INF5110 – Compiler Construction

Parsing

Spring 2016

Outline

Outline

Bottom-up parsing: intro

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

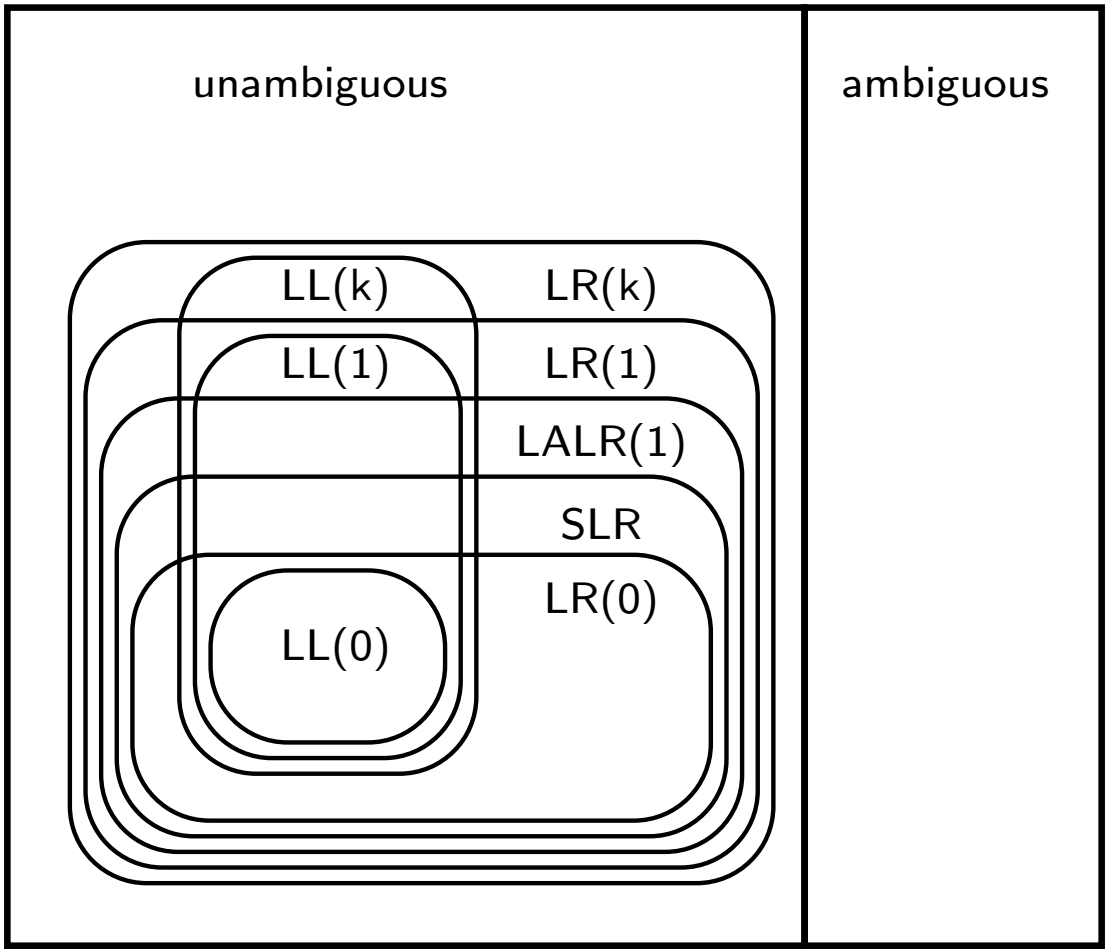
LR(0)
- only for very simple grammars
- approx 300 states for standard programming languages
- only as intro tro SLR(1) and LALR(1)

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

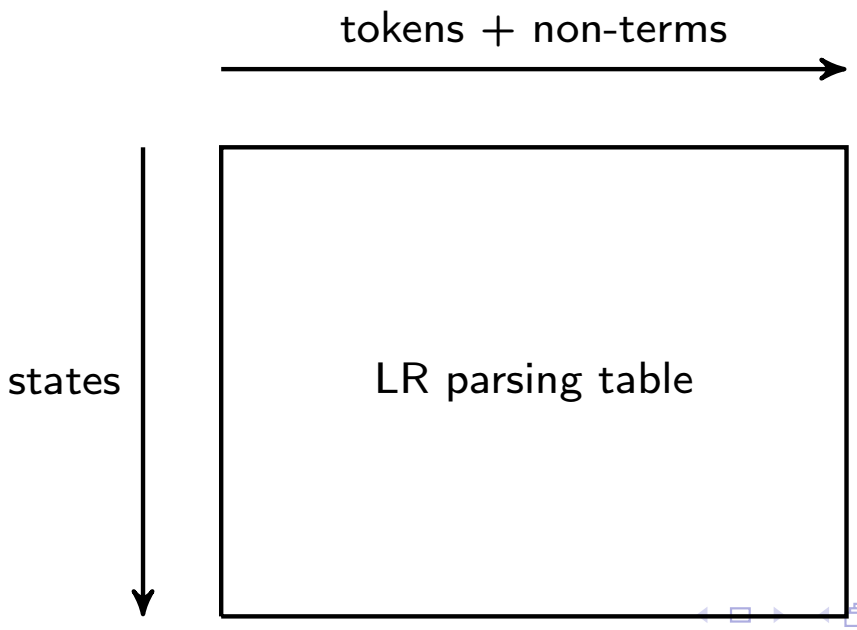
LR(1) covers all grammars, which can in principle be parsed by looking at the next token

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 it's descendants (like bison, CUP, etc)
- another name: *shift-reduce* parser



Example grammar

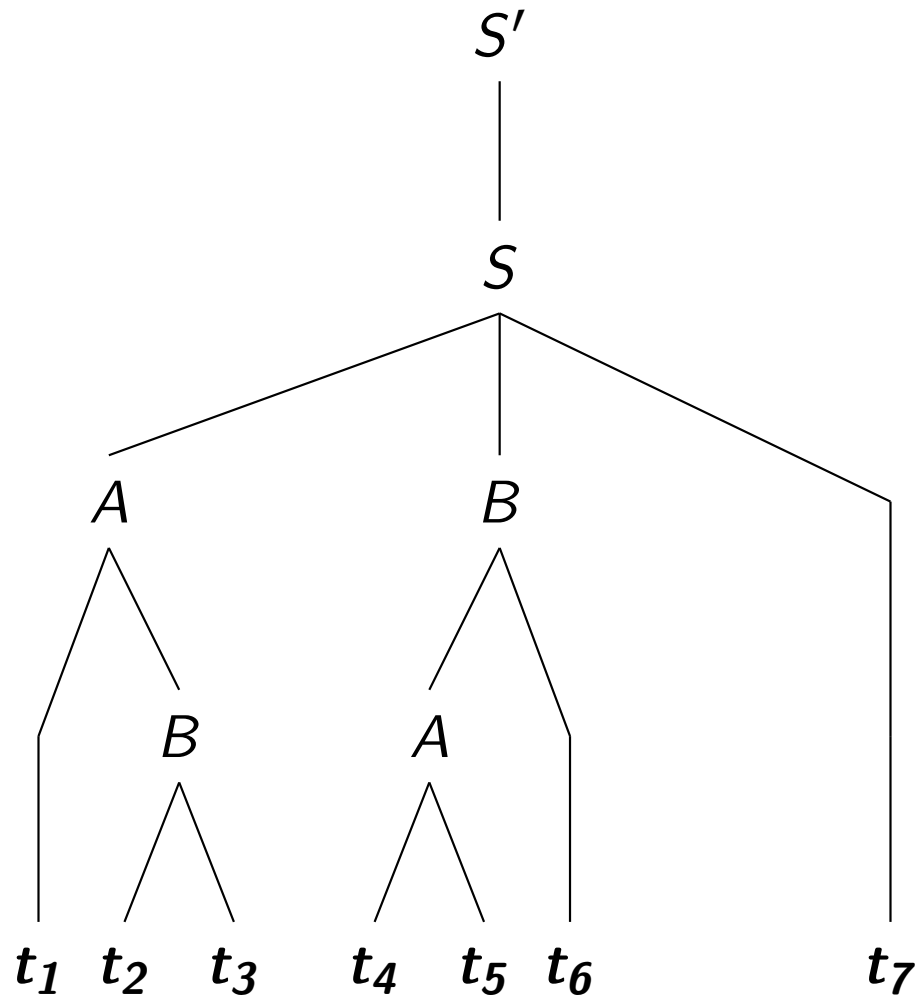
$$
\begin{array}{rcl}
S' & \to & S \\
S & \to & AB\,\mathbf{t_7} \mid \ldots \\
A & \to & \mathbf{t_4}\,\mathbf{t_5} \mid \mathbf{t_1}\,B \mid \ldots \\
B & \to & \mathbf{t_2}\,\mathbf{t_3} \mid A\,\mathbf{t_6} \mid \ldots
\end{array}
$$

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

- general agreement for bottom-up parsing:
  - start symbol *never* on the right-hand side or a production
  - routinely add another "extra" start-symbol (here S')[1]

[1] That will later be used when constructing a DFA for "scanning" the stack, to control the reactions of the stack machine. This restriction leads to a unique, well-define initial state.

Parse tree for $t_1 \ldots t_7$



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 can the parser do a *right*-most derivation, when it parses from *left*-to-right?

- short answer: parser builds the parse tree bottom-up
- derivation:
  - replacement of nonterminals by right-hand sides
  - *derivation*: builds (implicitly) a parse-tree *top-down*
- sentential form: word from Σ^* derivable from start-symbol

Right-sentential form: right-most derivation

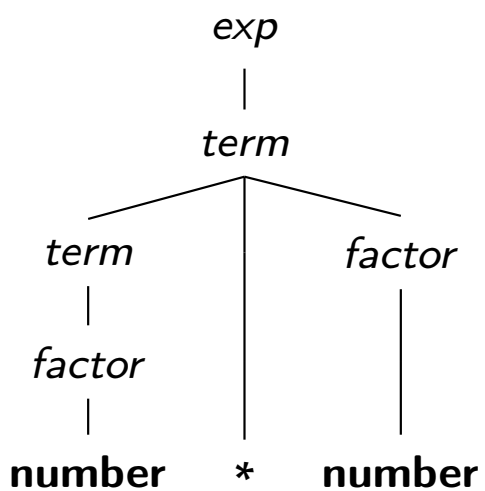
$$S \Rightarrow_r^* \alpha$$

Slighly longer answer

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

Example expression grammar (from before)

$$
\begin{array}{rcll}
\mathit{exp} & \rightarrow & \mathit{expaddopterm} \mid \mathit{term} & (1) \\
\mathit{addop} & \rightarrow & + \mid - & \\
\mathit{term} & \rightarrow & \mathit{termmulopterm} \mid \mathit{factor} & \\
\mathit{mulop} & \rightarrow & * & \\
\mathit{factor} & \rightarrow & (\ \mathit{exp}\ ) \mid \textbf{number} &
\end{array}
$$

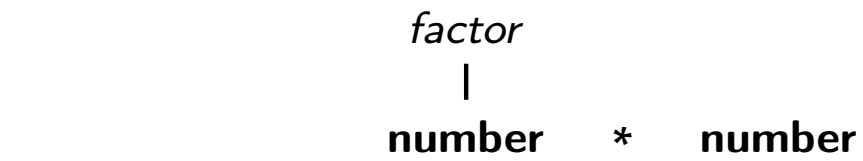


Bottom-up parse: Growing the parse tree

number * **number**

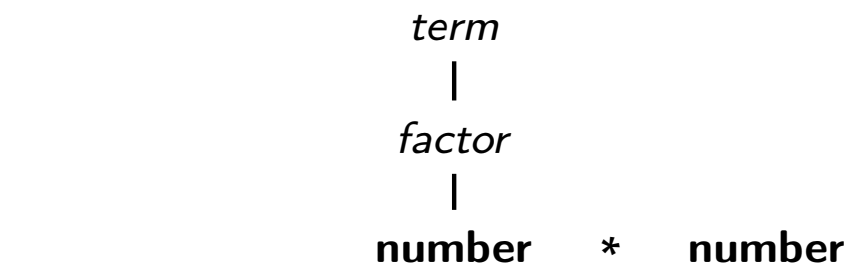
number * **number**

Bottom-up parse: Growing the parse tree



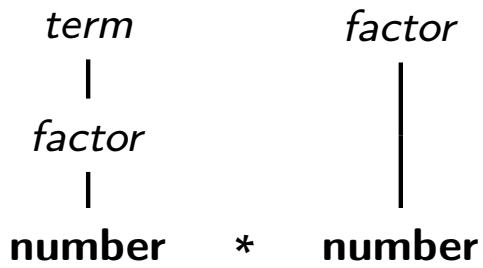
$$\underline{\textbf{number}} * \textbf{number} \quad \hookrightarrow \quad \underline{\textit{factor}} * \textbf{number}$$

Bottom-up parse: Growing the parse tree



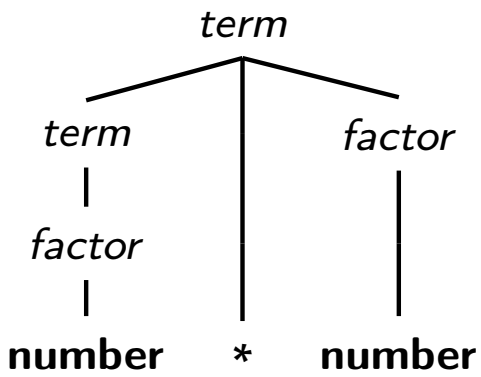
$$\underline{\textbf{number}} * \textbf{number} \hookrightarrow \underline{\textit{factor}} * \textbf{number} \hookrightarrow \textit{term} * \underline{\textbf{number}}$$

Bottom-up parse: Growing the parse tree



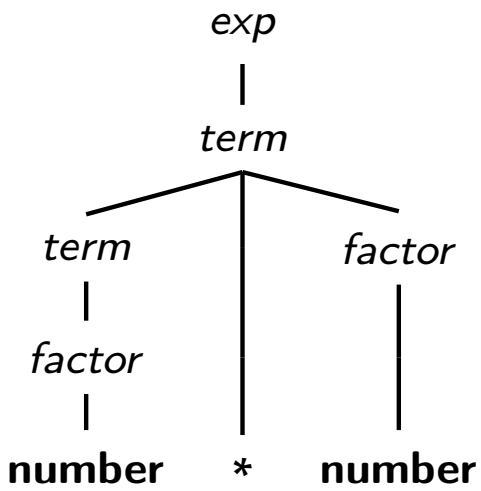
$$\underline{\textbf{number}} * \textbf{number} \hookrightarrow \underline{\textit{factor}} * \textbf{number} \hookrightarrow \textit{term} * \underline{\textbf{number}} \hookrightarrow \underline{\textit{term} * \textit{factor}}$$

Bottom-up parse: Growing the parse tree



$$\underline{\textbf{number}} * \textbf{number} \hookrightarrow \underline{\textit{factor}} * \textbf{number} \hookrightarrow \textit{term} * \underline{\textbf{number}} \hookrightarrow \underline{\textit{term} * \textit{factor}} \hookrightarrow \underline{\textit{term}}$$

Bottom-up parse: Growing the parse tree



$$\underline{\textbf{number}} * \textbf{number} \hookrightarrow \underline{\textit{factor}} * \textbf{number} \hookrightarrow \textit{term} * \underline{\textbf{number}} \hookrightarrow \underline{\textit{term} * \textit{factor}} \hookrightarrow \underline{\textit{term}} \hookrightarrow \textit{exp}$$

Reduction in reverse = right derivation

Reduction

$$
\begin{aligned}
\underline{\boldsymbol{n}} * \boldsymbol{n} &\hookrightarrow \underline{\mathit{factor}} * \boldsymbol{n} \\
&\hookrightarrow \mathit{term} * \underline{\boldsymbol{n}} \\
&\hookrightarrow \underline{\mathit{term} * \mathit{factor}} \\
&\hookrightarrow \underline{\mathit{term}} \\
&\hookrightarrow \mathit{exp}
\end{aligned}
$$

Right derivation

$$
\begin{aligned}
\boldsymbol{n} * \boldsymbol{n} &\Leftarrow_r \underline{\mathit{factor}} * \boldsymbol{n} \\
&\Leftarrow_r \underline{\mathit{term}} * \boldsymbol{n} \\
&\Leftarrow_r \mathit{term} * \underline{\mathit{factor}} \\
&\Leftarrow_r \underline{\mathit{term}} \\
&\Leftarrow_r \underline{\mathit{exp}}
\end{aligned}
$$

- 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
- note: all intermediate words are *right-sentential forms*

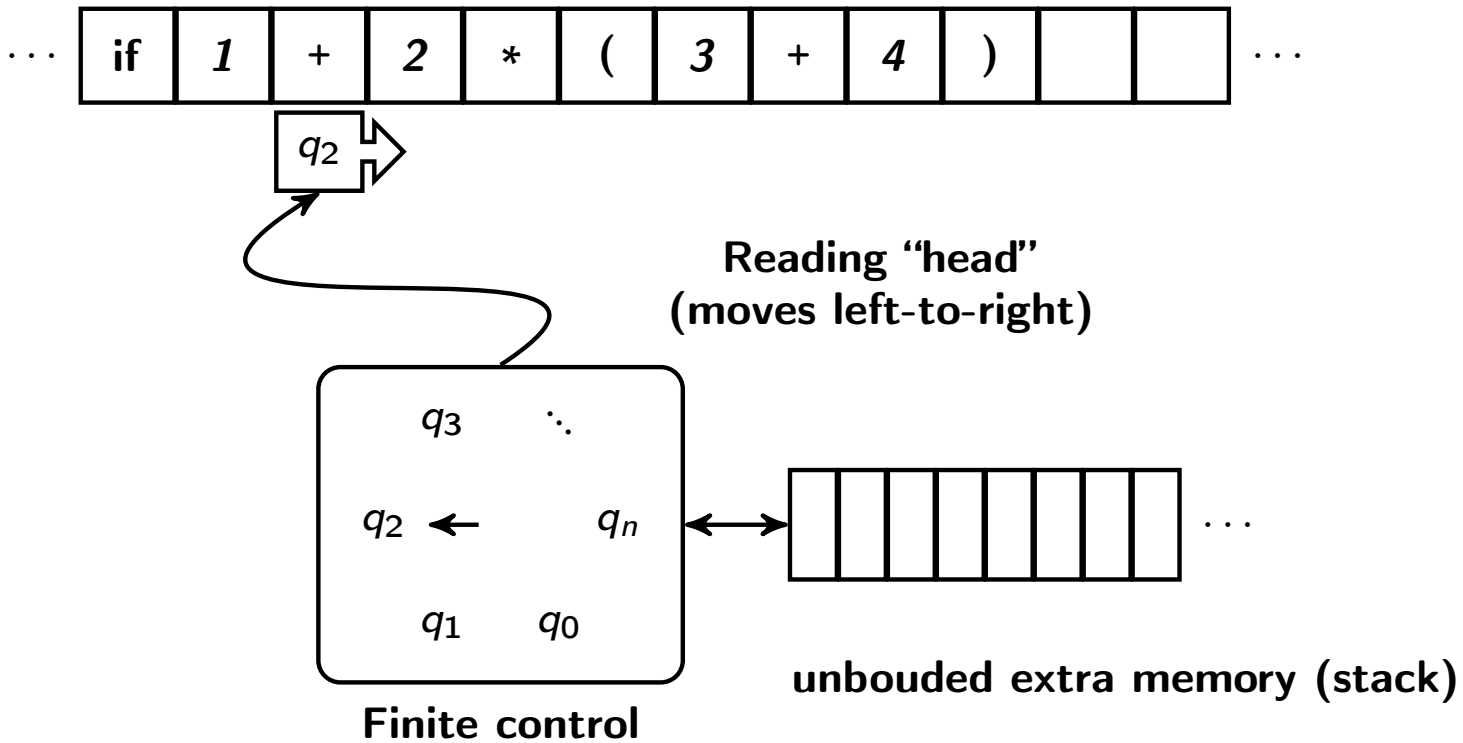
Definition (Handle)

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.
- 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" 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 will be: how do achieve the same *without that tree already given* (just parsing left-to-right)

Schematic run (reduction: from top to bottom)

$\$$	$t_1 t_2 t_3 t_4 t_5 t_6 t_7 \$$
$\$ t_1$	$t_2 t_3 t_4 t_5 t_6 t_7 \$$
$\$ t_1 t_2$	$t_3 t_4 t_5 t_6 t_7 \$$
$\$ t_1 t_2 t_3$	$t_4 t_5 t_6 t_7 \$$
$\$ t_1 B$	$t_4 t_5 t_6 t_7 \$$
$\$ A$	$t_4 t_5 t_6 t_7 \$$
$\$ A t_4$	$t_5 t_6 t_7 \$$
$\$ A t_4 t_5$	$t_6 t_7 \$$
$\$ A A$	$t_6 t_7 \$$
$\$ A A t_6$	$t_7 \$$
$\$ A B$	$t_7 \$$
$\$ A B t_7$	$\$$
$\$ S$	$\$$
$\$ S'$	$\$$

2 basic steps: shift and reduce

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

Shift

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

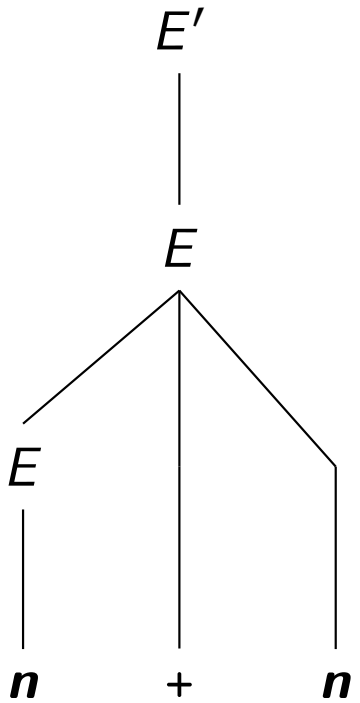
Reduce

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

- *easy* to do if one has the parse tree already!

Example: LR parsing for addition (given the tree)

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



	parse stack	input	action
1	\$	$\boldsymbol{n}+\boldsymbol{n}\$$	shift
2	$\$\boldsymbol{n}$	$+\boldsymbol{n}\$$	red:. $E \to \boldsymbol{n}$
3	$\$E$	$+\boldsymbol{n}\$$	shift
4	$\$E+$	$\boldsymbol{n}\$$	shift
5	$\$E+\boldsymbol{n}$	\$	reduce $E \to E+\boldsymbol{n}$
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"

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

Example with ϵ-transitions: parentheses

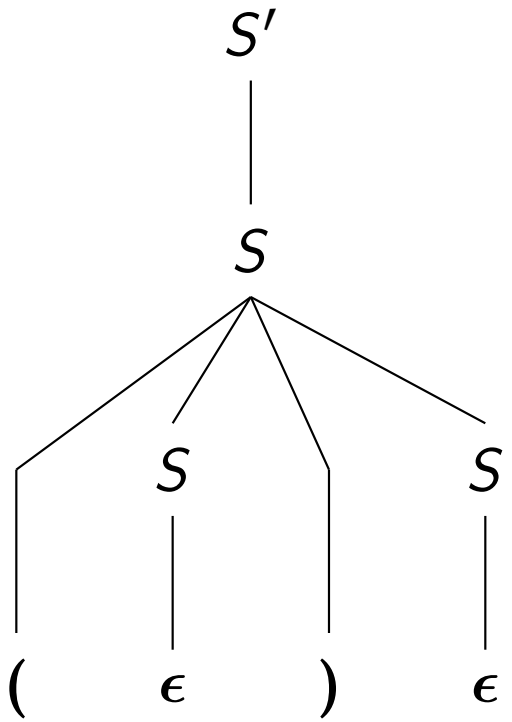
$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow (\,S\,)\,S \mid \epsilon \end{aligned}$$

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



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

Note: the 2 reduction steps for the ϵ productions

Right-most derivation and right-sentential forms

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

Right-sentential forms & the stack

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

Right-sentential form: right-most derivation

$$S \Rightarrow_r^* \alpha$$

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

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

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

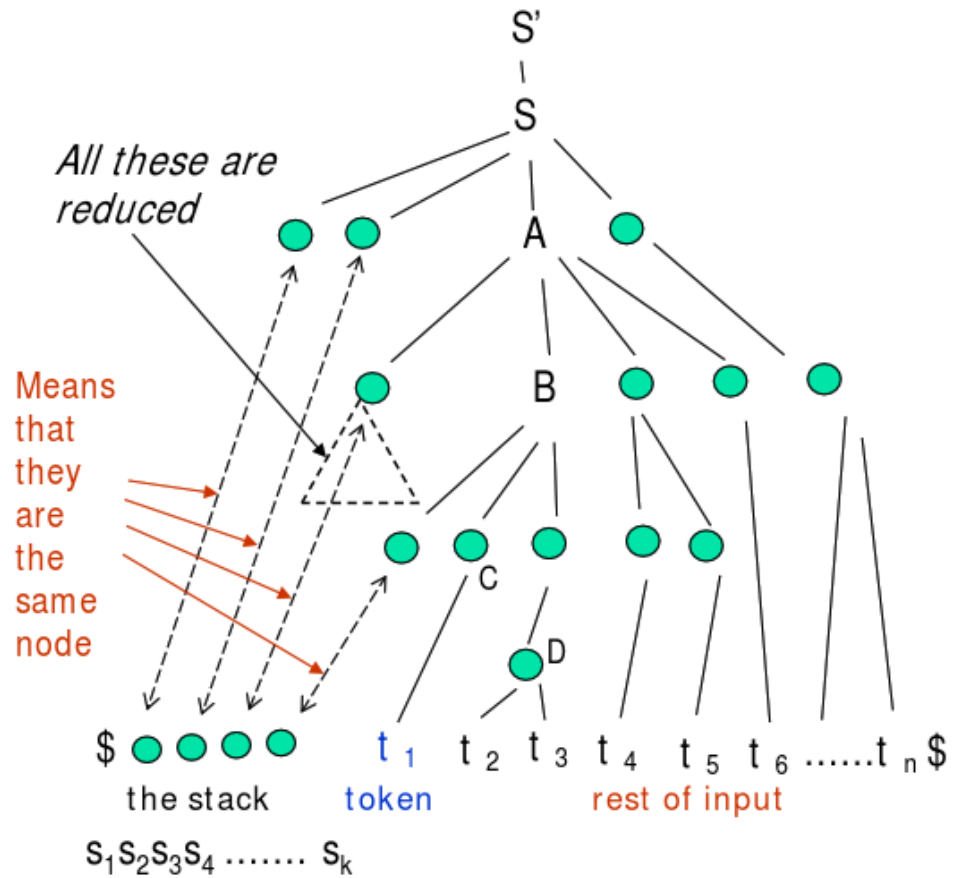
$$\underline{\boldsymbol{n}}+\boldsymbol{n} \hookrightarrow \underline{E+\boldsymbol{n}} \hookrightarrow \underline{E} \hookrightarrow E'$$

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

Viable prefixes of right-sentential forms and handles

- right-sentential form: $E + \boldsymbol{n}$
- viable prefixes of RSF
  - prefixes of that RSF *on the stack*
  - here: 3 viable prefixes of that RSF: E, $E +$, $E + \boldsymbol{n}$
- *handle*: remember the definition earlier
- here: for instance in the sentential form $\boldsymbol{n} + \boldsymbol{n}$
  - handle is production $E \to \boldsymbol{n}$ on the *left* occurrence of $\boldsymbol{n}$ in $\boldsymbol{n} + \boldsymbol{n}$ (let's write $\boldsymbol{n}_1 + \boldsymbol{n}_2$ for now)
  - note: in the stack machine:
    - the left $\boldsymbol{n}_1$ on the stack
    - rest $+ \boldsymbol{n}_2$ on the input (unread, because of LR(0))
- if the parser engine detects handle $\boldsymbol{n}_1$ on the stack, it does a *reduce*-step
- However (later): depends on current "state" of the parser engine

A typical situation during LR-parsing



The stack is reduced version of the processed input

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

C -> t1

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

D -> t2 t3

Then, what's next?

General design for an LR-engine

- some ingredients clarified up-to now:
  - bottom-up tree building as reverse right-most derivation,
  - stack vs. input,
  - shift and reduce steps
- however, one 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)

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

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

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

- **.** 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

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

- 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

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

8 items

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

- note: $S \rightarrow \epsilon$ gives $S \rightarrow .$ as item (not $S \rightarrow \epsilon.$ and $S \rightarrow .\epsilon$)
- side remark: see later, it will turn out: grammar *not LR(0)*

Another example: items for addition grammar

Grammar for addition: 3 productions

$$\begin{aligned} E' &\to E \\ E &\to E + \textbf{number} \mid \textbf{number} \end{aligned}$$

(coincidentally also:) 8 items

$$\begin{aligned} E' &\to .E \\ E' &\to E. \\ E &\to .E + \textbf{number} \\ E &\to E. + \textbf{number} \\ E &\to E + .\textbf{number} \\ E &\to E + \textbf{number}. \\ E &\to .\textbf{number} \\ E &\to \textbf{number}. \end{aligned}$$

- 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

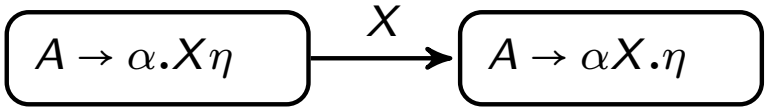
$$A \to \beta . \gamma$$

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

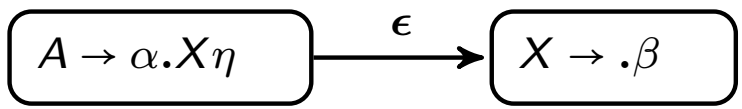
State transitions of the NFA

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

Terminal or non-terminal



Epsilon (X: non-terminal here)



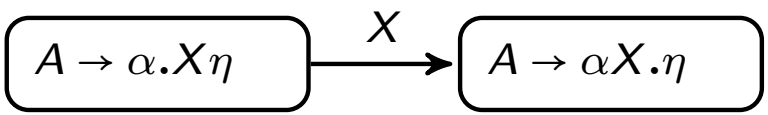
- In case $X =$ *terminal* (i.e. token) =
  - the left step corresponds to a shift step[2]
- for non-terminals (see next slide):
  - interpretation more complex: non-terminals are officially never on the input
  - note: in that case, item $A \to \alpha.X\eta$ has two (kind of) outgoing transitions

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

Transitions for non-terminals and ϵ

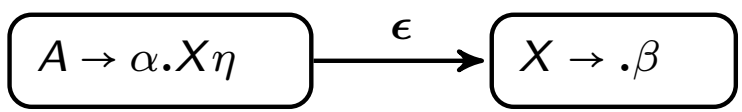
- so far: we never pushed a non-terminal from the input to the stack, we replace in a reduce-step the right-hand side by a left-hand side
- however: the replacement in a reduce steps can be seen as
  1. pop right-hand side off the stack,
  2. instead, "assume" corresponding non-terminal on input &
  3. eat the non-terminal an push it on the stack.
- two kind of transitions
  1. the ϵ-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



Epsilon (X: non-terminal here)

Given production $X \to \beta$:



Initial and final states

initial states:

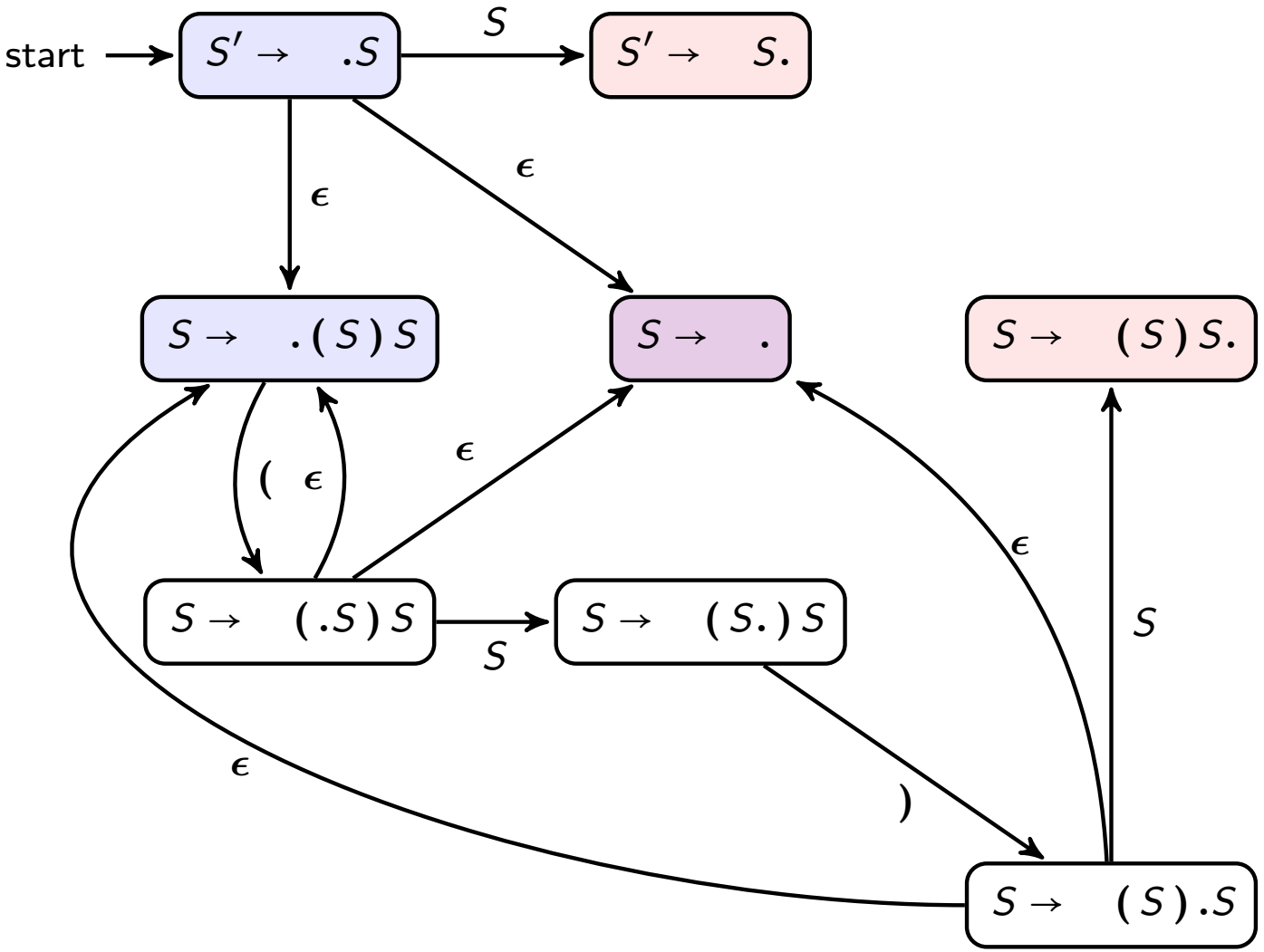
- we make our lives *easier*
- we assume (as said): one *extra* start symbol say S' (augmented grammar)

$\Rightarrow$ 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

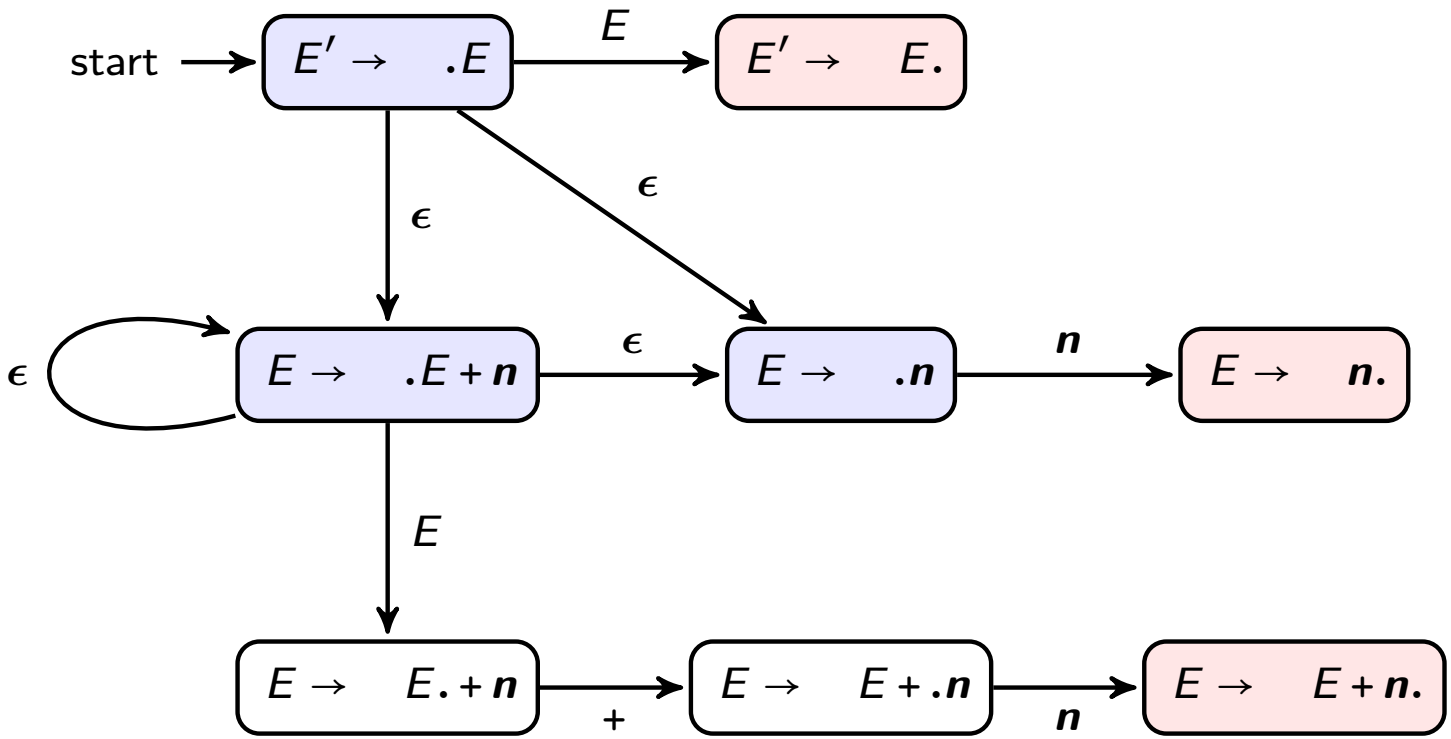


Remarks on the NFA

- 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

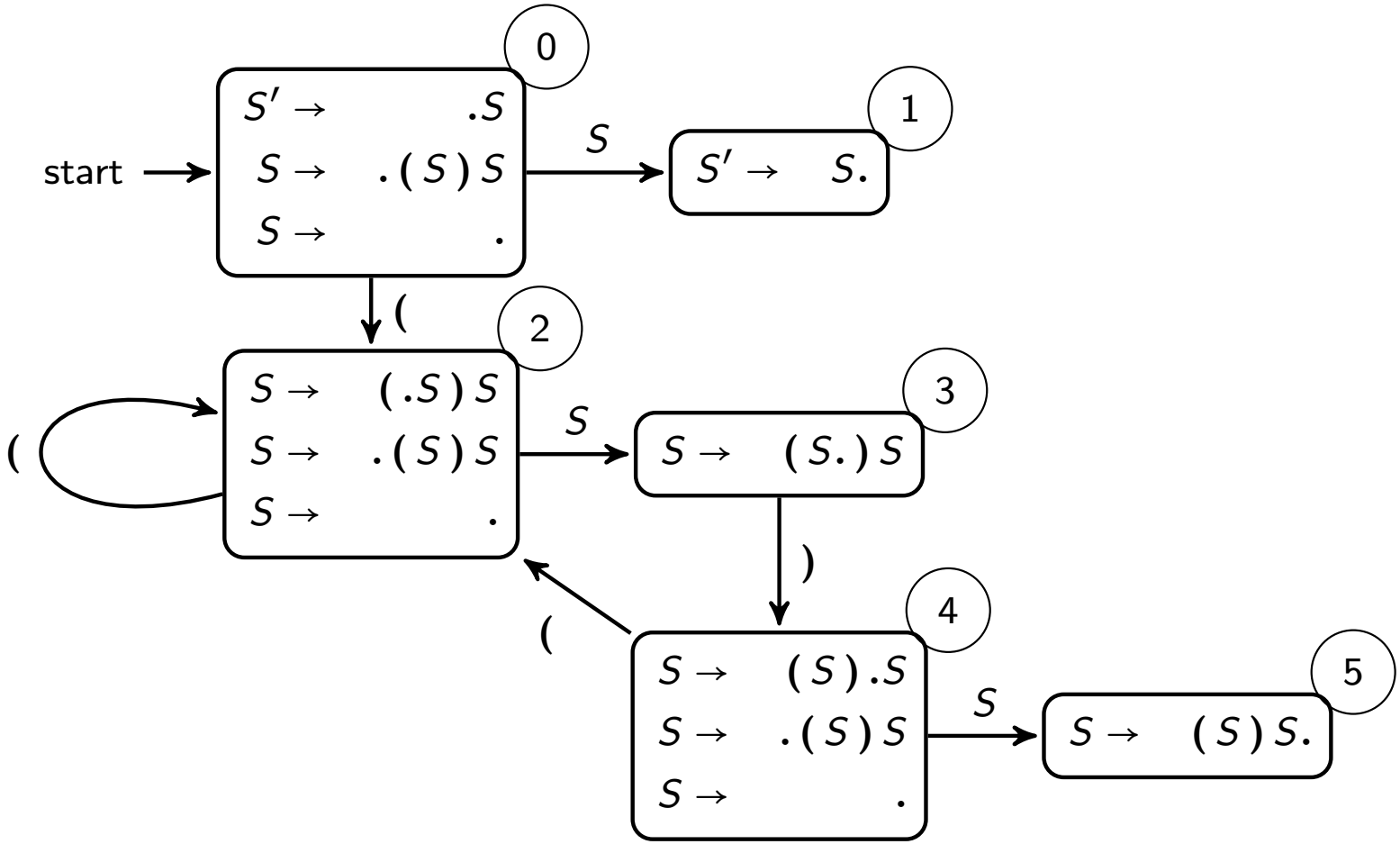


Determinizing: from NFA to DFA

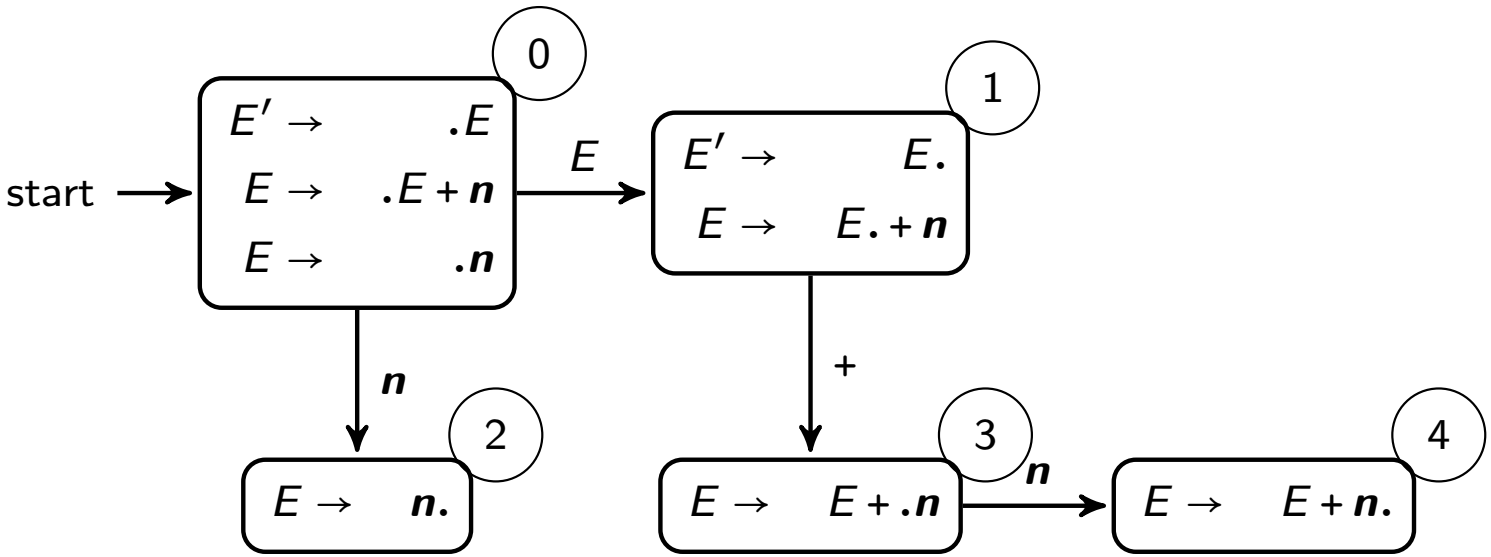
- standard subset-construction[3]
- states then contains *sets* of items
- especially important: ϵ-closure
- also: *direct* construction of the DFA possible

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

DFA: parentheses



DFA: addition



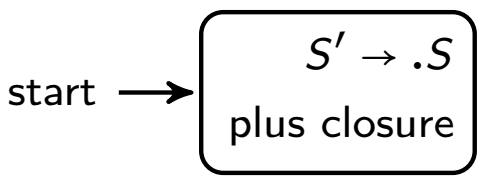
Direct construction of an LR(0)-DFA

- quite easy: simply build in the closure already

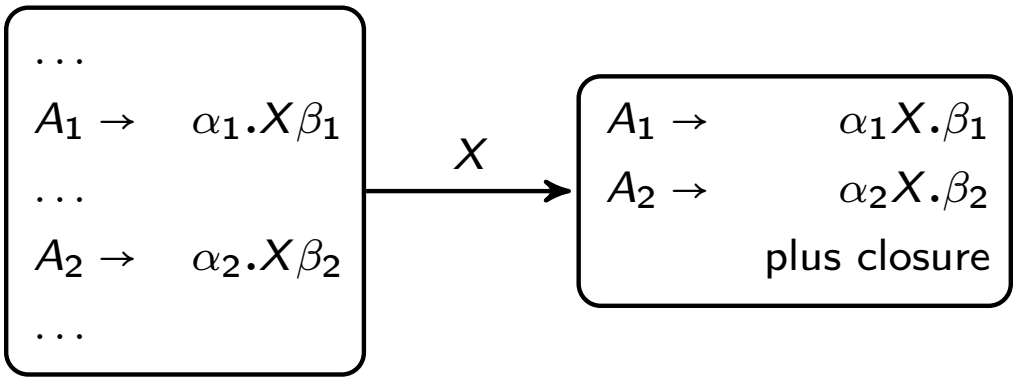
ϵ-closure

- if $A \to \alpha.B\gamma$ is an item in a state
- there is productions $B \to \beta_1 \mid \beta_2 \ldots$
- 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)

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 (and 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

  $\Rightarrow$ stack content determines state of the DFA
- actually: each prefix also determines uniquely a state
- top state:
  - state after the complete stack content
  - corresponds to the current state of the stack-machine

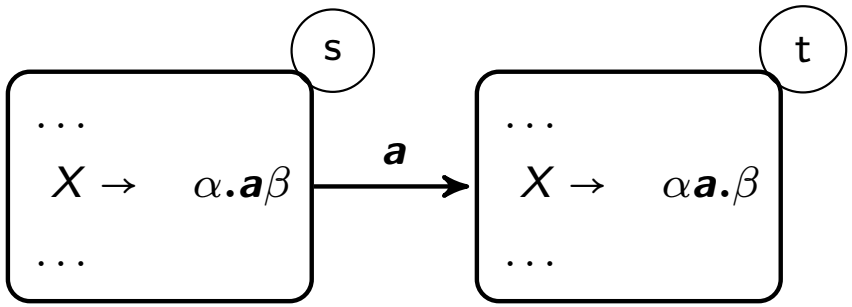
  $\Rightarrow$ crucial when determining *reaction*

State transition allowing a shift

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

$$X \to \alpha . a \beta$$

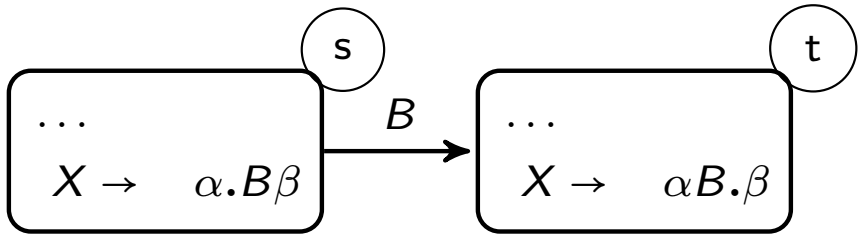
- construction thus has transition as follows



- 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

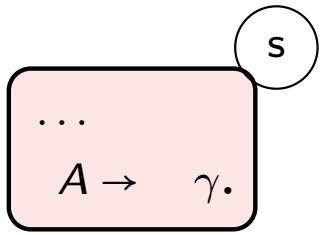
$X \to \alpha.B\beta$



- 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

$\Rightarrow$ reduce step

- builds up (implicitly) new parent node A in the bottom-up procedure
- Note: A on top of the stack instead of γ:[4]
  - new top state!
  - remember the "goto-transition" (shift of a non-terminal)

[4] 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.

Remarks: states, transitions, and reduce steps

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

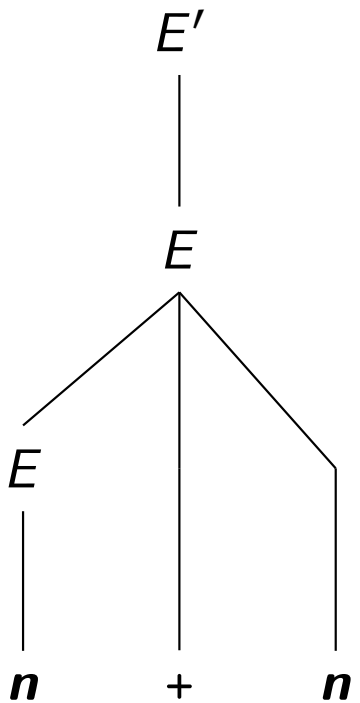
No edges to represent (all of) a reduce step!

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

- if the 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)

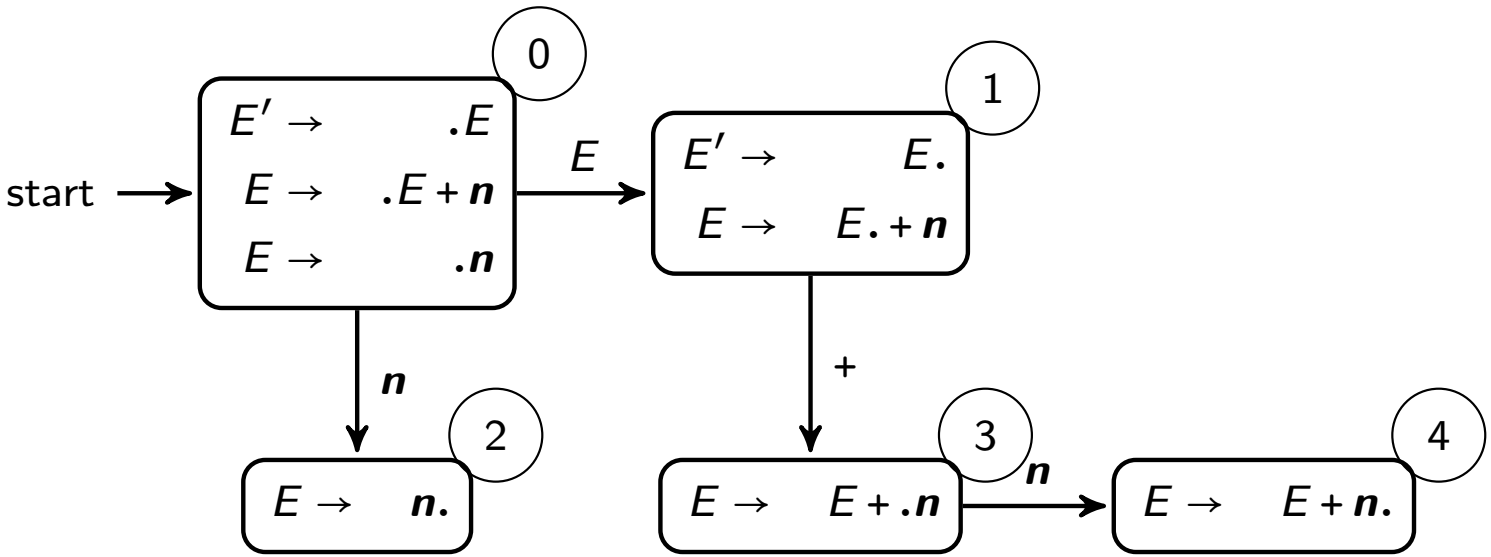
$$
\begin{array}{rcl}
E' & \rightarrow & E \\
E & \rightarrow & E + \boldsymbol{n} \mid \boldsymbol{n}
\end{array}
$$



	parse stack	input	action
1	$	$\boldsymbol{n}+\boldsymbol{n}\$$	shift
2	$\boldsymbol{n}$	$+\boldsymbol{n}\$$	red:. $E \rightarrow \boldsymbol{n}$
3	E	$+\boldsymbol{n}\$$	shift
4	$E+$	$\boldsymbol{n}\$$	shift
5	$E+\boldsymbol{n}$	$\$$	reduce $E \rightarrow E+\boldsymbol{n}$
6	E	$\$$	red.: $E' \rightarrow E$
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)

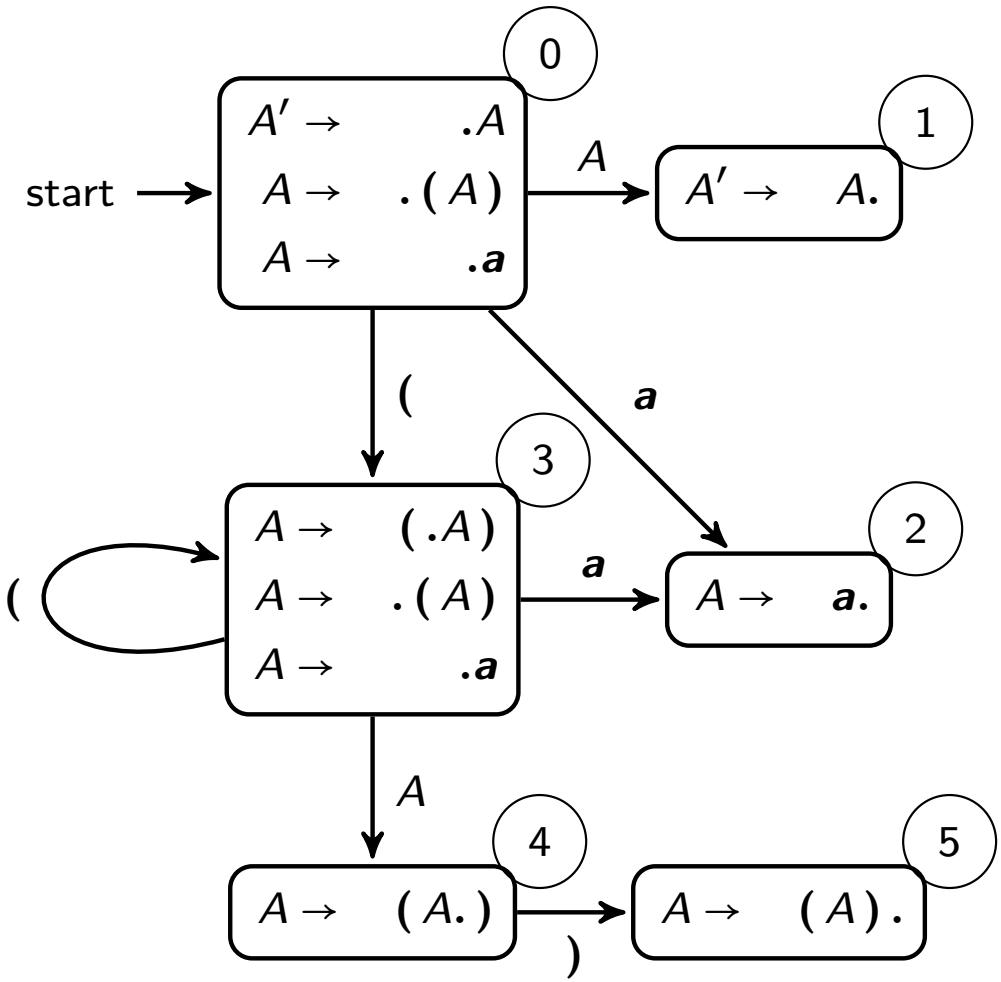
LR(0) grammar

The top-state alone determines the next step.

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

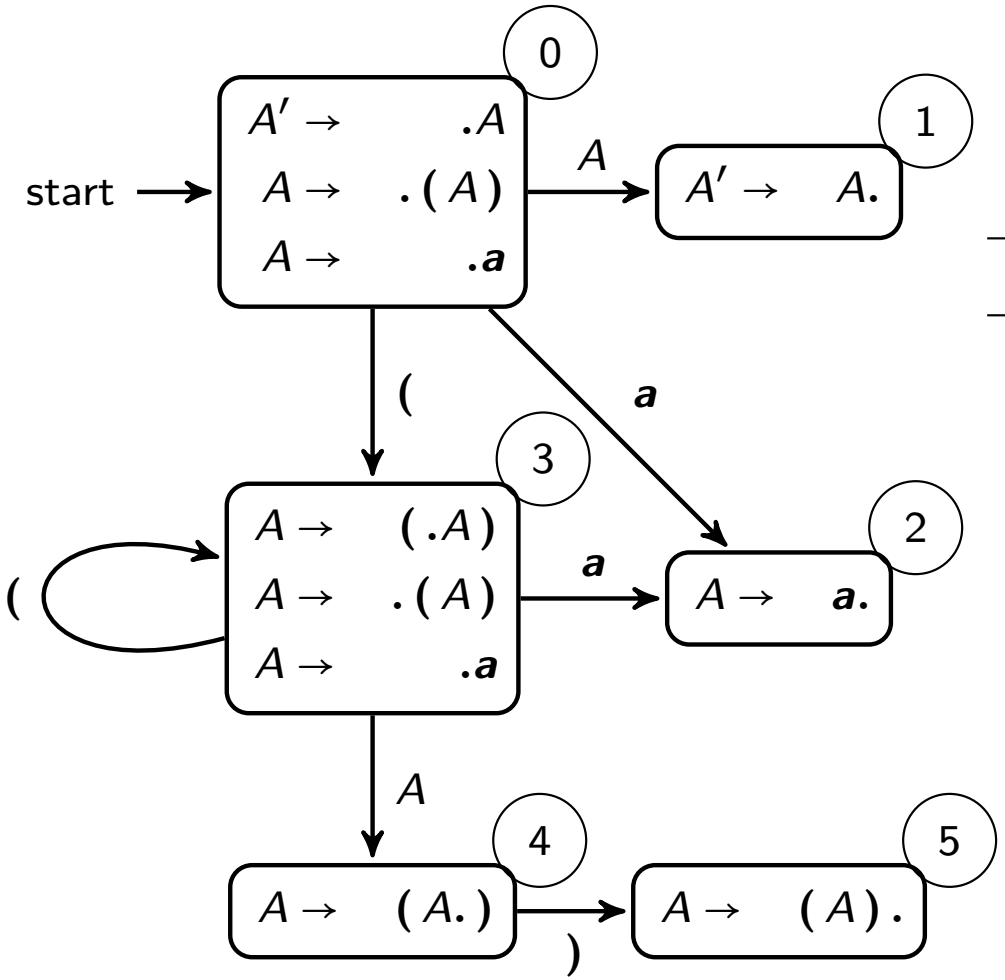
Simple parentheses

$$A \rightarrow (\,A\,) \mid a$$



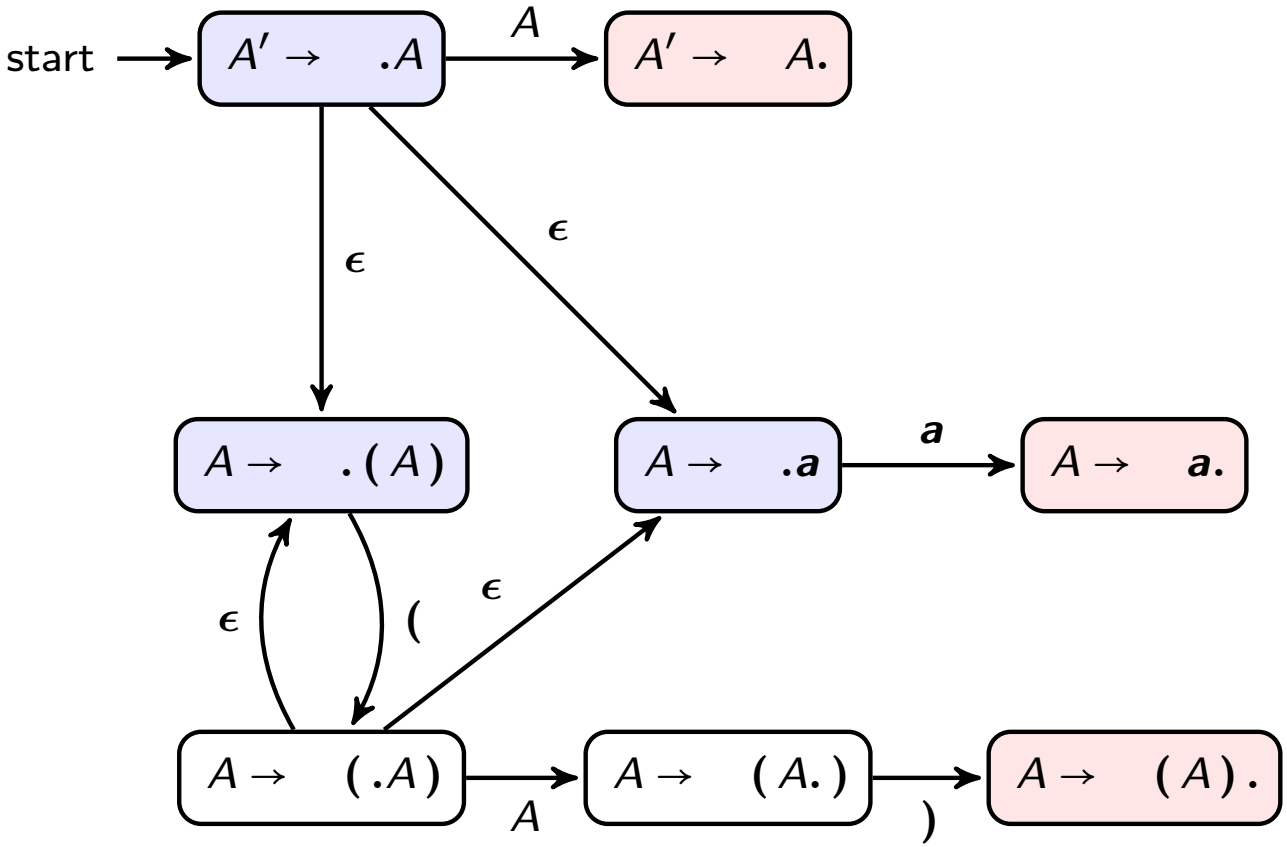
- for *shift*:
  - many shift transitions in state allowed
  - shift counts as *one* action (including "shifts" on non-terms)
- but for reduction: also the *production* must be clear

Simple parentheses is LR(0)



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

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 one non-terminal 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	$A \to a$				
3	shift		3			4
4	shift				5	
5	reduce	$A \to (\, A\,)$				

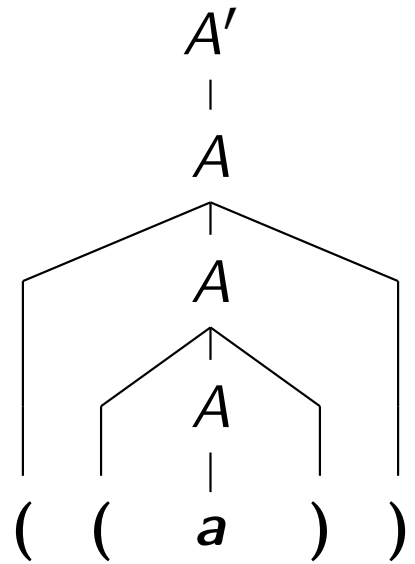
Parsing of $(\,(\,\boldsymbol{a}\,)\,)$

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

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

  $\Rightarrow$ accept

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	$\$_0$	$(\,(\,\boldsymbol{a}\,)\,\$$	shift
2	$\$_0(_3$	$(\,\boldsymbol{a}\,)\,\$$	shift
3	$\$_0(_3(_3$	$\boldsymbol{a}\,)\,\$$	shift
4	$\$_0(_3(_3\boldsymbol{a}_2$	$)\,\$$	reduce $A \to \boldsymbol{a}$
5	$\$_0(_3(_3A_4$	$)\,\$$	shift
6	$\$_0(_3(_3A_4)_5$	$\$$	reduce $A \to (\,A\,)$
7	$\$_0(_3A_4$	$\$$	????

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

Invariant

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

LR(0) parsing algo, given DFA

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

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

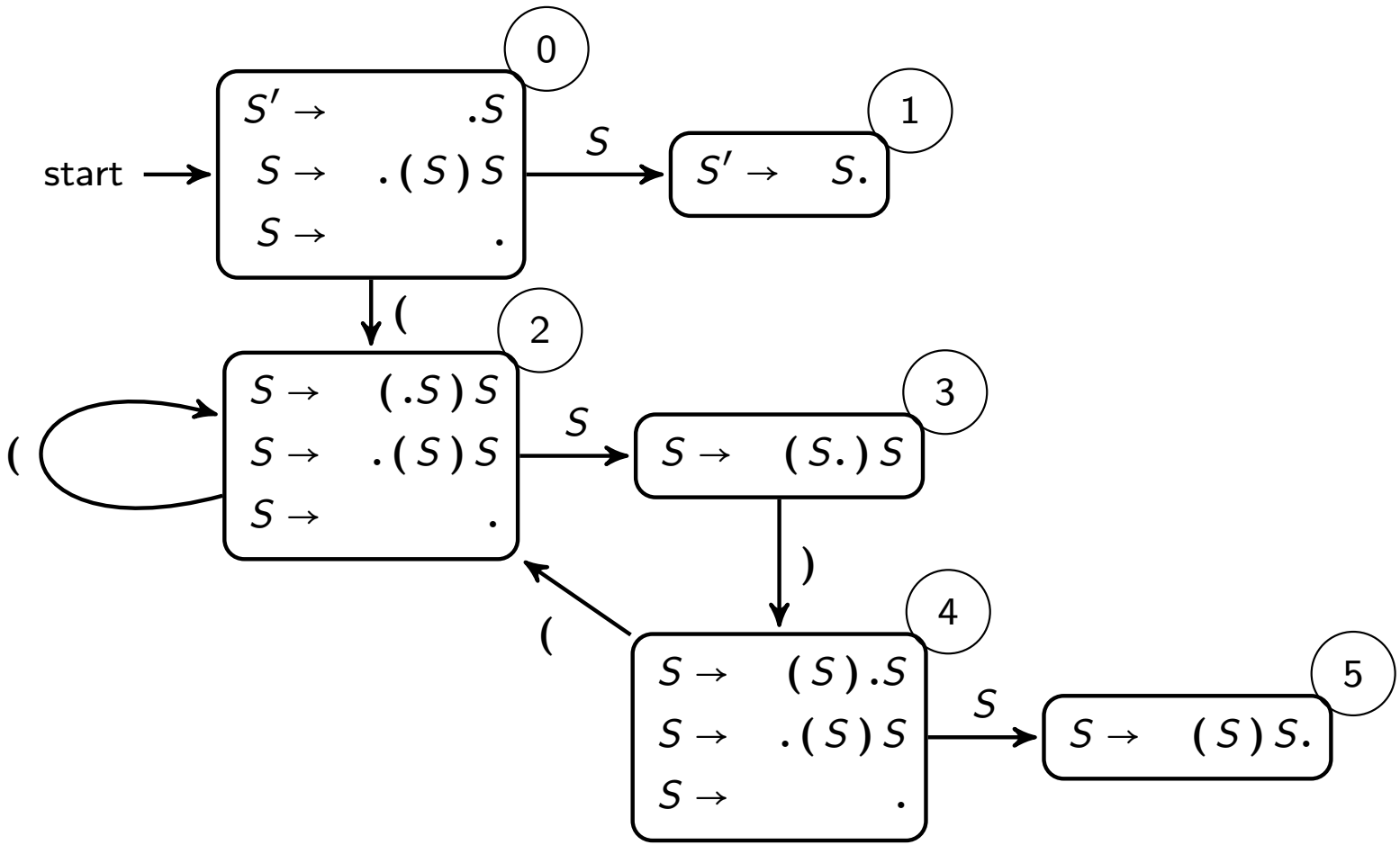
     pop: remove γ (including "its" states from the stack)
     back up: assume to be in state u which is *now* head state
     push: push A to the stack, new head state t where $u \xrightarrow{A} t$

LR(0) parsing algo remarks

- in [Louden, 1997]: 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)

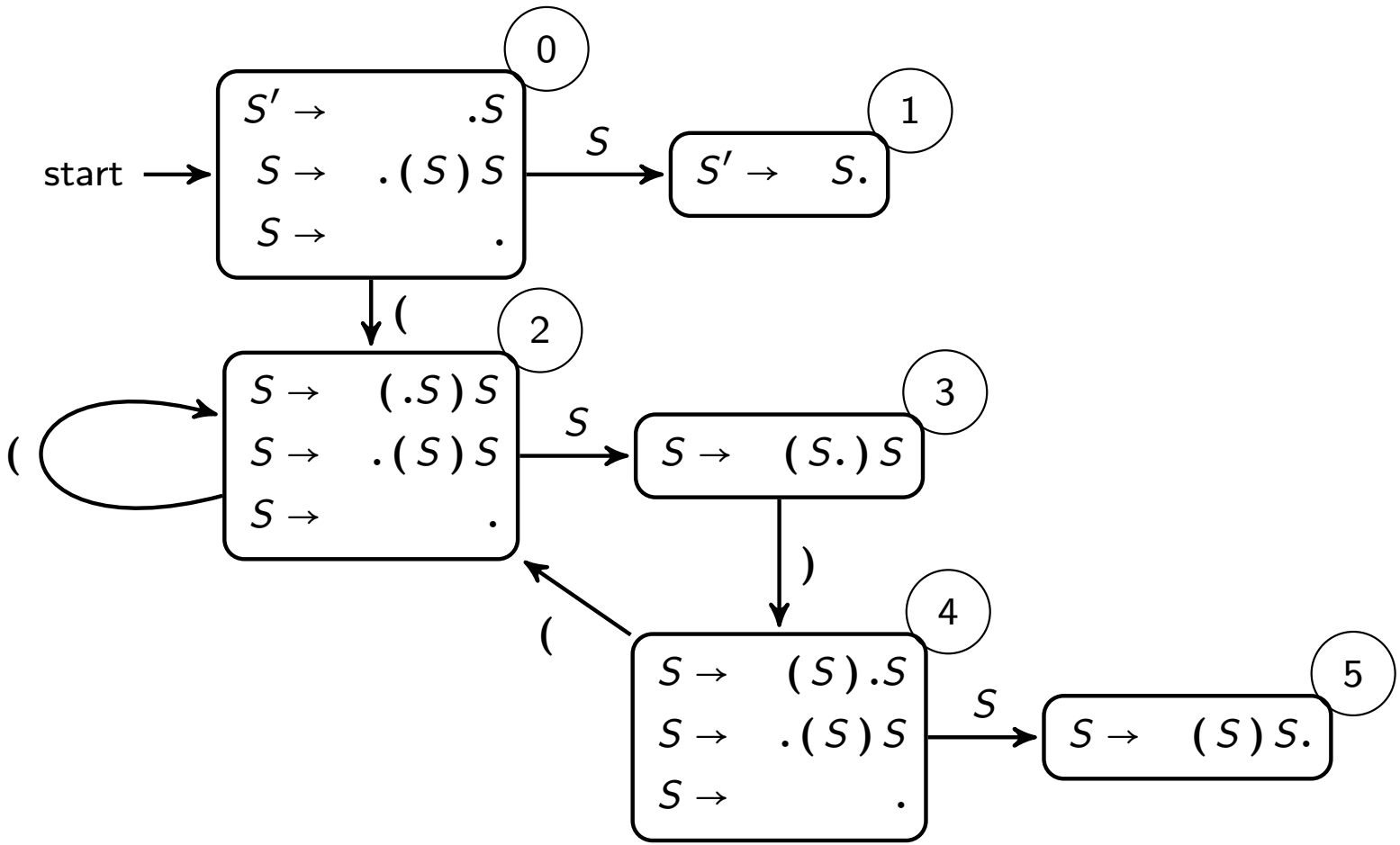
DFA parentheses again: LR(0)?

$$S' \rightarrow S$$
$$S \rightarrow (S)S \mid \epsilon$$



DFA parentheses again: LR(0)?

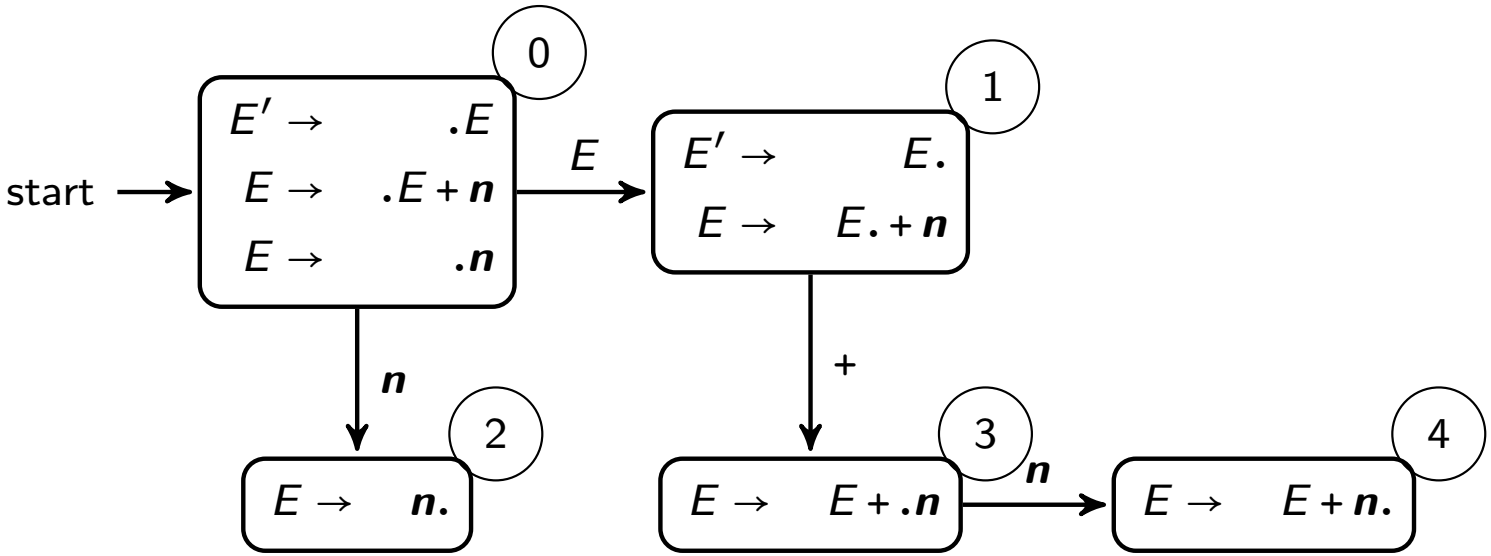
$$S' \rightarrow S$$
$$S \rightarrow (S)S \mid \epsilon$$



Look at states 0, 2, and 4

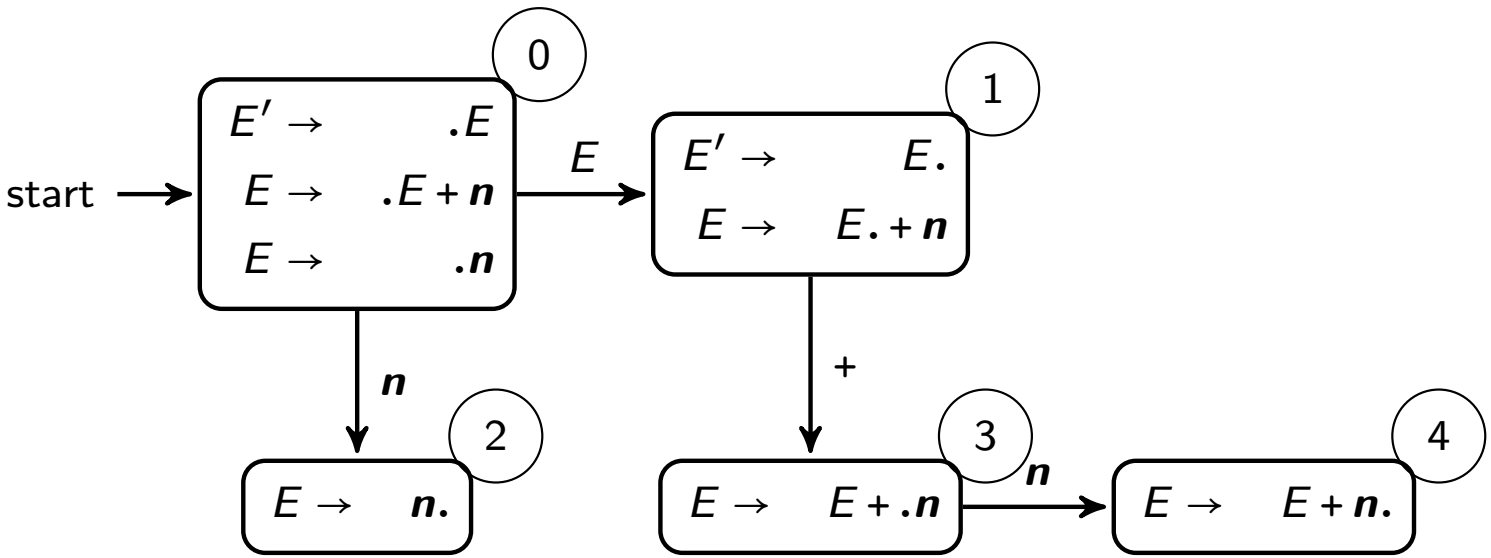
DFA addition again: LR(0)?

$$E' \rightarrow E$$
$$E \rightarrow E + \textbf{number} \mid \textbf{number}$$



DFA addition again: LR(0)?

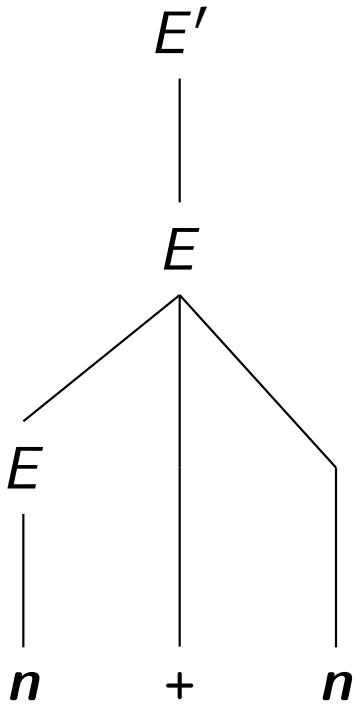
$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + \textbf{number} \mid \textbf{number}
\end{aligned}
$$



How to make a decision in state 1?

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

. . . especially the parts still to come



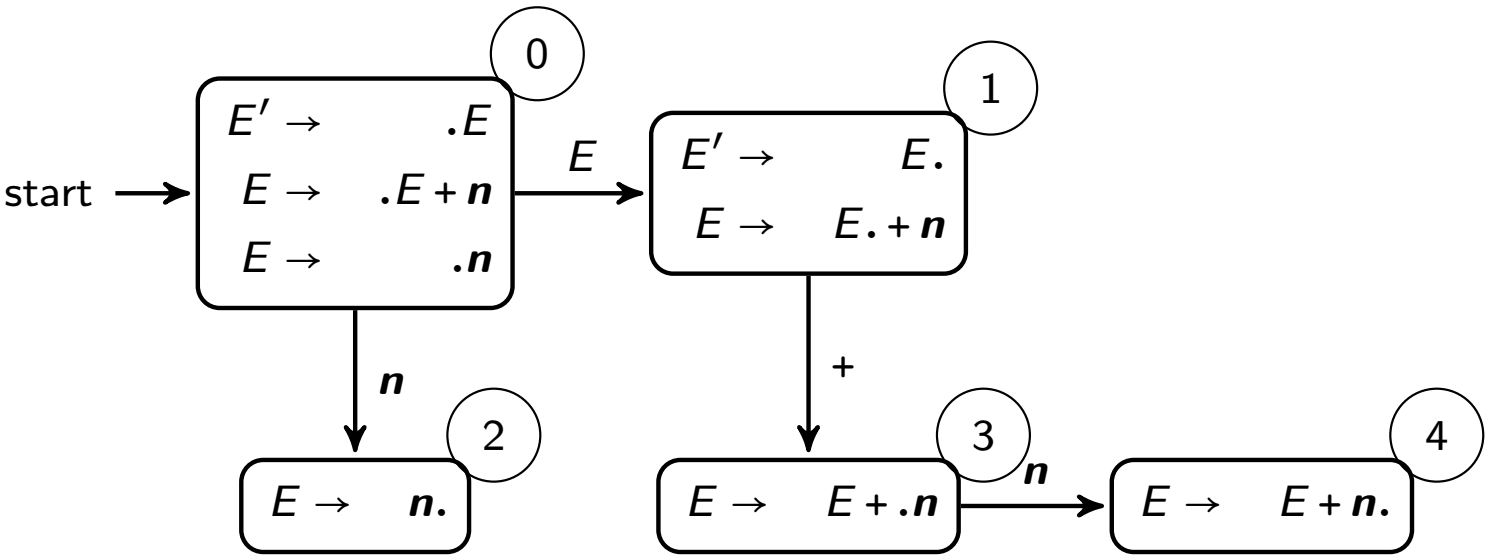
	parse stack	input	action
1	$	$\boldsymbol{n}+\boldsymbol{n}\$$	shift
2	$\$\boldsymbol{n}$	$+\boldsymbol{n}\$$	red:. $E \to \boldsymbol{n}$
3	$\$E$	$+\boldsymbol{n}\$$	shift
4	$\$E+$	$\boldsymbol{n}\$$	shift
5	$\$E+\boldsymbol{n}$	$	reduce $E \to E+\boldsymbol{n}$
6	$\$E$	$	red.: $E' \to E$
7	$\$E'$	$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

- current stack: representation of the 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 value of token)

Addition grammar (again)



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

$\Rightarrow$ look at the next input symbol (in the token variable)

One look-ahead

- LR(0), not very useful, much 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)-DFA's

Resolving LR(0) reduce/reduce conflicts

LR(0) reduce/reduce conflict:

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

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

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

$\Rightarrow$ 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) reduce/reduce conflicts

LR(0) reduce/reduce conflict:

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

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

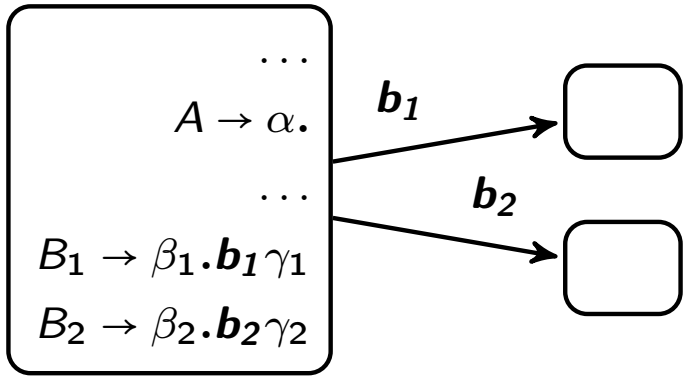
- If $\mathit{Follow}(A) \cap \mathit{Follow}(B) = \varnothing$

$\Rightarrow$ 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

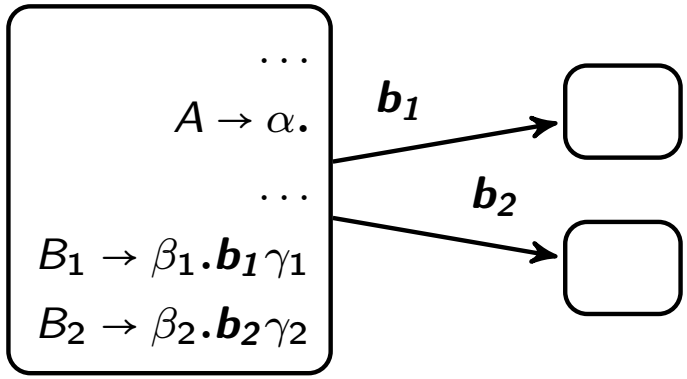
- If $\mathit{Follow}(A) \cap \{\boldsymbol{b_1}, \boldsymbol{b_2}, \ldots\} = \varnothing$

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

Resolving LR(0) shift/reduce conflicts

LR(0) shift/reduce conflict:



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

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

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

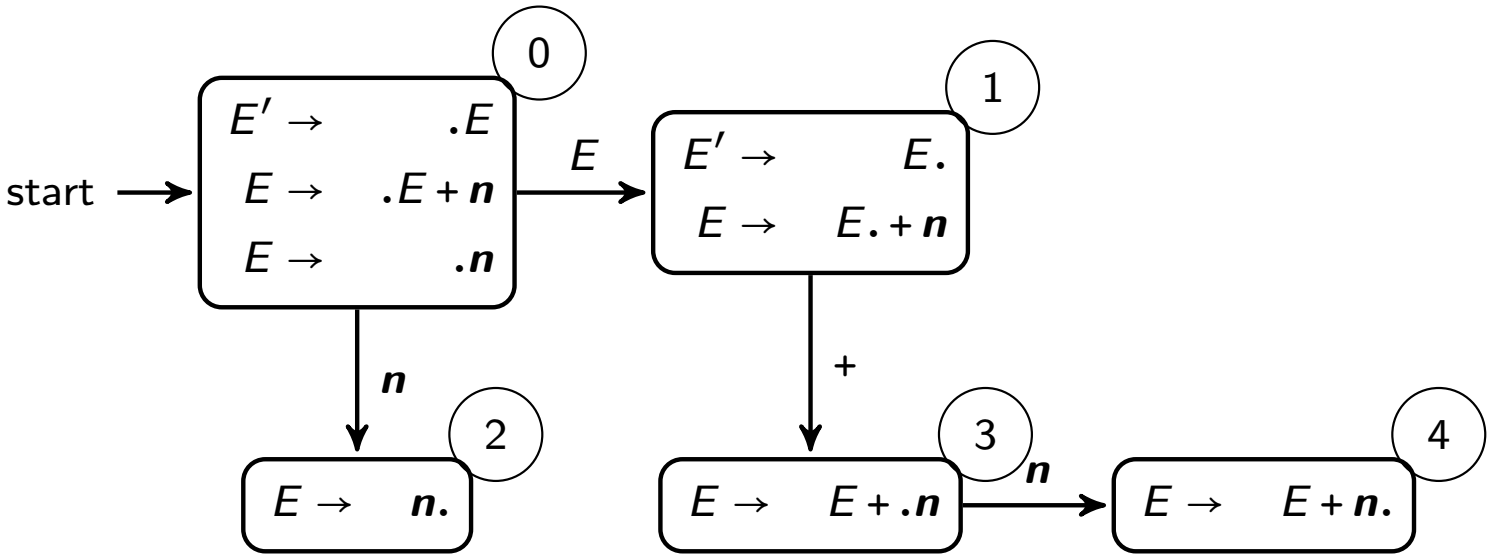
SLR(1) requirement on states (as in the book)

- 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

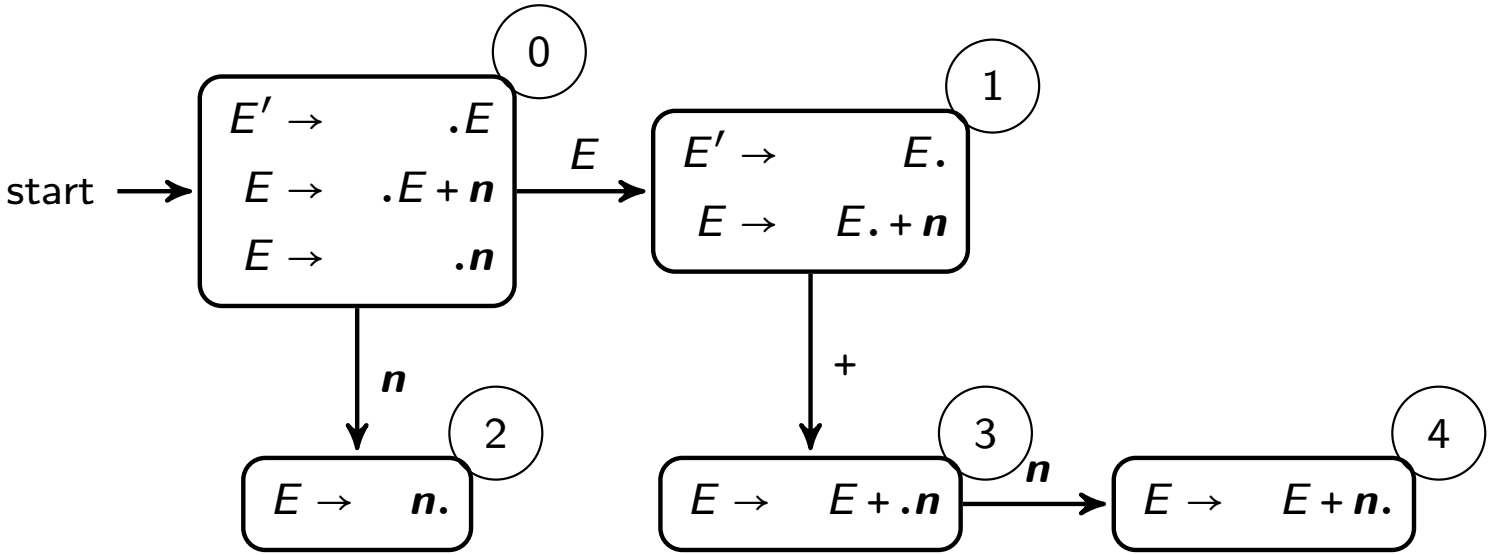
1. s contains $A \to \alpha.X\beta$, where X is a terminal and X is the next token on the input, then
   - shift X from input to top of stack. the new *state* pushed on the stack: state t where $s \xrightarrow{X} t$[5]
2. s contains a *complete* item (say $A \to \gamma.$) and the next token in the input is in $Follow(A)$: *reduce* by rule $A \to \gamma$:
   - A reduction by $S' \to S$: *accept*, if input is empty[6]
   - 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*

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

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

Parsing table for SLR(1)



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

for state 2 and 4: $n \notin Follow(E)$

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
- 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[7]

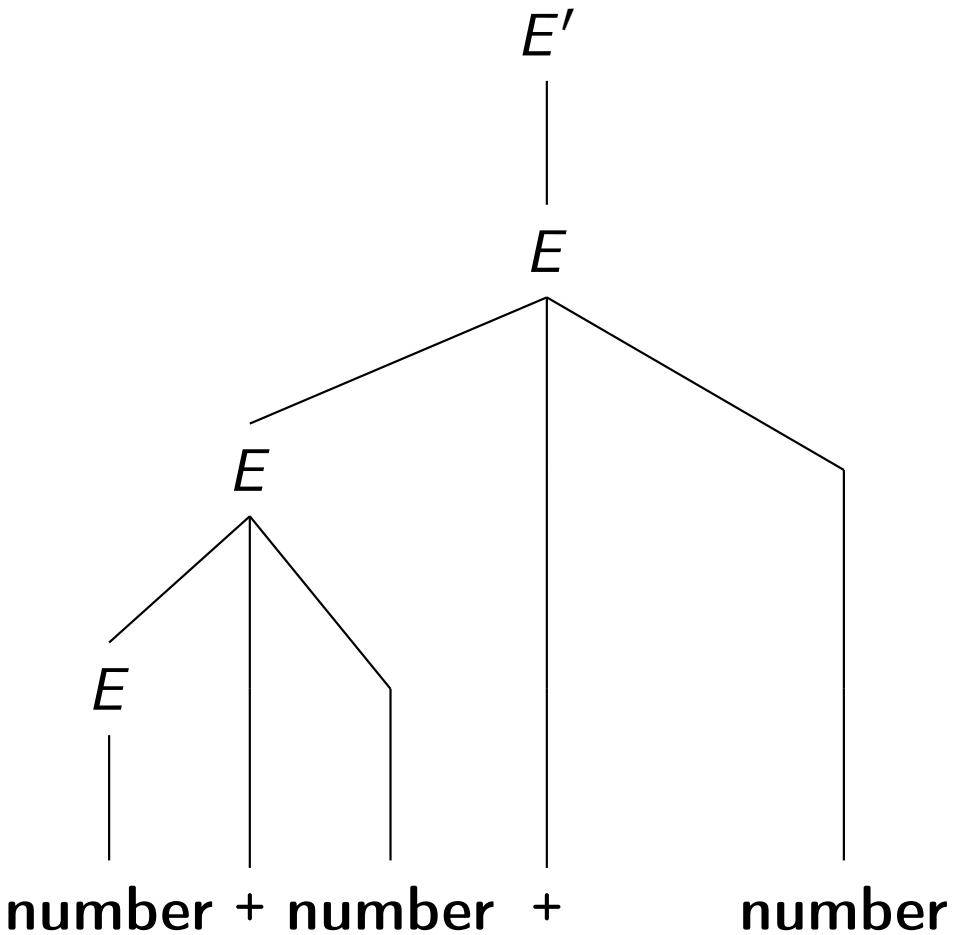
[7] by which it, strictly speaking, would no longer be an SRL(1)-table :-)

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

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

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

Corresponding parse tree



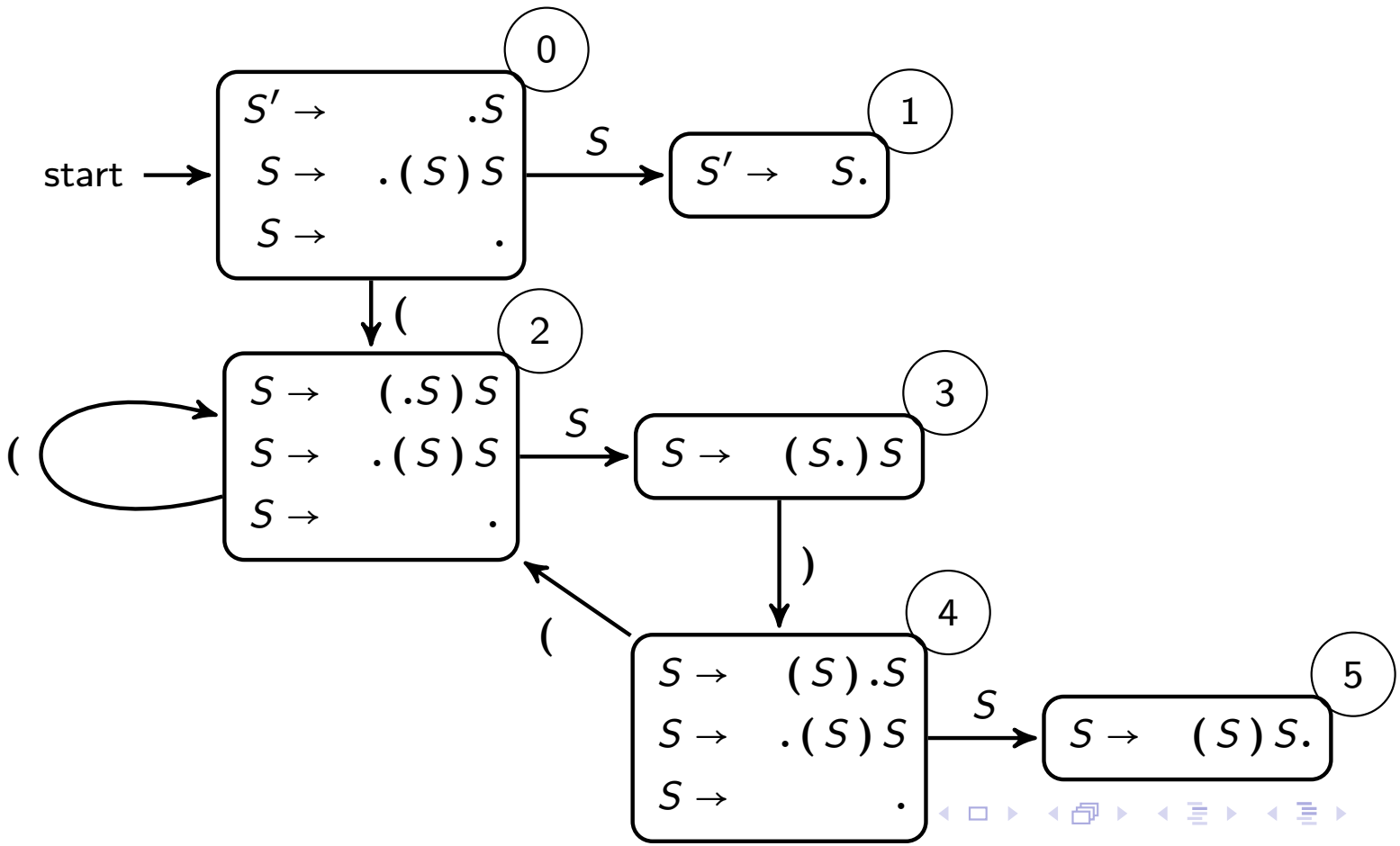
Revisit the parentheses again: SLR(1)?

Grammar: parentheses (from before)

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

Follow set

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



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	$\$_0 S_2$	$\$$	accept

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 $Xw \in \text{First}_k(X \beta)$ 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 $w \in \text{Follow}_k(A)$ 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*

- 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)
  - a grammar with many S/R-conflicts: go back to the drawing board

Example of an ambiguous grammar

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

In the following E for *exp* etc.

Simplified "schematic" if-then-else

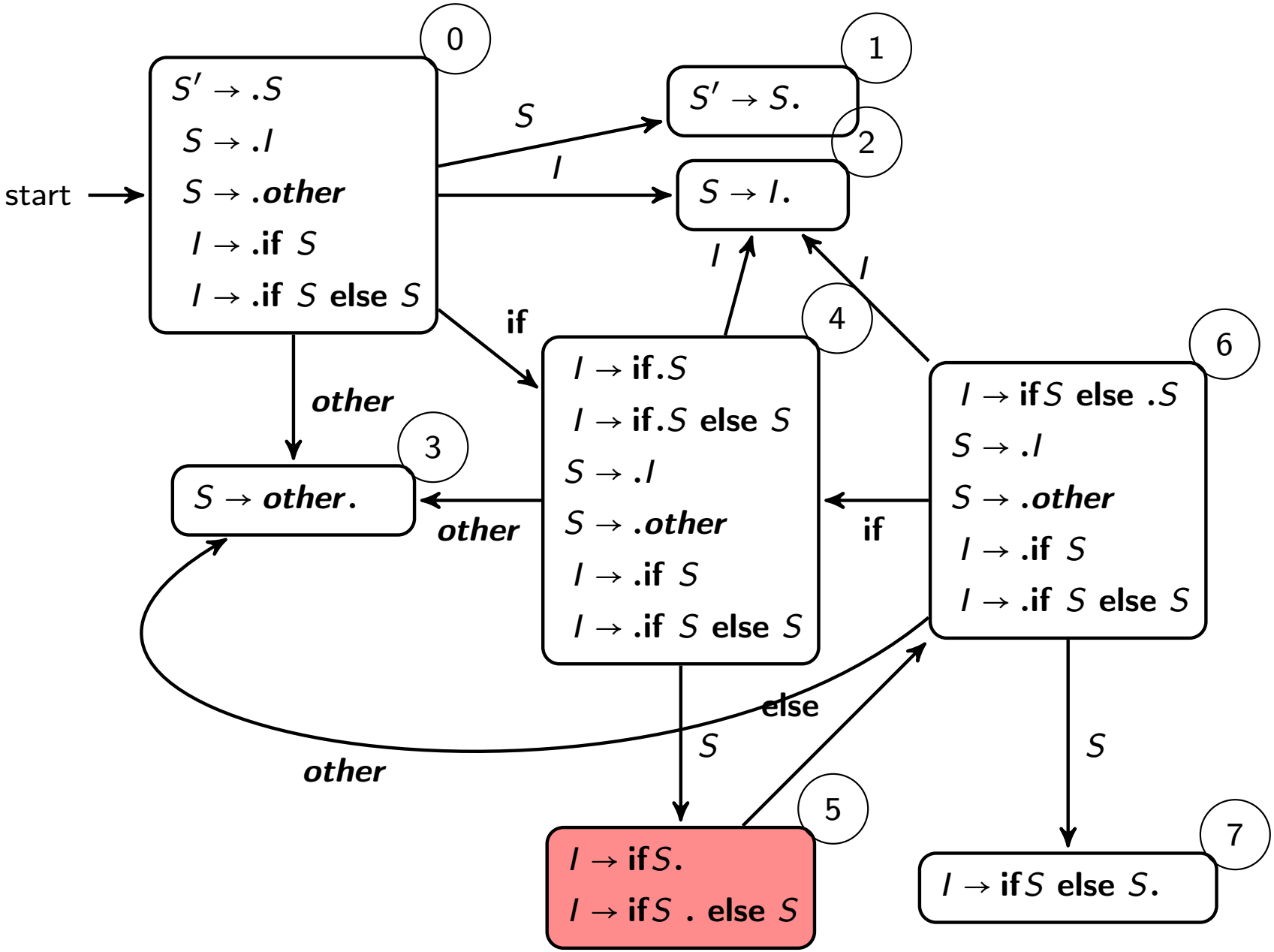
$$
\begin{array}{rcl}
S & \rightarrow & I \mid \textit{other} \\
I & \rightarrow & \textbf{if } S \mid \textbf{if } S \textbf{ else } S
\end{array}
$$

Follow-sets

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

- since ambiguous: at least one conflict must be somewhere

DFA of LR(0) items



Simple conditionals: parse table

Grammar

S	$\rightarrow$	I	(1)
	$\mid$	***other***	(2)
I	$\rightarrow$	**if** S	(3)
	$\mid$	**if** S **else** S	(4)

SLR(1)-parse-table, conflict resolved

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

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

Parser run (= reduction)

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	$\$_0\mathbf{if}_4$	**if *other* else *other*$**	shift: 4
3	$\$_0\mathbf{if}_4\mathbf{if}_4$	***other* else *other*$**	shift: 3
4	$\$_0\mathbf{if}_4\mathbf{if}_4\mathit{other}_3$	**else *other*$**	reduce: 2
5	$\$_0\mathbf{if}_4\mathbf{if}_4 S_5$	**else *other*$**	shift 6
6	$\$_0\mathbf{if}_4\mathbf{if}_4 S_5\mathbf{else}_6$	***other*$**	shift: 3
7	$\$_0\mathbf{if}_4\mathbf{if}_4 S_5\mathbf{else}_6\mathit{other}_3$	**$**	reduce: 2
8	$\$_0\mathbf{if}_4\mathbf{if}_4 S_5\mathbf{else}_6 S_7$	**$**	reduce: 4
9	$\$_0\mathbf{if}_4 I_2$	**$**	reduce: 1
10	$\$_0 S_1$	**$**	accept

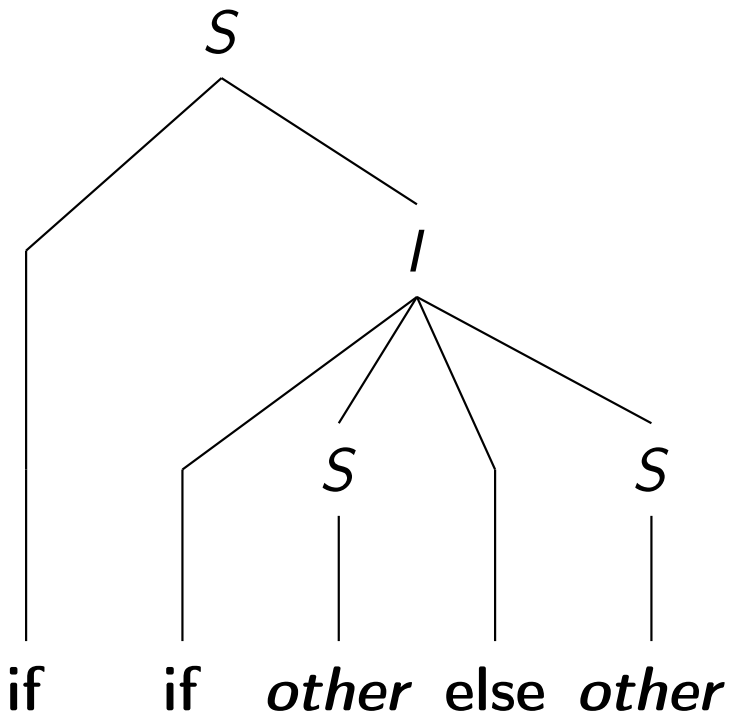
Parser run, different choice

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		

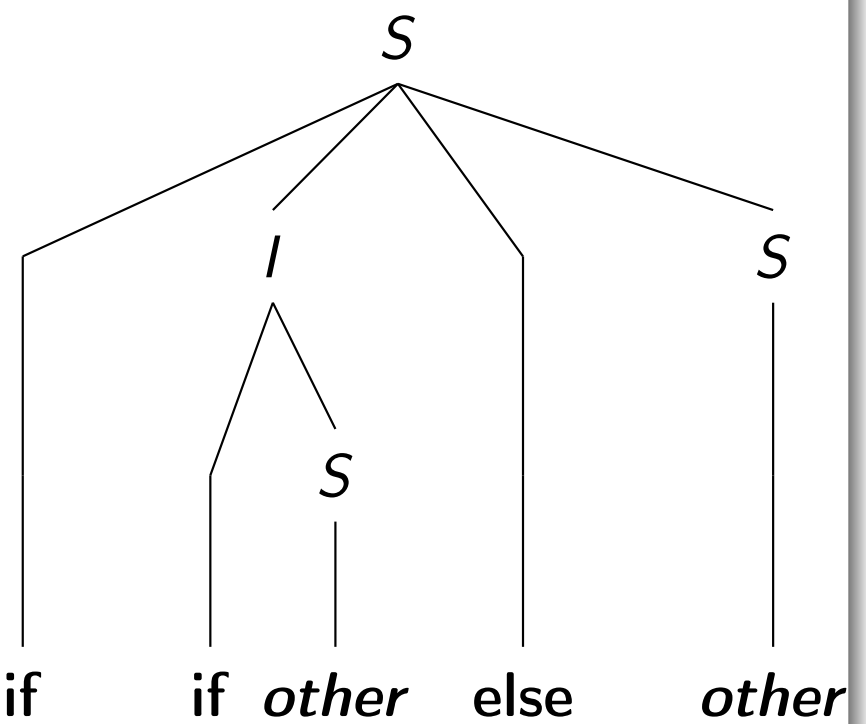
stage	parsing stack	input	action
1	$\$_0$	**if if *other* else *other*$**	shift: 4
2	$\$_0\mathbf{if}_4$	**if *other* else *other*$**	shift: 4
3	$\$_0\mathbf{if}_4\mathbf{if}_4$	***other* else *other*$**	shift: 3
4	$\$_0\mathbf{if}_4\mathbf{if}_4\boldsymbol{other}_3$	**else *other*$**	reduce: 2
5	$\$_0\mathbf{if}_4\mathbf{if}_4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

Parse trees: simple conditions

shift-precedence: conventional



"wrong" tree



"dangling else"

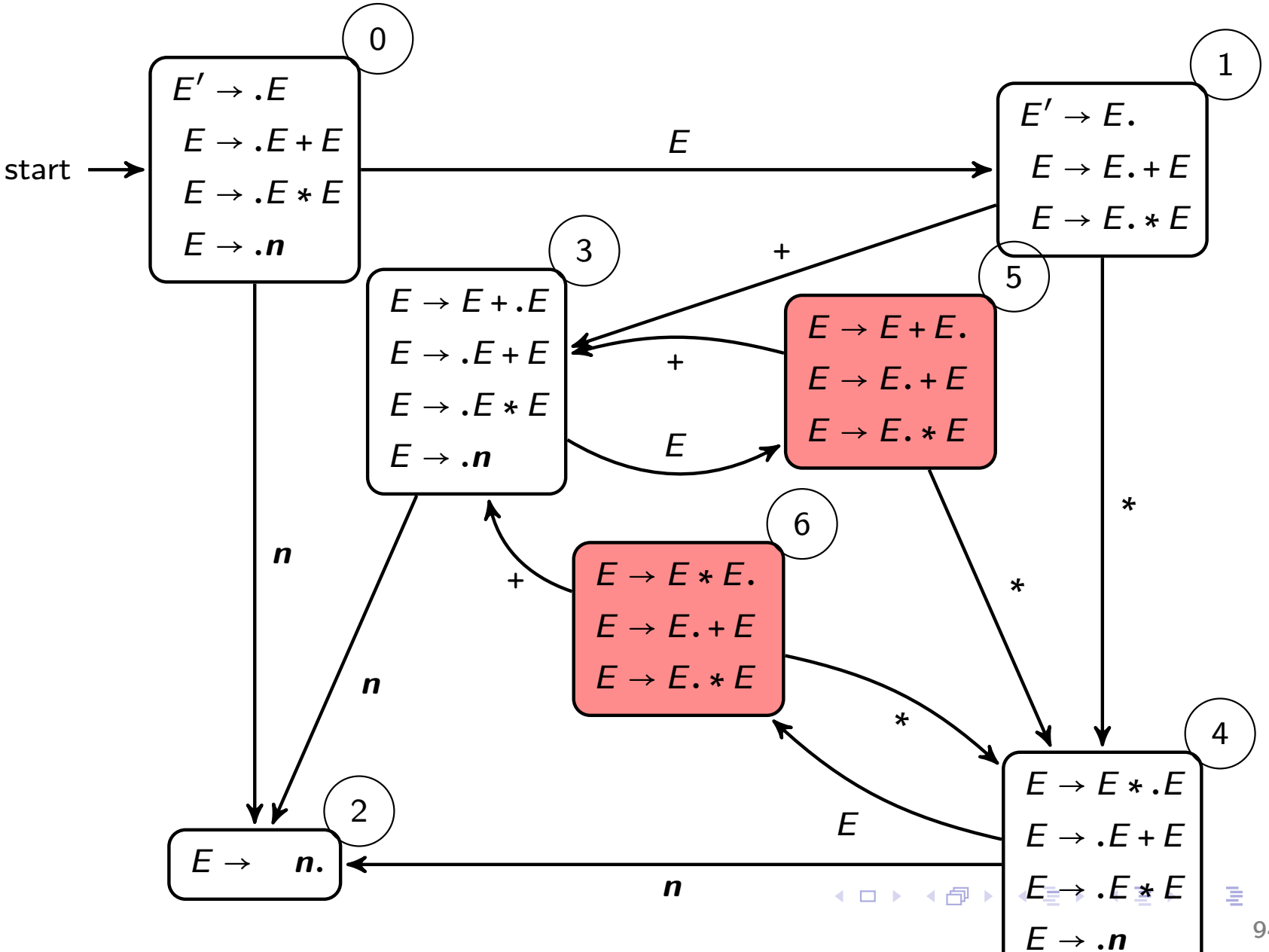
"an **else** belongs to the last previous, still open (= dangling) if-clause"

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` ...

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

DFA for + and ×


start
0
$E' \to .E$
$E \to .E + E$
$E \to .E * E$
$E \to .n$
1
$E' \to E.$
$E \to E. + E$
$E \to E. * E$
2
$E \to n.$
3
$E \to E + .E$
$E \to .E + E$
$E \to .E * E$
$E \to .n$
4
$E \to E * .E$
$E \to .E + E$
$E \to .E * E$
$E \to .n$
5
$E \to E + E.$
$E \to E. + E$
$E \to E. * E$
6
$E \to E * E.$
$E \to E. + E$
$E \to E. * E$
E
n
+
*

States with conflicts

- state 5
  - stack contains $\ldots E + E$
  - 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 $\ldots E * E$
  - 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 ×

state	input				goto
	n	+	*	$	E
0	$s:2$				1
1		$s:3$	$s:4$	accept	
2		$r: E \to \boldsymbol{n}$	$r: E \to \boldsymbol{n}$	$r: E \to \boldsymbol{n}$	
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 ↑ or $**$)?

Defined as *right-associative*. See exercise

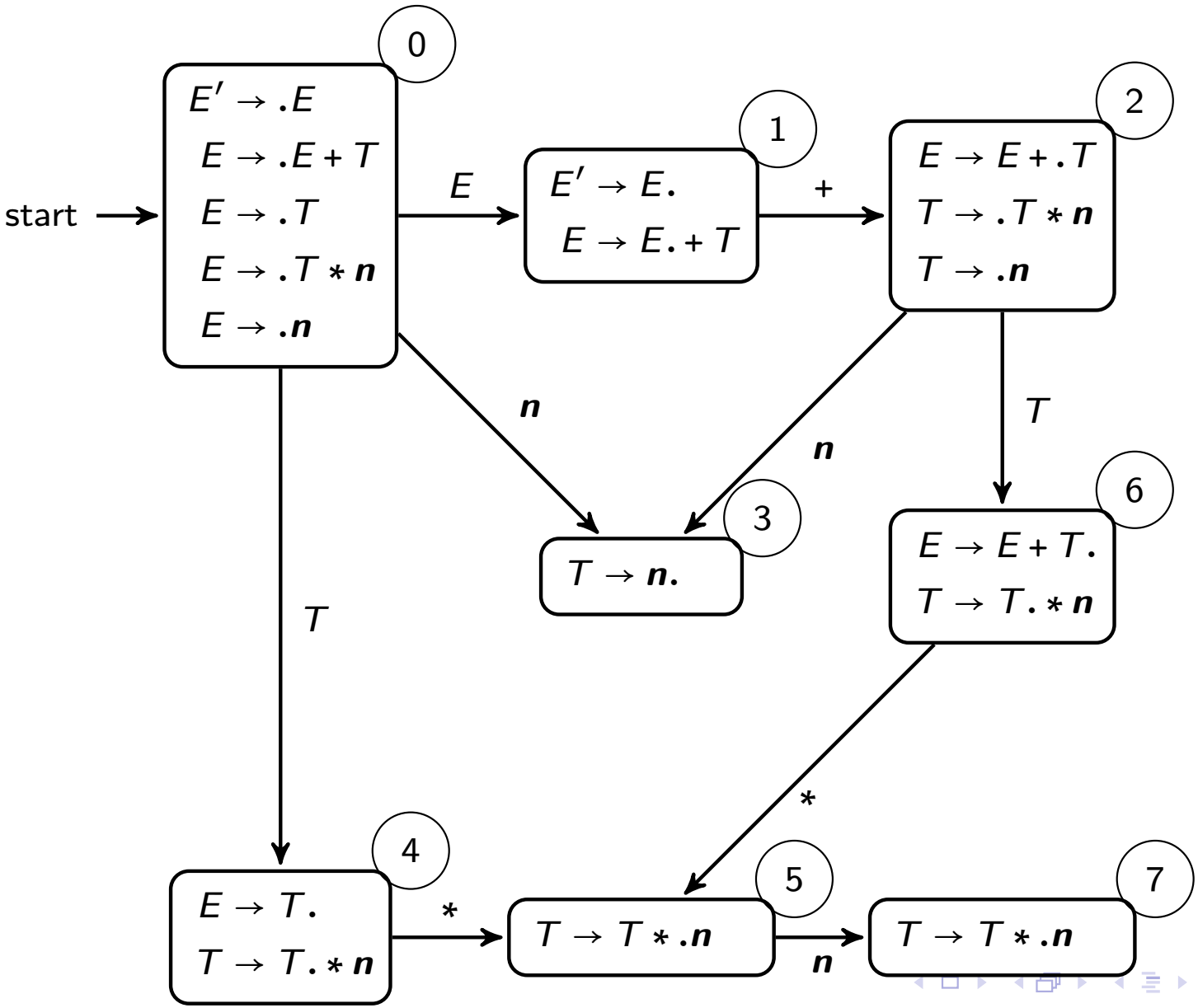
For comparison: unambiguous grammar for + and *

Unambiguous grammar: precedence and left-assoc built in

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

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

DFA for unambiguous + and ×



DFA remarks

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

    state 1: $Follow(E') = \{\$\}$
    state 4: $Follow(E) = \{\$, +\}$
    state 6: $Follow(E) = \{\$, +\}$
    state 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)

Assignment grammar fragment[a]

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

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{call-stmt} \mid \textit{assign-stmt} \\
\textit{call-stmt} & \rightarrow & \textbf{identifier} \\
\textit{assign-stmt} & \rightarrow & \textit{var} \textbf{:=} \textit{exp} \\
\textit{var} & \rightarrow & [\,\textit{exp}\,] \mid \textbf{identifier} \\
\textit{exp} & \mid & \textit{var} \mid \textbf{number}
\end{array}
$$

Assignment grammar fragment, simplified

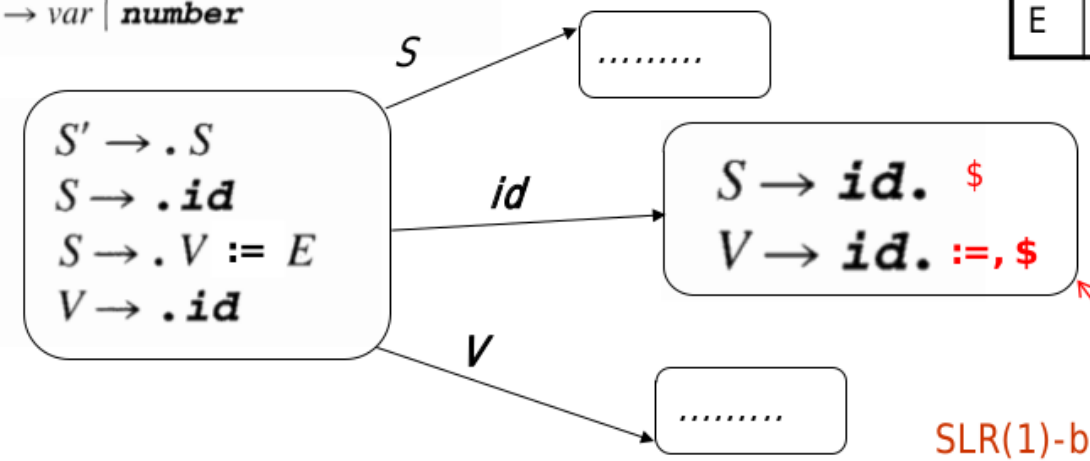
$$
\begin{array}{rcl}
S & \rightarrow & \textbf{\textit{id}} \mid V \textbf{:=} E \\
V & \rightarrow & \textbf{\textit{id}} \\
E & \rightarrow & V \mid \textbf{n}
\end{array}
$$

non-SLR(1): Reduce/reduce conflict

stmt → *call-stmt* | *assign-stmt*
call-stmt → **identifier**
assign-stmt → *var* **:=** *exp*
var → *var* **[** *exp* **]** | **identifier**
exp → *var* | **number**

$S \rightarrow \mathbf{id} \mid V \mathbf{:=} E$
$V \rightarrow \mathbf{id}$
$E \rightarrow V \mid \mathbf{n}$

	First	Follow
S	id	$
V	id	:=, $
E	id, n	$



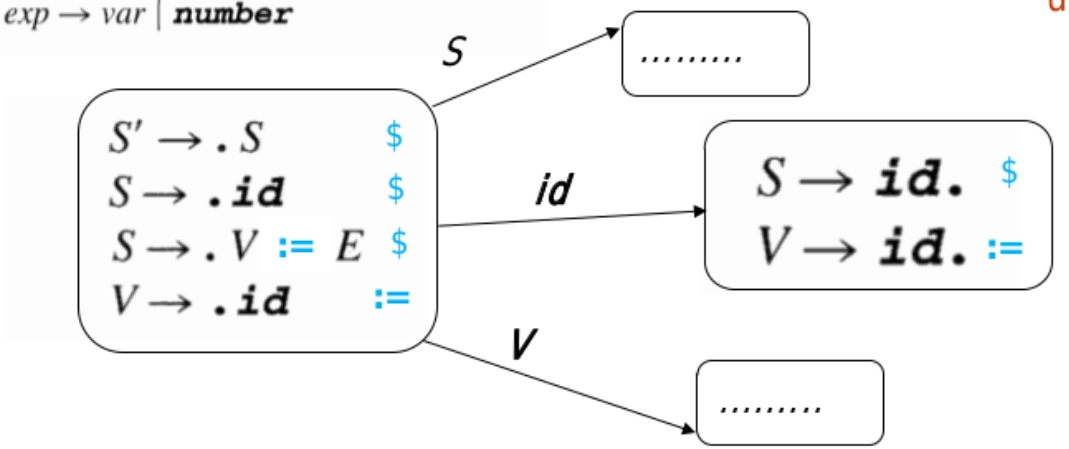
SLR(1)-betrakning: Gir her reduser/reduser-konflikt for input = $. Se First og Follow over.

Situation can be saved

stmt → *call-stmt* | *assign-stmt*
call-stmt → **identifier**
assign-stmt → *var* **:=** *exp*
var → *var* **[** *exp* **]** | **identifier**
exp → *var* | **number**

$S \rightarrow \mathbf{id} \mid V \mathbf{:=} E$
$V \rightarrow \mathbf{id}$
$E \rightarrow V \mid \mathbf{n}$

Altså, for SLR(1): Gir her reduser/reduser-konflikt for input = $. Se First og Follow under.



	First	Follow
S	id	$
V	id	:=, $
E	id, n	$

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

Adding precision in the states of the automaton already

Instead of using LR(0)-items and, when the LR(0) DFA is done, try to disambiguate with the help of the follow sets for states containing complete items: make more fineg rained items:

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

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

LR(1) items

- main idea: simply make the look-ahead part of the item
- obviously: proliferation of states[9]

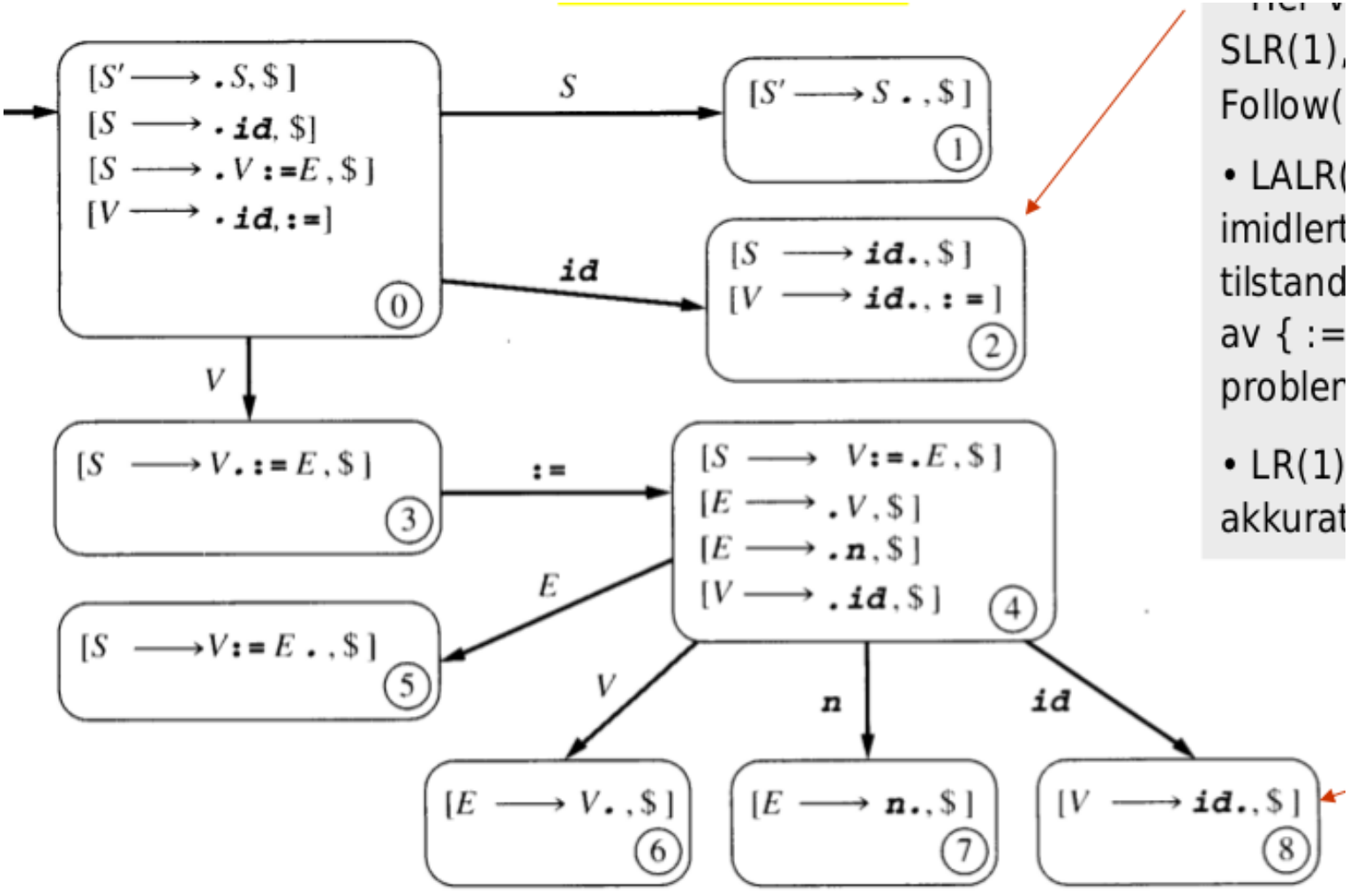
LR(1) items

$$[A \to \alpha.\beta, \boldsymbol{a}] \tag{2}$$

- $\boldsymbol{a}$: terminal/token, including \$

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

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



Remarks on the DFA

- Cf. state *2* (seen before)
  - in SLR(1): problematic (reduce/reduce), as $\mathit{Follow}(V) = \{\texttt{:=}, \$\}$
  - 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

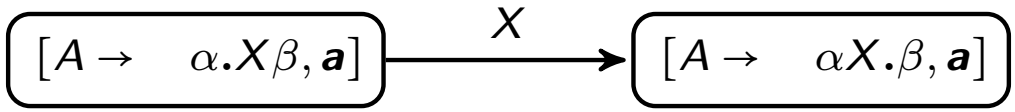
LALR(1)

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

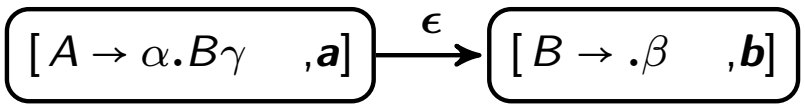


LR(1) transitions: ϵ

ϵ-transition

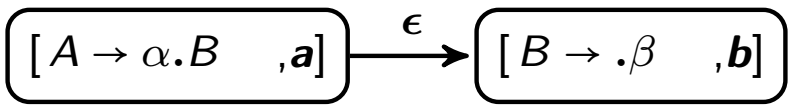
for all

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



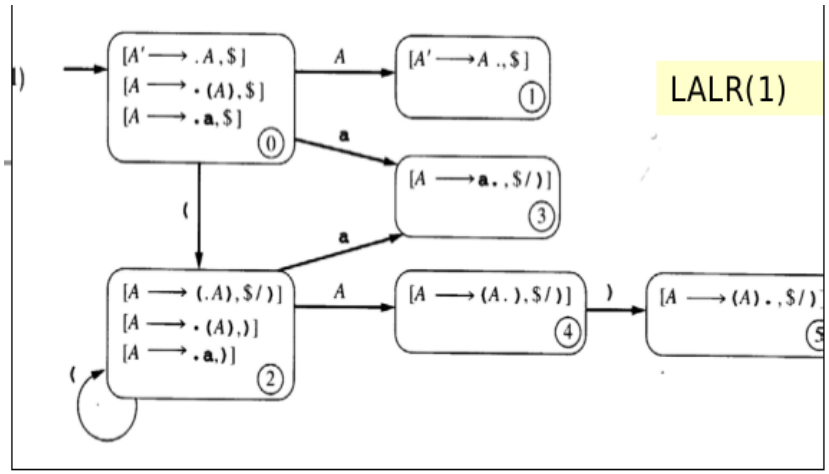
Special case ($\gamma = \epsilon$)

for all $B \to \beta_1 \mid \beta_2 \ldots$

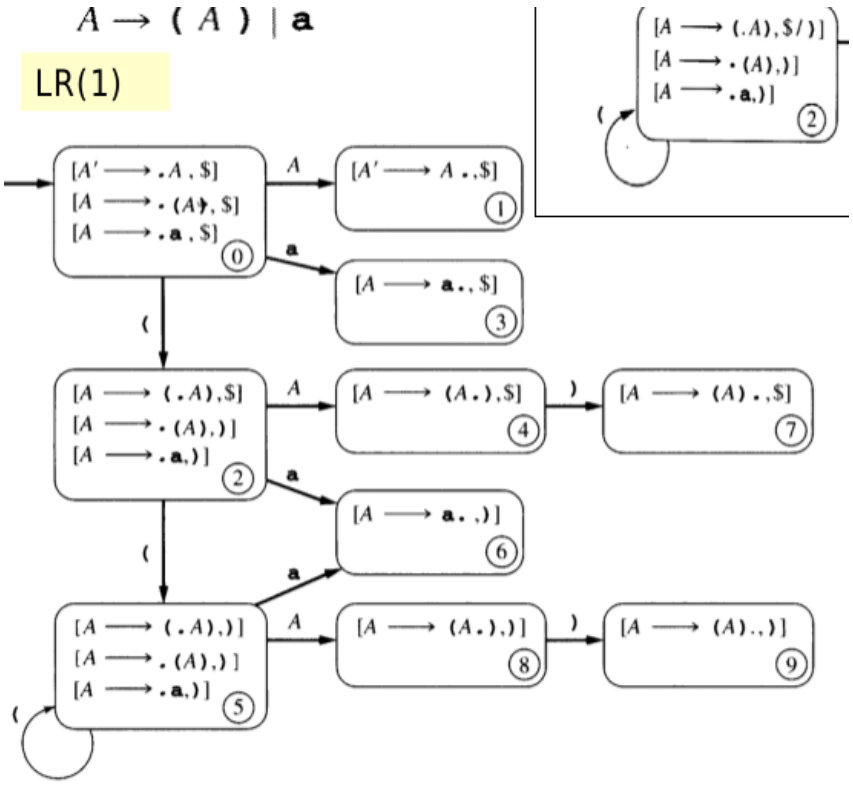


LALR(1) vs LR(1)

LALR(1)



LR(1)



Core of LR(1)-states

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

Core of an LR(1) state

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

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

LALR(1)-DFA by as collapse

- collapse all states with the same core
- based on above observations: edges are also consistent
- Result: almost like a LR(0)-DFA but additionally
  - still each individual item has still look ahead attached: the union of the "collapsed" items
  - especially for states with *complete* items $[A \to \alpha, \boldsymbol{a}, \boldsymbol{b}, \ldots]$ 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[10]
  - 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)

[10] 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.

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

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

Minimal requirement

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

- 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:
  - try to avoid error messages t hat only occur because of an already report ed 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`.

Error recovery in bottom-up parsing

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

$\Rightarrow$ 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	$\$_0 a_1 b_2 c_3 (_4 d_5 e_6$	$f\ )\ g\ h \ldots \$$	no entry for f

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

Possible error situation

	parse stack	input	action
1	$\$_0 a_1 b_2 c_3 (_4 d_5 e_6$	$f\ )\ g\ h \ldots \$$	no entry for f
2	$\$_0 a_1 b_2 c_3 B_v$	$g\ h \ldots \$$	back to normal
3	$\$_0 a_1 b_2 c_3 B_v g_7$	$h \ldots \$$	...

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

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	$\$_0 a_1 b_2 c_3 (_4 d_5 e_6$	$f\ )\ g\ h \ldots \$$	no entry for f

- 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)

Example again

	parse stack	input	action
1	$\$_0 a_1 b_2 c_3 (_4 d_5 e_6$	$f\ )\ g\ h \ldots \$$	no entry for f
2	$\$_0 a_1 b_2 c_3 B_v$	$g\ h \ldots \$$	back to normal
3	$\$_0 a_1 b_2 c_3 B_v g_7$	$h \ldots \$$	...

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

Panic mode may loop forever

	parse stack	input	action
1	$\$_0$	**(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

$$
\begin{aligned}
\textit{command} &\rightarrow \textit{exp} \\
\textit{exp} &\rightarrow \textit{term} * \textit{factor} \mid \textit{factor} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \mid \textit{factor} \\
\textit{factor} &\rightarrow \textbf{number} \mid (\ \textit{exp}\ )
\end{aligned}
$$

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).

Outline

References I

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