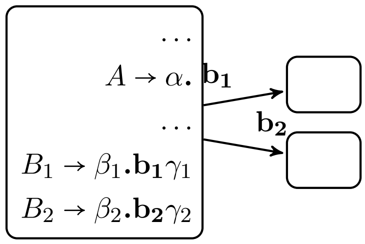
SLR(1) solution: use follow sets of non-terms

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

⇒ 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:



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

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

$\Rightarrow$ 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` $\in \{\mathbf{b_1}, \mathbf{b_2}, \dots\}$ then *shift*. Input symbol $\mathbf{b_i}$ determines new top state
  - ...

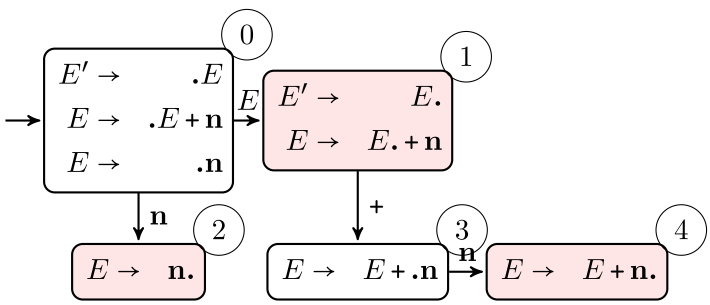
SLR(1) requirement on states (as in the book)

- 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, $\mathit{Follow}(\alpha) \cap \mathit{Follow}(\beta) = \varnothing$

Revisit addition one more time



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

$\Rightarrow$
  - 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 $s \xrightarrow{X} t$[17]
2. s contains a *complete* item (say $A \to \gamma.$) **and the next token in the input is in $Follow(A)$**: *reduce* by rule $A \to \gamma$:
   - A reduction by $S' \to S$: *accept*, if input is empty[18]
   - else:

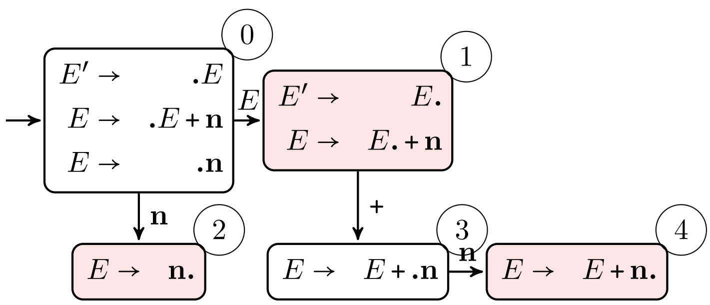
     pop: remove γ (including "its" states from the stack)

     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*

Repeat frame: given DFA

Parsing table for SLR(1)



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

[18] 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 $\$$

state	input			goto
	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 SRL(1) *shift-reduce* conflicts
  - would result in non-unique entries in SRL(1)-table[19]

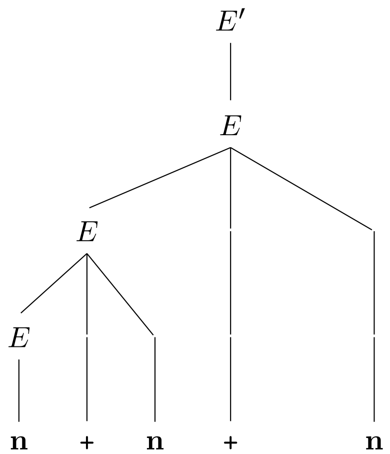
SLR(1) parser run (= "reduction")

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

[19] by which it, strictly speaking, would no longer be an SRL(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)?

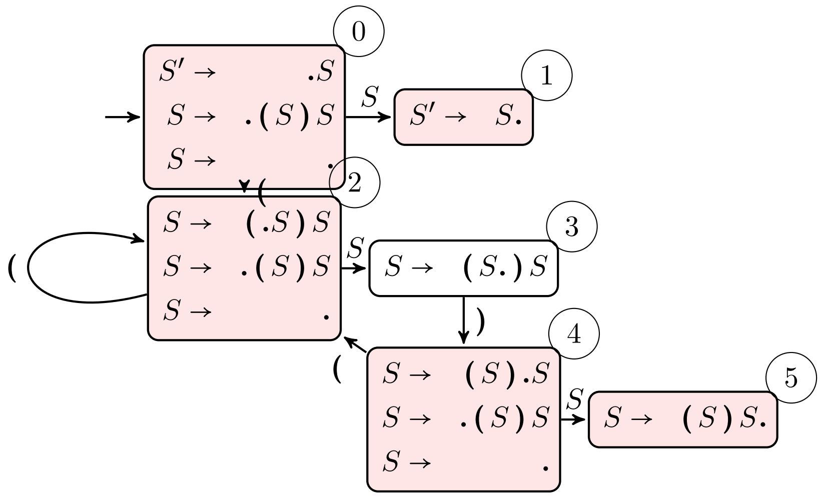
Grammar: parentheses (from before)

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

Follow set

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

DFA



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	$\$_0(_2 S_3$	$)\,(\,)\,\$$	shift: 4
4	$\$_0(_2 S_3)_4$	$(\,)\,\$$	shift: 2
5	$\$_0(_2 S_3)_4(_2$	$)\,\$$	reduce: $S \to \epsilon$
6	$\$_0(_2 S_3)_4(_2 S_3$	$)\,\$$	shift: 4
7	$\$_0(_2 S_3)_4(_2 S_3)_4$	$\$$	reduce: $S \to \epsilon$
8	$\$_0(_2 S_3)_4(_2 S_3)_4 S_5$	$\$$	reduce: $S \to (\,S\,)\,S$
9	$\$_0(_2 S_3)_4 S_5$	$\$$	reduce: $S \to (\,S\,)\,S$
10	$\$_0 S_1$	$\$$	accept

Remarks

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

SLR(k)

- in principle: straightforward: k look-ahead, instead of 1
- rarely used in practice, using $First_k$ and $Follow_k$ instead of the $k = 1$ versions
- tables grow *exponentially* with k!

As with other parsing algorithms, the SLR(1) parsing algorithm can be extended to SLR(k) parsing where parsing actions are based on $k \geq 1$ symbols of lookahead. Using the sets First$_k$ and Follow$_k$ as defined in the previous chapter, an SLR(k) parser uses the following two rules:

1. If state s contains an item of the form $A \rightarrow \alpha.X\,\beta$ (X a token), and $Xw \in$ First$_k(X\,\beta)$ are the next k tokens in the input string, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item $A \rightarrow \alpha\,X\,.\,\beta$.
2. If state s contains the complete item $A \rightarrow \alpha.$, and $w \in$ Follow$_k(A)$ are the next k tokens in the input string, then the action is to reduce by the rule $A \rightarrow \alpha$.

SLR(k) parsing is more powerful than SLR(1) parsing when $k > 1$, but at a substantial cost in complexity, since the parsing table grows exponentially in size with k.

Ambiguity & LR-parsing

- in principle: 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 "outside" the grammar
2. by "living with the fact" that LR parser (commonly) *prioritizes shifts over reduces*

Rest

- 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

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

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

Simplified conditionals

Simplified "schematic" if-then-else

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

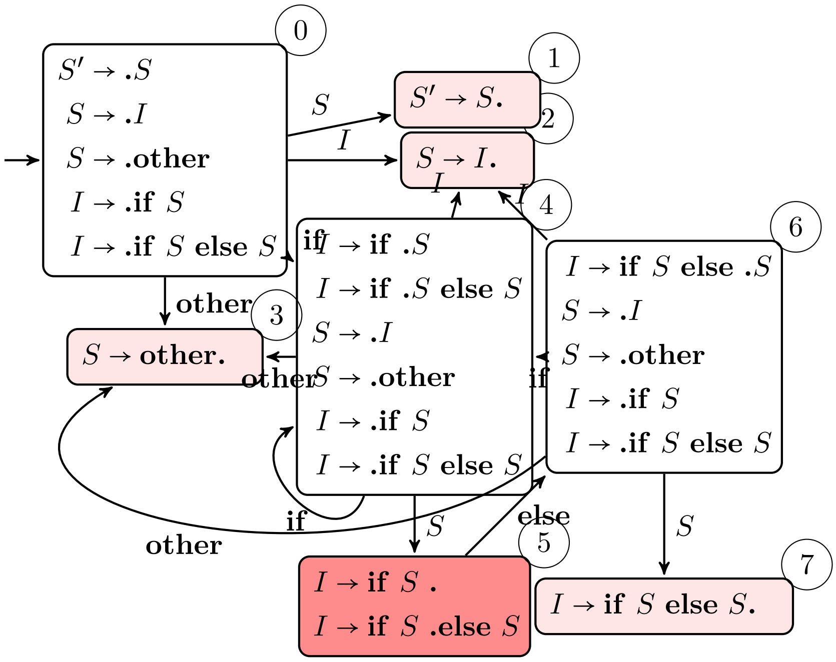
Follow-sets

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

Rest

- since ambiguous: at least one conflict must be somewhere

DFA of LR(0) items



Simple conditionals: parse table

Grammar

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

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		$r:4$		$r:4$		

Explanation

- *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)

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

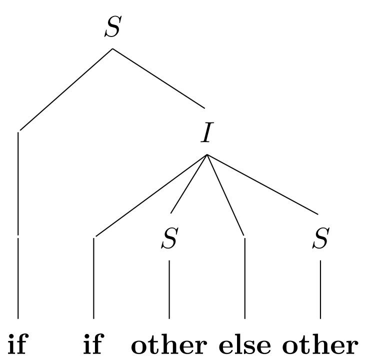
Parser run, different choice

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		$r:4$		$r:4$		

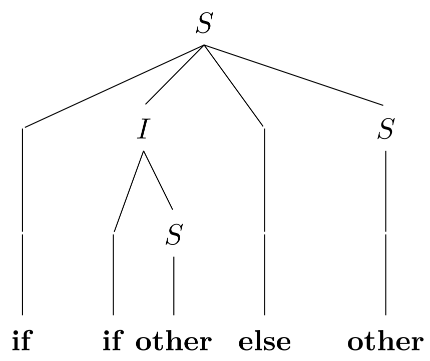
stage	parsing stack	input	action
1	$\$_0$	**if if other else other $**	shift: 4
2	$\$_0$**if**$_4$	**if other else other $**	shift: 4
3	$\$_0$**if**$_4$**if**$_4$	**other else other $**	shift: 3
4	$\$_0$**if**$_4$**if**$_4$**other**$_3$	**else other $**	reduce: 2
5	$\$_0$**if**$_4$**if**$_4 S_5$	**else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

shift-precedence: conventional



"wrong" tree



standard "dangling else" convention

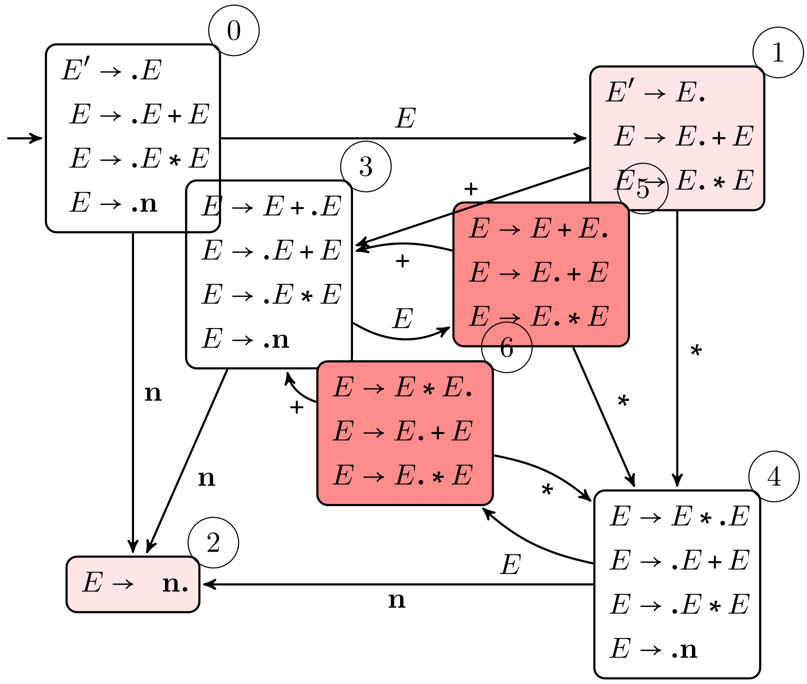
"an **else** belongs to the last previous, still open (= dangling) if-clause"

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` ...

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

DFA for $+$ and $\times$



States with conflicts

- state 5
  - stack contains \$... E +E\$
  - for input **\$**: reduce, since shift not allowed from **\$**
  - 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 from **\$**
  - for input **+**; reduce, a $*$ has *precedence* over **+**
  - for input $*$: shift, 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

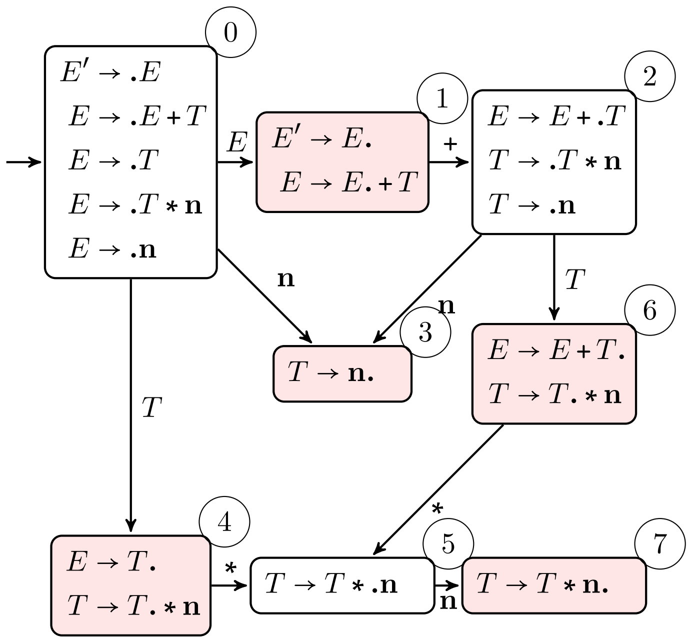
For comparison: unambiguous grammar for + and $*$

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: $Follow(E') = \{\$\}$

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

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

state 3/7: $Follow(T) = \{\$, +, *\}$

- 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

SRL(1) "improved" LR(0) parsing LALR(1) is "crippled" LR(1) parsing.

Limits of SLR(1) grammars

Assignment grammar fragment[20]

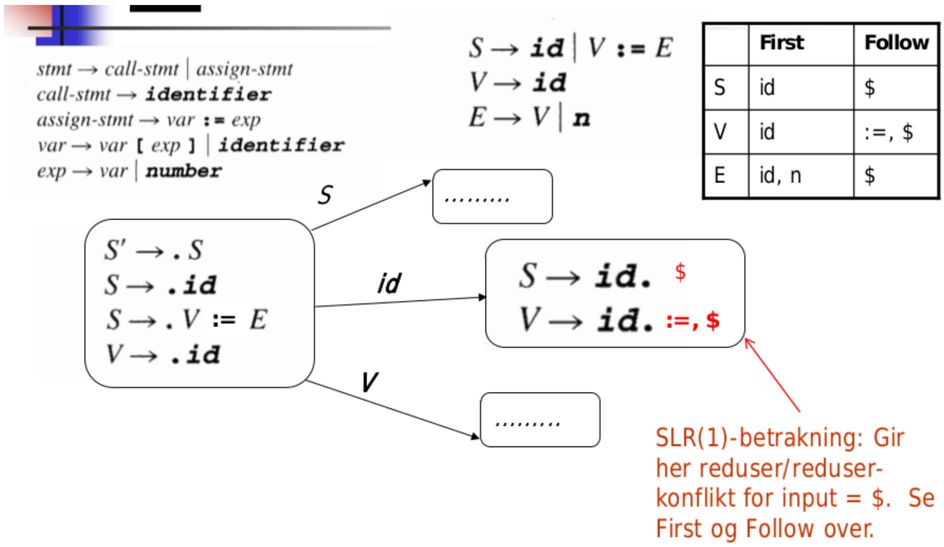
$$
\begin{array}{rcl}
\mathit{stmt} & \to & \mathit{call\text{-}stmt} \mid \mathit{assign\text{-}stmt} \\
\mathit{call\text{-}stmt} & \to & \textbf{identifier} \\
\mathit{assign\text{-}stmt} & \to & \mathit{var} := \mathit{exp} \\
\mathit{var} & \to & [\,\mathit{exp}\,] \mid \textbf{identifier} \\
\mathit{exp}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

Assignment grammar fragment, simplified

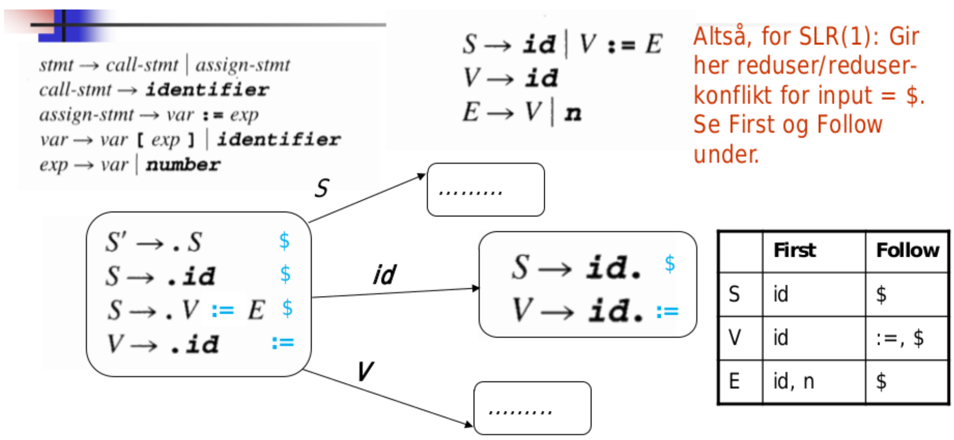
$$
\begin{array}{rcl}
S & \to & \textbf{id} \mid V := E \\
V & \to & \textbf{id} \\
E & \to & V \mid \textbf{n}
\end{array}
$$

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

non-SLR(1): Reduce/reduce conflict



Situation can be saved: more look-ahead



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[21] due to subset construction
  - the items are LR(0)-items
  - follow-sets as an *after-thought*

[21] 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 disambiguate with the help of the follow sets for states containing complete items: **make more fine-grained items**:

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

LR(1) items

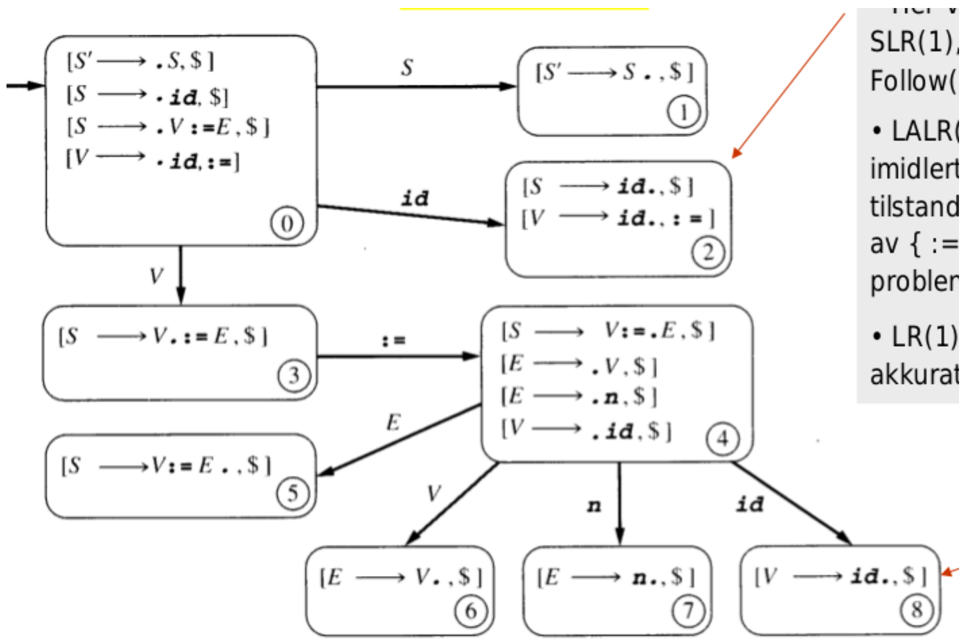
- main idea: simply make the look-ahead part of the item
- obviously: proliferation of states[22]

LR(1) items

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

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

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



[22] 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

$$\boxed{[A \to \quad \alpha.X\beta, \mathbf{a}]} \xrightarrow{X} \boxed{[A \to \quad \alpha X.\beta, \mathbf{a}]}$$

LR(1) transitions: ϵ

ϵ-transition

for all

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

$$\boxed{[A \to \alpha.B\gamma \quad ,\mathbf{a}]} \xrightarrow{\epsilon} \boxed{[B \to .\beta \quad ,\mathbf{b}]}$$

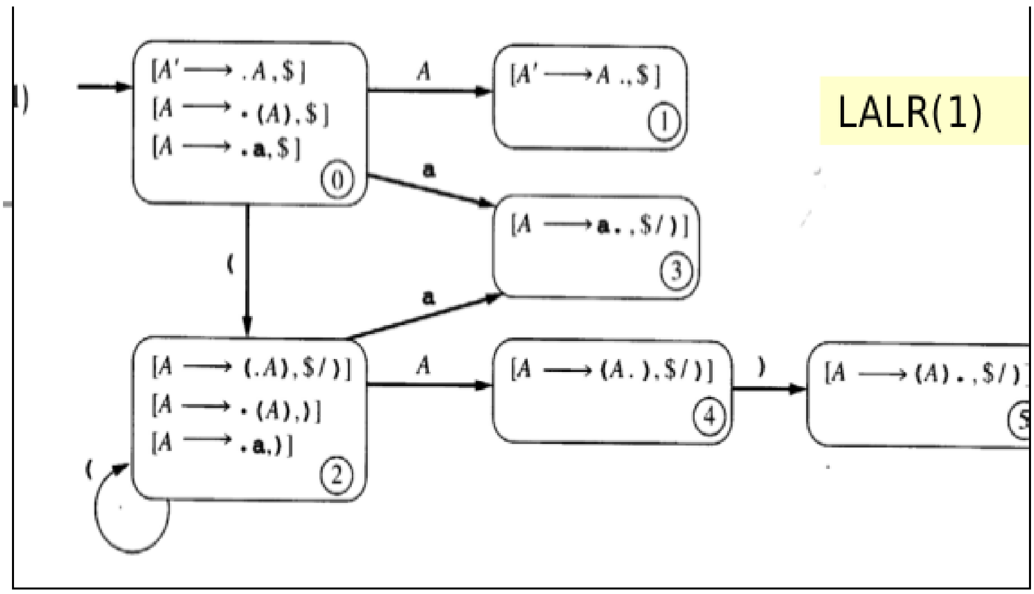
including special case ($\gamma = \epsilon$)

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

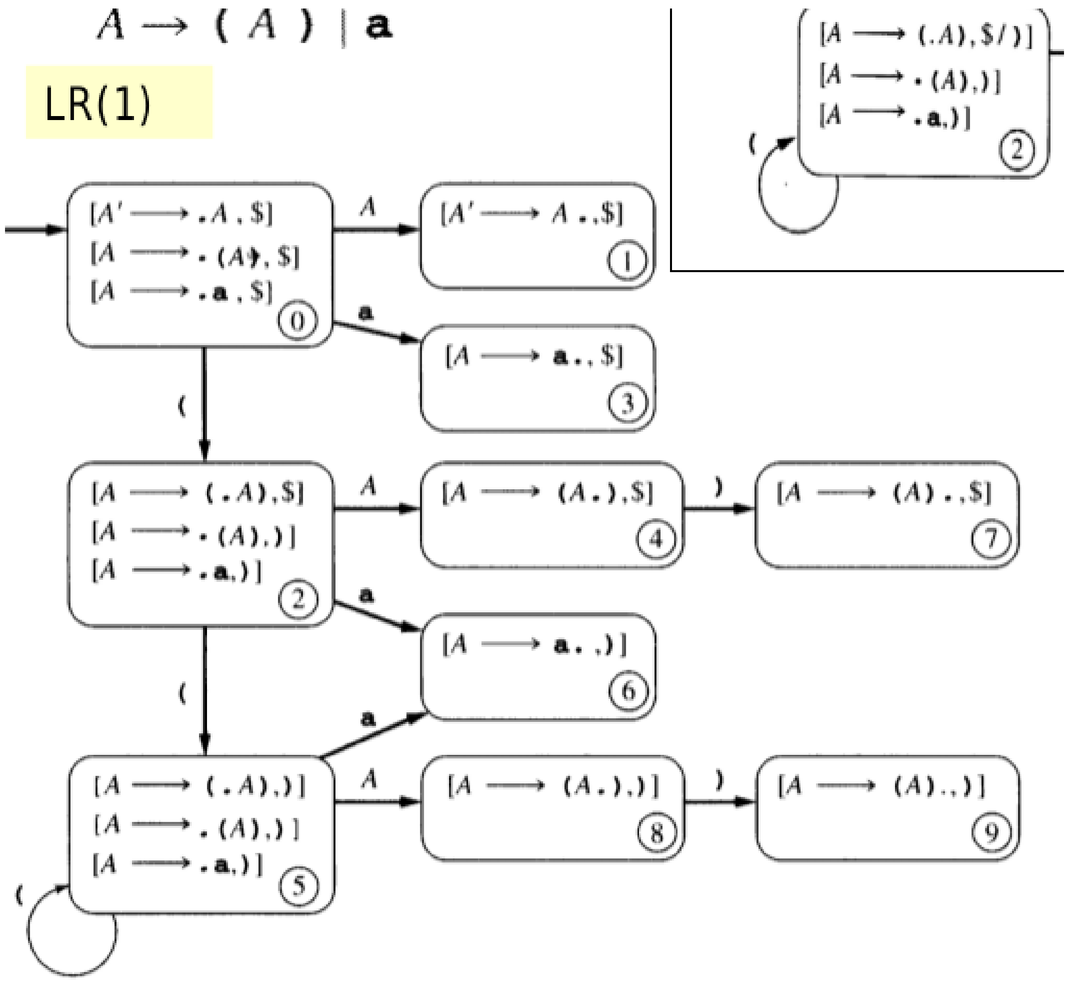
$$\boxed{[A \to \alpha.B \quad ,\mathbf{a}]} \xrightarrow{\epsilon} \boxed{[B \to .\beta \quad ,\mathbf{a}]}$$

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)

Rest

- 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[23]
  - 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

[23] 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.

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

Again: Error handling

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

Rest

- 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 : `left 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 left paranthesis 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	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3(_4\mathbf{d}_5\mathbf{e}_6$	**f**) **gh** . . . \$	no entry for **f**
2	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3B_v$	**gh** . . . \$	back to normal
3	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3B_v\mathbf{g}_7$	**h** . . . \$	. . .

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

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

Typical yacc parser table

some variant of the expression grammar again

$$
\begin{array}{rcl}
command & \rightarrow & exp \\
exp & \rightarrow & term * factor \mid factor \\
term & \rightarrow & term * factor \mid factor \\
factor & \rightarrow & \mathbf{n} \mid (\, exp\,)
\end{array}
$$

State	Input							Goto			
	NUMBER	(	+	-	*	)	$	*command*	*exp*	*term*	*factor*
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		s14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

Panicking and looping

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

- 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

Bibliography

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

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

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

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

[5] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

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

Index