

Chapter 4

Parsing: Capsule Summary (tbc.)

Course “Compiler Construction”
Martin Steffen
Spring 2022

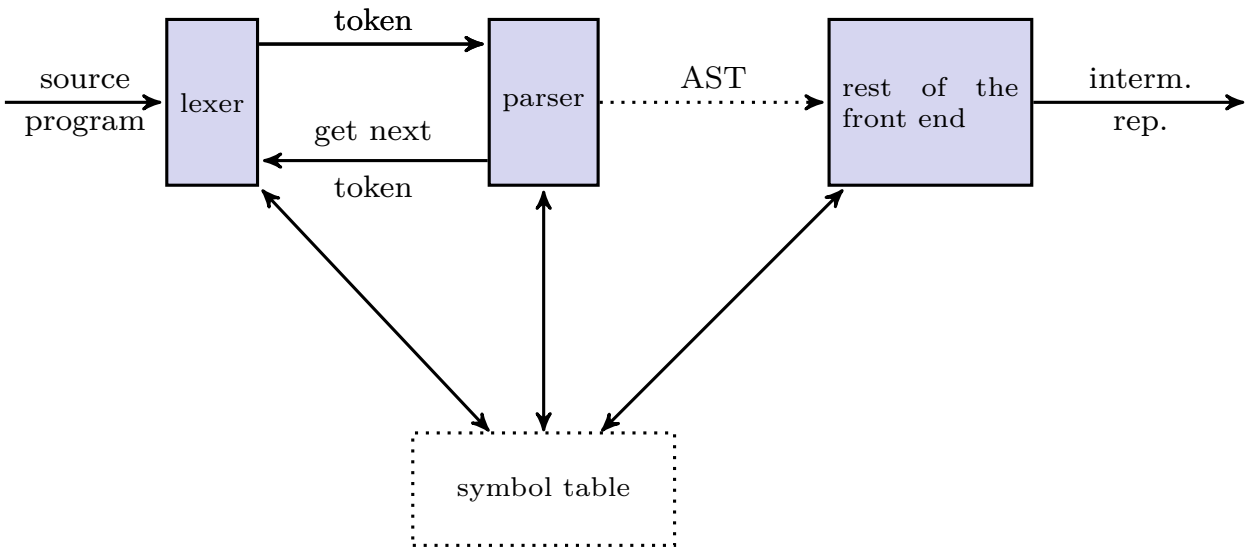


Section

Intro

Chapter 4 “Parsing: Capsule Summary (tbc.)”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Overview



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)

Parsing restricted classes of CFGs

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



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

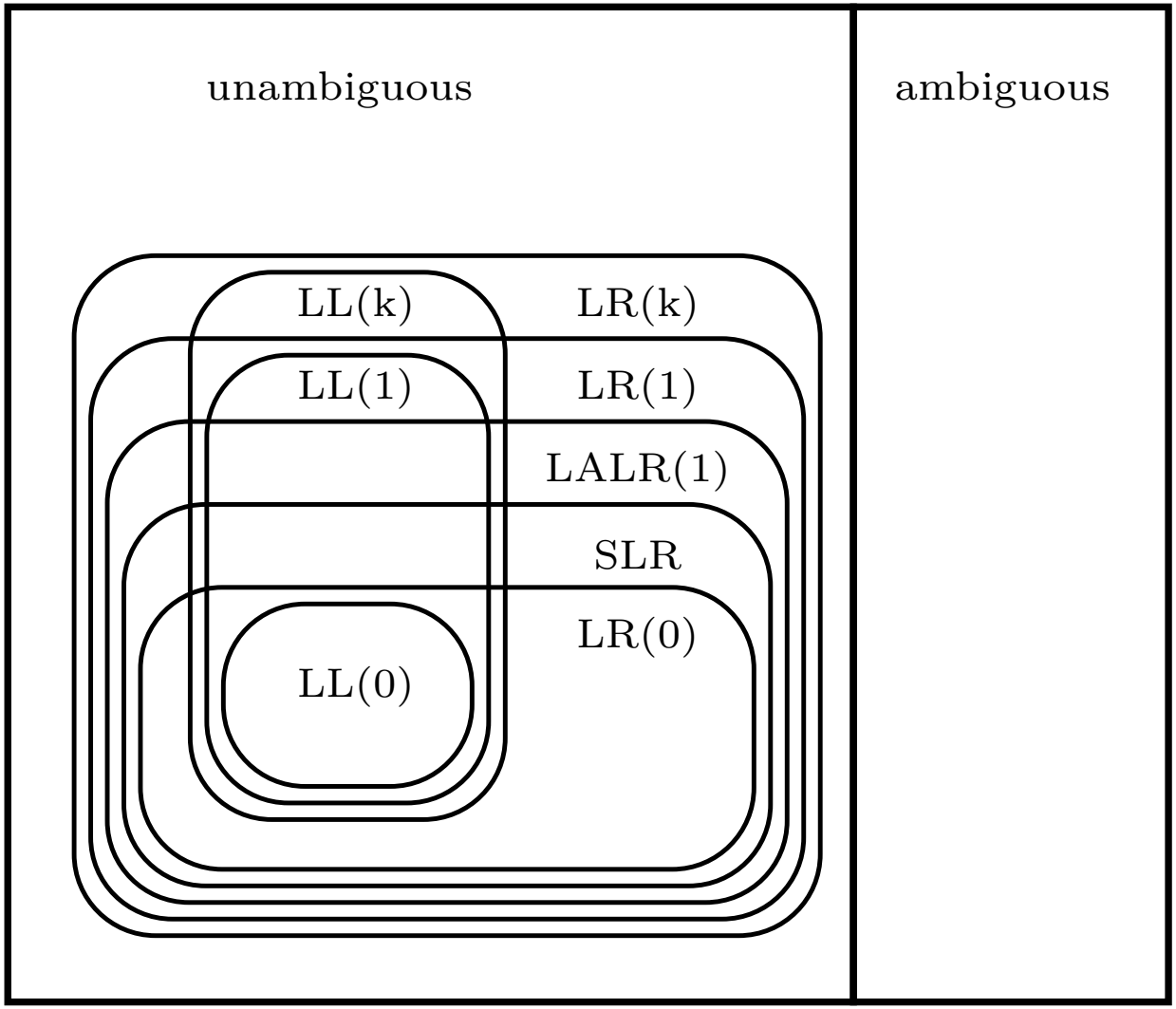
Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)

Hierarchy





Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)

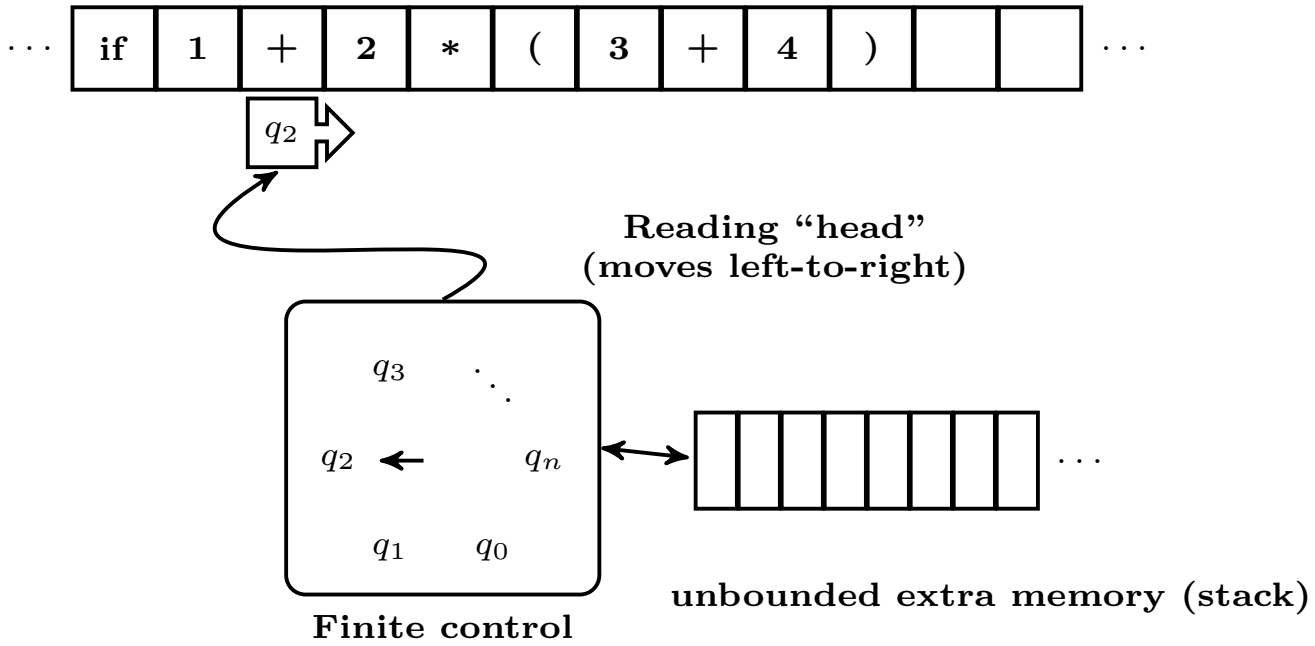


Section

Top-down parsing

Chapter 4 “Parsing: Capsule Summary (tbc.)”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Schematic view



Remember: Derivation and parse trees

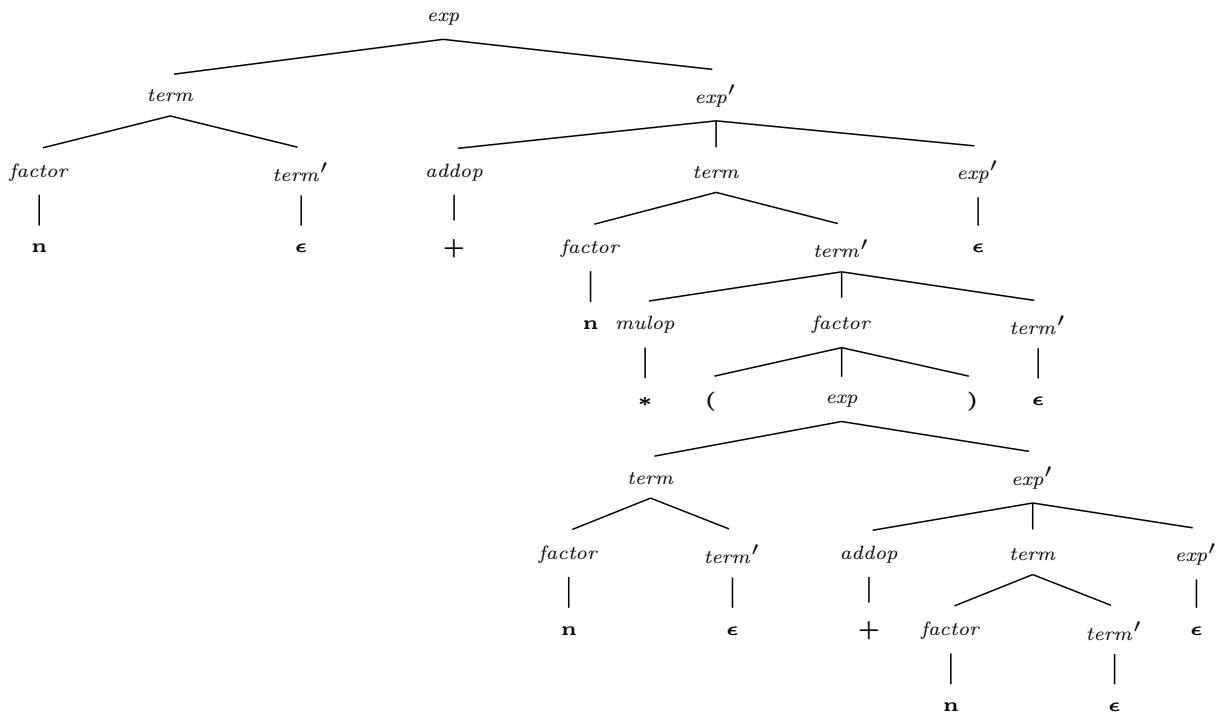
$$
\begin{array}{rcll}
exp & \rightarrow & term\ exp' & (1) \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon & \\
addop & \rightarrow & + \mid - & \\
term & \rightarrow & factor\ term' & \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon & \\
mulop & \rightarrow & * & \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number} &
\end{array}
$$

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



Remember: Derivation and parse trees

$$
\begin{array}{rcll}
exp & \rightarrow & term\ exp' & (1) \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon & \\
addop & \rightarrow & + \mid - & \\
term & \rightarrow & factor\ term' & \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon & \\
mulop & \rightarrow & * & \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number} &
\end{array}
$$



Remember: Derivation and parse trees

- **unambiguous** grammar ⇒ input **determines** parse-tree
- but what about the **derivation**?



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Two source of non-determinism

There are **2 choices to make**:

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

Left-most, right-most, who cares?



Definition (Left-most and right-most derivation)

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

Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Left-most, right-most, who cares?



Definition (Left-most and right-most derivation)

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

Lemma (Left or right, who cares)

$S \Rightarrow_l^* w$ *iff* $S \Rightarrow_r^* w$ *iff* $S \Rightarrow^* w$.

Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Unrealistic parsing device

assume unambiguous grammar

parser with full information

total recall + total foresight

Question

Under these ideal conditions: does the input determines each step of the parser run?



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Oracular



The way we presented the case, it seems more than plausible that a parser with full information, the past as well as the future of its input, can make the right decision at each point, for a given unambigous grammar.

Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Oracular

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



(State and) future must determine next step

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

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

unbounded lookahead





Section

First- and follow-sets

Chapter 4 “Parsing: Capsule Summary (tbc.)”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Informal definition

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

First-sets

- besides informal one: 3 definitions or characterizations given

  based on derivations:

  1. formalization of the informal one,

  based on grammar:

  2. a "recursive" one
  3. an algorithmic one

- first sets of symbols and words
- **nullable** symbols make it (moderately) more complex



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

First-set (derivations)



Definition (First set)

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

$$First_G(A) = \{a \mid A \Rightarrow_G^* a\alpha, \quad a \in \Sigma_T\} + \{\epsilon \mid A \Rightarrow_G^* \epsilon\} \,. \tag{2}$$

Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Recursive characterization

Definition (First set of a symbol (no ϵ-productions))

Given a grammar G and grammar symbol X. The *First-set* of $X \neq \epsilon$, 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

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

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



Algorithmic one (ignoring ϵ)

```
1 initialize(First);
2 while there are changes to any First[A] do
3   for each production A → X1 ... Xn do
4       First[A] := First[A] ∪ First[X1]
5   end;
6 end
```



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

With ϵ

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



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** }		



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)

Follow-sets

- similar **route** than for the first sets: 3 charaterizations

  based on derivations:

  1. formalization of the informal one,

  based on grammar:

  2. a "recursive" one
  3. an algorithmic one



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Follow-set (derivation)



Definition (Follow set)

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

$$Follow_G(A) = \{a \mid S\,\$ \Rightarrow_G^* \alpha_1 A a \alpha_2, \quad a \in \Sigma_T + \{\,\$\,\}\}\,. \tag{3}$$

Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Follow-set (rec.)

Definition (Follow set of a non-terminal)

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

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



Follow-set (algo)

```
1  Follow[S] := {$}
2  for all non-terminals A ≠ S do
3     Follow[A] := {}
4  end
5  while there are changes to any Follow−set do
6     for each production A → X1 ... Xn do
7        for each Xi which is a non−terminal do
8           Follow[Xi] := Follow[Xi] ∪ ( First(Xi+1 ... Xn) \ {ϵ})
9           if ϵ ∈ First(Xi+1 Xi+2 ... Xn)
10          then Follow[Xi] := Follow[Xi] ∪ Follow[A]
11       end
12    end
13 end
```



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Run of the follow-set algo

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



What's next

- massaging grammars
- Top-down parsing (cont'd)
- Bottom-up parsing



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)



Section

Massaging grammars

Chapter 4 “Parsing: Capsule Summary (tbc.)”
Course “Compiler Construction”
Martin Steffen
Spring 2022

From before

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

two troublesome pattern that require massage

- left-recursion
- common left factors



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)

Definition (Left-recursion and common left-factors.)

A *left-recursive* production is of the form

$$A \rightarrow A\alpha \tag{4}$$

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

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \qquad \text{where } \alpha \neq \boldsymbol{\epsilon} \tag{5}$$

Both commonplace

$$exp \rightarrow exp + term \ .$$

$$\begin{array}{lcl} if\text{-}stmt & \rightarrow & \textbf{if} \ (\ exp \ ) \ stmt \ \textbf{end} \\ & | & \textbf{if} \ (\ exp \ ) \ stmt \ \textbf{else} \ stmt \ \textbf{end} \ . \end{array}$$



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Left-recursion removal

Two stage transformation or algo

1. immediate left-recursion
2. (indirect) left-recursion

Immediate left-recursion

- pretty simple transformation
- introduce new non-terminal and an ϵ-production

Definition (Left-recursion removal: immediate recursion)

The transformation for removal of *immediate* left recursion is as follows:

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

before after



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Left-recursion: indirect case

```
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
```

- Note the case $i = j$ on the "diagonal"



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

$A = A_1,\ B = A_2.$

A_1, A_1 (direct)

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

A_2, A_1 (indirect)

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

A_2, A_2 (direct)

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

left-factor removal

before

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

after

$$
\begin{array}{rcl}
\textit{if-stmt} & \rightarrow & \textbf{if} \; (\; \textit{exp} \;) \; \textit{stmts} \; \textit{else-or-empty} \\
\textit{else-or-empty} & \rightarrow & \textbf{else} \; \textit{stmts} \; | \; \epsilon
\end{array}
$$



One thing to watch out for



- previous example **too** simple

What to do in a situation like this?

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

Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

One thing to watch out for

- previous example **too** simple

What to do in a situation like this?

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

- Should one factor the **longest** prefix ($\mathbf{ab}$ or should one factor with the **most rules**?
- factorization is not a one-step transformation



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Answer: choose the longest

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 $\alpha \neq \epsilon$
    then
      let $A \to \alpha_1 \mid \ldots \mid \alpha_n$ *be all prod. for* A *and suppose that* $\alpha_1, \ldots, \alpha_k$ *share* α *so that* $A \to \alpha\beta_1 \mid \ldots \mid \alpha\beta_k \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$, *that the* β_j*'s share no common prefix, and that the* $\alpha_{k+1}, \ldots, \alpha_n$ do not share α.
      replace rule $A \to \alpha_1 \mid \ldots \mid \alpha_n$ *by the rules*
      $A \to \alpha A' \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$
      $A' \to \beta_1 \mid \ldots \mid \beta_k$
    end
  end
end



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)



Section

LL-parsing, mostly LL(1)

Chapter 4 “Parsing: Capsule Summary (tbc.)”
Course “Compiler Construction”
Martin Steffen
Spring 2022

LL

- remember earlier section: principles of top-down parsing, derivation etc.
- Now: armed with first- and follow sets.

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Easy recursive recipe

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

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Illustration

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

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

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

What's with left-recursion?



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Criterion

Lemma (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 $\alpha_1 \neq \alpha_2$:

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

More complex

Lemma (LL(1))

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

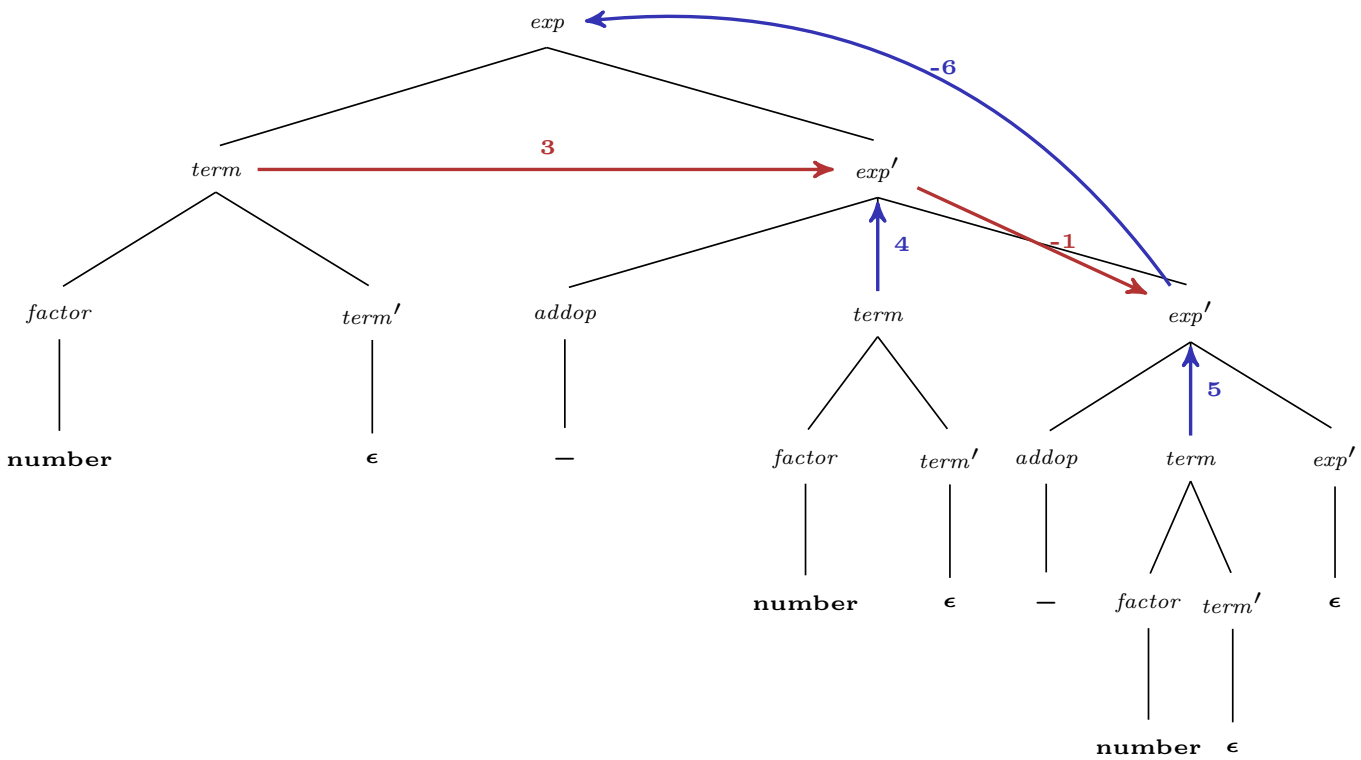
$$First_1(\alpha_1 Follow_1(A)) \cap First_1(\alpha_2 Follow_1(A)) = \emptyset \ .$$



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Side track: repair ill-associated ASTs

- remember: left-factor removal can **change associativity**
- recursive parse functions with extra "accumulator" argument



Side track: OO design for ASTs

Recipe:

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Illustration

```
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;}
}
```

Illustration (2): $3 - (4 - 5)$

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Table-based LL(1)-parsing

- instead of recursion, **explicit stack** and **table**

LL(1) parsing table

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

- e.g. two-dimensional array
- "double entries": **conflict**



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Recipe

cf. also the previous (equivalent) LL(1)-criterion

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

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Example(s)

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**	

more in the exercises, exams etc.



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Algo (not discussed in class)

```
1  while the top of the parsing stack ≠ $
2     if the top of the parsing stack is terminal a
3        and the next input token = a
4     then
5        pop the parsing stack;
6        advance the input; // ``match''
7     else if    the top the parsing is non-terminal A
8           and   the next input token is a terminal or $
9           and   parsing table M[A, a] contains
10                production A → X1 X2 ... Xn
11          then (* generate *)
12                pop the parsing stack
13                for i := n to 1 do
14                push Xi onto the stack;
15          else error
16    if    the top of the stack = $
17    then accept
18 end
```



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Run

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



Error handling

not pensum



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)



Section

Bottom-up parsing

LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Chapter 4 “Parsing: Capsule Summary (tbc.)”
Course “Compiler Construction”
Martin Steffen
Spring 2022

Intro

- LR-parsing
- **shift-reduce** parsing

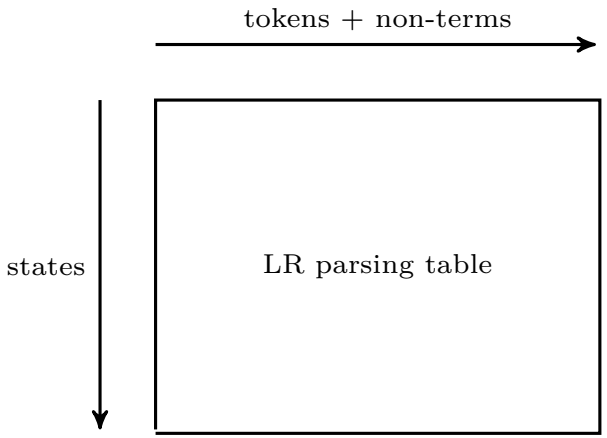
different variations

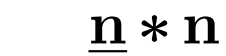
- LR(0)
- SLR(1)
- LALR(1)
- LR(1)



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

LR or shift-reduce parsers





(a) Reduction

(b) Right derivation

Figure

$$\underline{\mathbf{n}} * \mathbf{n} \quad \hookrightarrow \quad \underline{factor} * \mathbf{n}$$

(a) Reduction

(b) Right derivation

Figure



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)

$$\underline{\mathbf{n}} * \mathbf{n} \hookrightarrow \underline{factor} * \mathbf{n} \hookrightarrow term * \underline{\mathbf{n}}$$

(b) Right derivation

(a) Reduction

Figure



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

$$\underline{\mathbf{n}} * \mathbf{n} \hookrightarrow \underline{factor} * \mathbf{n} \hookrightarrow term * \underline{\mathbf{n}} \hookrightarrow \underline{term * factor}$$

(a) Reduction

(b) Right derivation

Figure



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

$$\underline{\mathbf{n}} * \mathbf{n} \hookrightarrow \underline{factor} * \mathbf{n} \hookrightarrow term * \underline{\mathbf{n}} \hookrightarrow \underline{term * factor} \hookrightarrow \underline{term}$$

(a) Reduction

$$\underline{factor} * \mathbf{n} \Rightarrow term * \underline{\mathbf{n}} \Rightarrow \underline{term * factor} \Rightarrow \underline{term}$$

(b) Right derivation

Figure



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

$$\underline{\mathbf{n}} * \mathbf{n} \hookrightarrow \underline{factor} * \mathbf{n}$$

(a) Reduction

$$\underline{factor} * \mathbf{n} \hookrightarrow term * \underline{\mathbf{n}} \hookrightarrow \underline{term * factor} \hookrightarrow \underline{term} \hookrightarrow exp$$

(b) Right derivation

Figure



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

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

(a) Reduction

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

(b) Right derivation

Figure

reduction in reverse = right-most derivation

Configuration

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Two kinds of steps

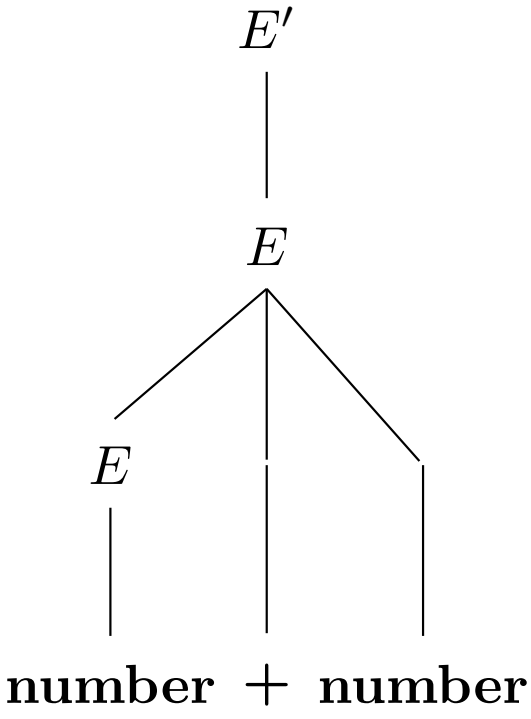
Shift: move the next input symbol (terminal) over to the top of the stack ("push").

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

plus: **accept** (which corresponds to some specific reduce step and some other conditions)



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)



(a) Parse tree

	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	$\$\,E + \mathbf{n}$	$\$$	red. $E \to E + \mathbf{n}$
6	$\$\,E$	$\$$	red.: $E' \to E$
7	$\$\,E'$	$\$$	accept

(b) run resp. reduction

Figure

Past determines the parser-machine's state

- in principle: unbounded memory
- but stack oriented, not RAM
- parser can *forget* (pop)
- past in the form of the *stack content:* **determines state**
- **infinitely** many stack contents: **too much**

LR(0)-DFA

DFA "working" on the stack. Stack decides the state of the DFA (and thus the state of the parser machine). Allowed stack content is a regular language!

How to construct the DFA?



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

LR(0) as easy pre-stage



Definition (LR(0) item)

An **LR(0) item** for a production $A \rightarrow \beta\gamma$ is of the form

$$A \rightarrow \beta.\gamma$$

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

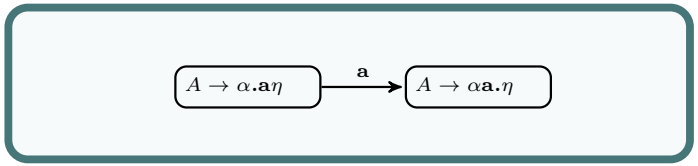
Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

2 routes

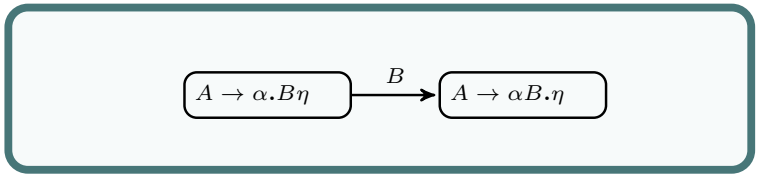
- construction of LR(0)-NFA first
- direct construction of LR(0)-DFA



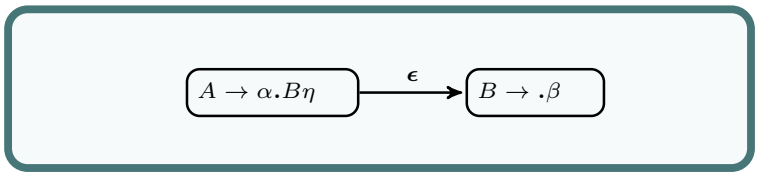
Transitions



(a) terminal



(b) non-terminal



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

Figure: Transitions of the LR(0)-NFA



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

How to think of that?

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

complete items connected to **reduce steps**

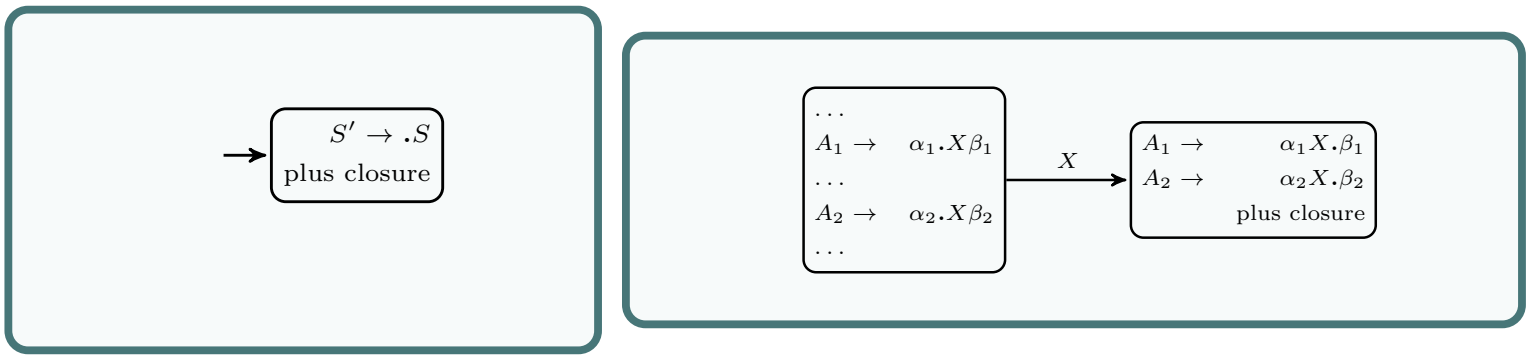
- initial state of NFA and initial configuration of machine
- accepting state of machine



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Direct construction of LR(0)-DFA

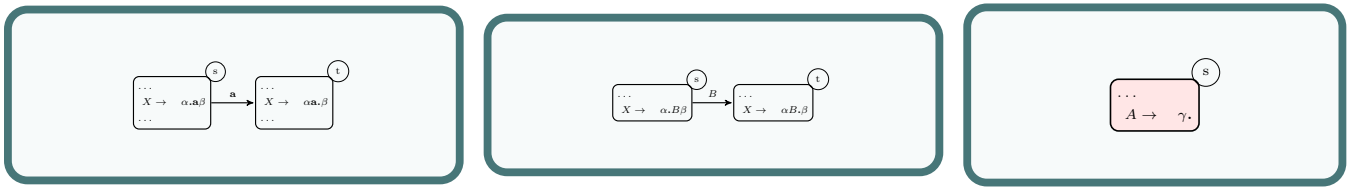
ϵ-closure: if $A \to \alpha.B\gamma$ is an item in a state, and the grammar has productions $B \to \beta_1 \mid \beta_2 \ldots$. Then **add** items $B \to .\beta_1$, $B \to .\beta_2$ ... to the state. Continue that process, **until saturation**.



(a) initial state

(b) transitions

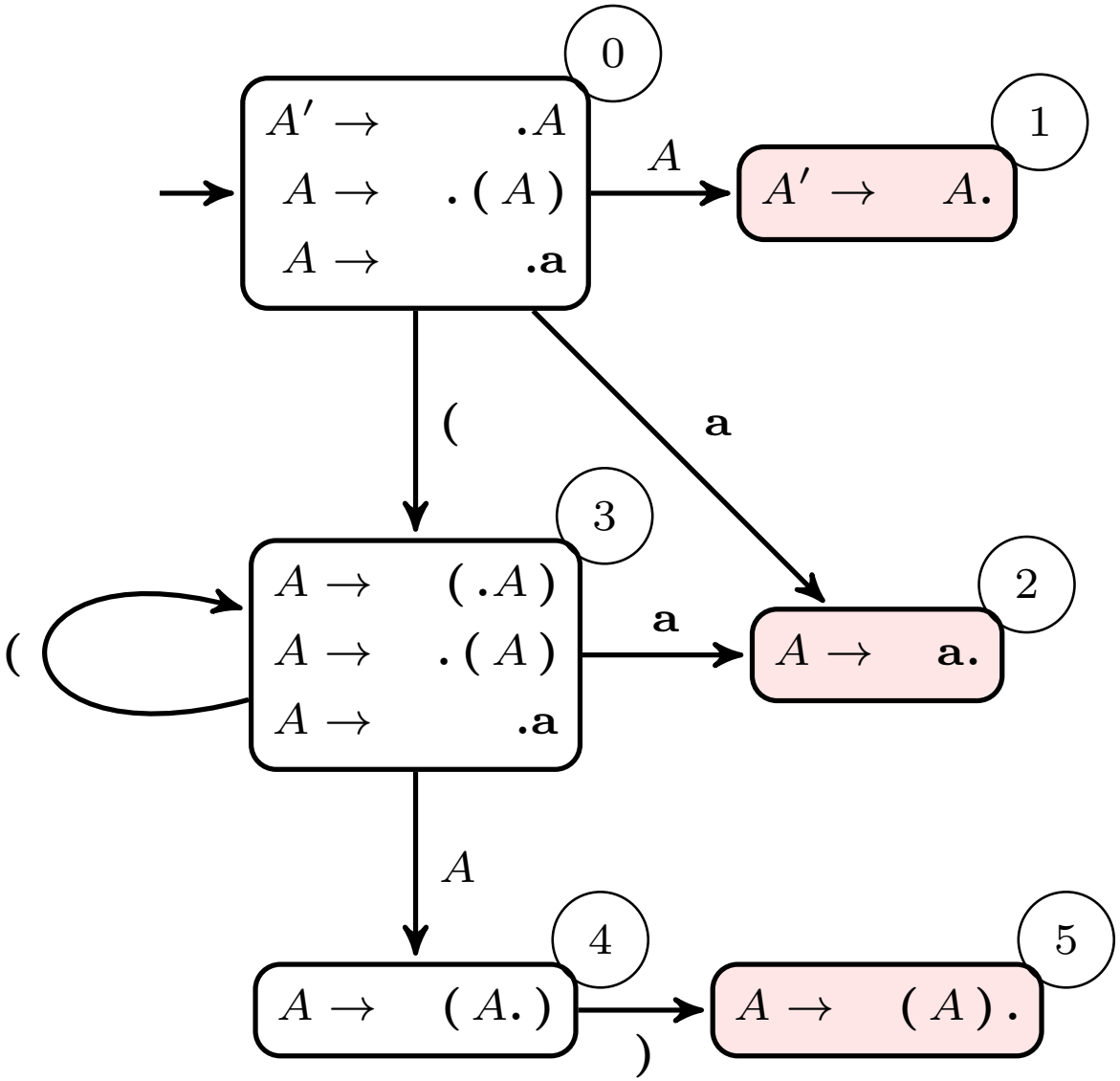
Figure: LR(0)-DFA



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

Figure: Parser machine transitions

LR(0)-DFA example



Table

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

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)



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

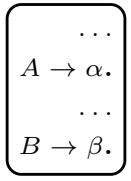
SLR

- Slight modification of LR(0)
- **Same** LR(0) automaton, refined decision criterion
- take **Follow-set** into account

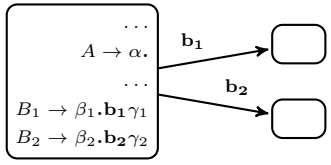


Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

LR(0)-conflicts & how-to deal with them



(a) Reduce-reduce



(b) Shift-reduce

Figure: LR(0) conflicts

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

SLR-parsing table (addition)

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



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Algorithm (not reproduced here)

Minor adaptation of the one shown before (checking also on the follow-sets)



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

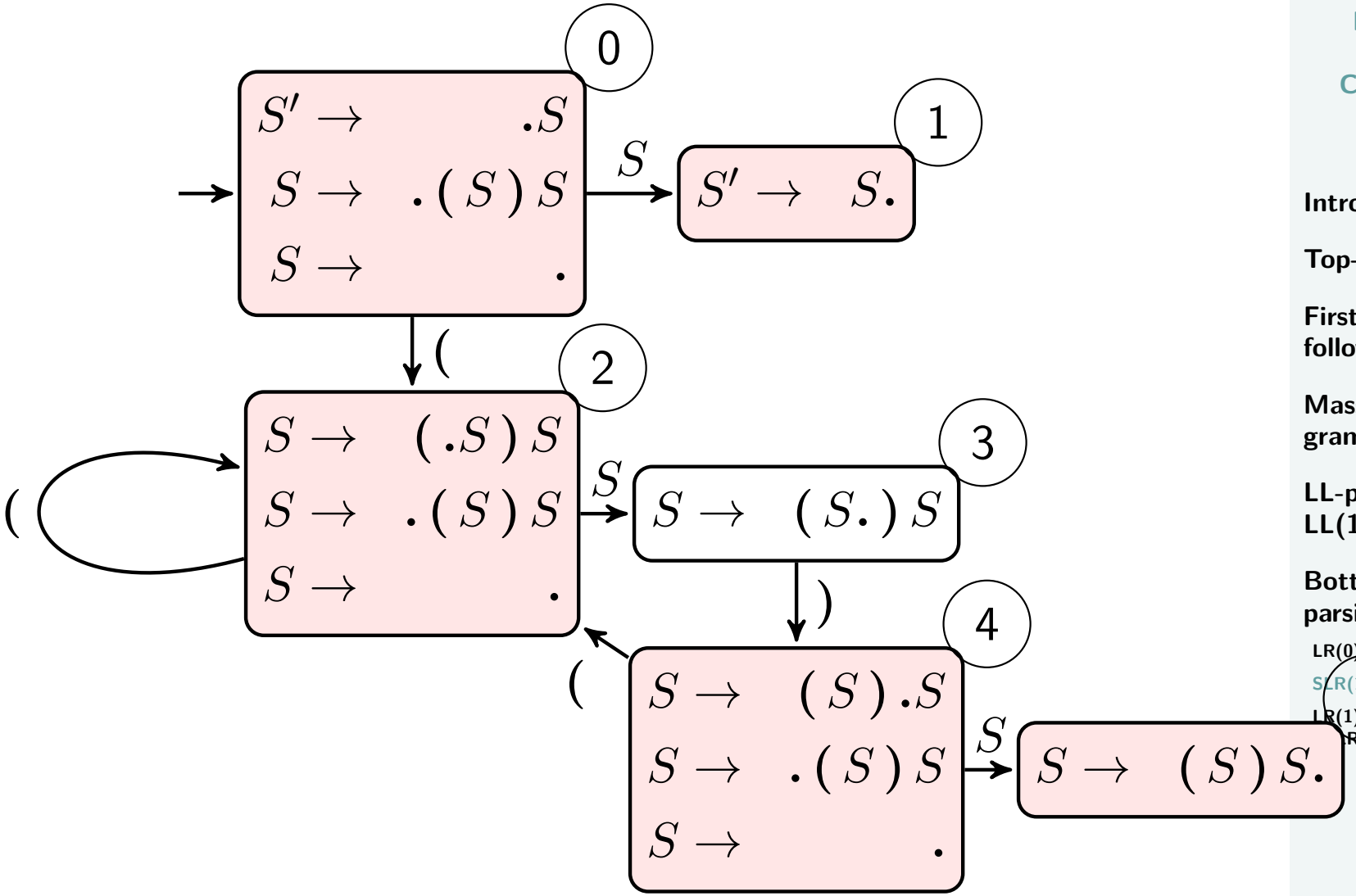
Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)

DFA for parentheses



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



Additionally

- further examples
- parser runs (same as before actually)
- ambiguous grammars: shift- vs. reduce?
- for instance: dangling else “solution”



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

LR(1)



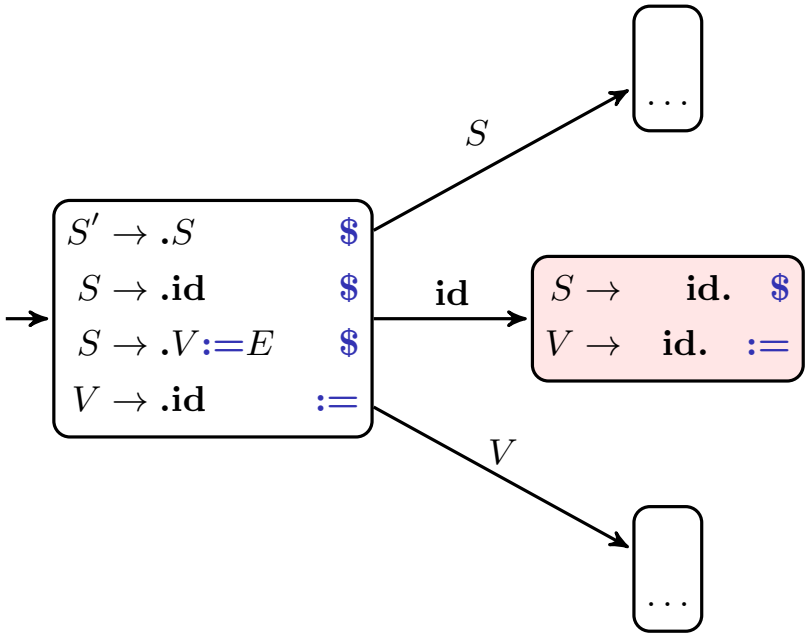
- now **one look-ahead for real**
- many states
-

LR(1)-item

building items with look-ahead baked in, but same principle

Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

Sample LR(1) DFA



LALR(1)

- yacc, Cup etc

LALR1 DFA

1. build LR(1)-DFA
   - many states
   - LR(1) with baked-in look-ahead
2. forget the look-ahead in the items
   - collapse the many states
   - LALR(1)-DFA same amount of states as LR(0)-DFA



Intro
Top-down parsing
First- and follow-sets
Massaging grammars
LL-parsing, mostly LL(1)
Bottom-up parsing
LR(0)
SLR(1) parsing
LR(1) parsing and LALR(1)

error handling

again not pensum



Intro

Top-down parsing

First- and follow-sets

Massaging grammars

LL-parsing, mostly LL(1)

Bottom-up parsing

LR(0)

SLR(1) parsing

LR(1) parsing and LALR(1)