

Chapter 4

Parsing

Course “Compiler Construction”

Martin Steffen

Spring 2023



Chapter 4

Learning Targets of Chapter "Parsing".

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





Chapter 4

Outline of Chapter “Parsing”.

Introduction to parsing

Top-down parsing

First and follow sets

Massaging grammars

LL-parsing (mostly LL(1))

Bottom-up parsing



Section

Introduction to parsing

Chapter 4 “Parsing”
Course “Compiler Construction”
Martin Steffen
Spring 2023

What's a parser generally doing

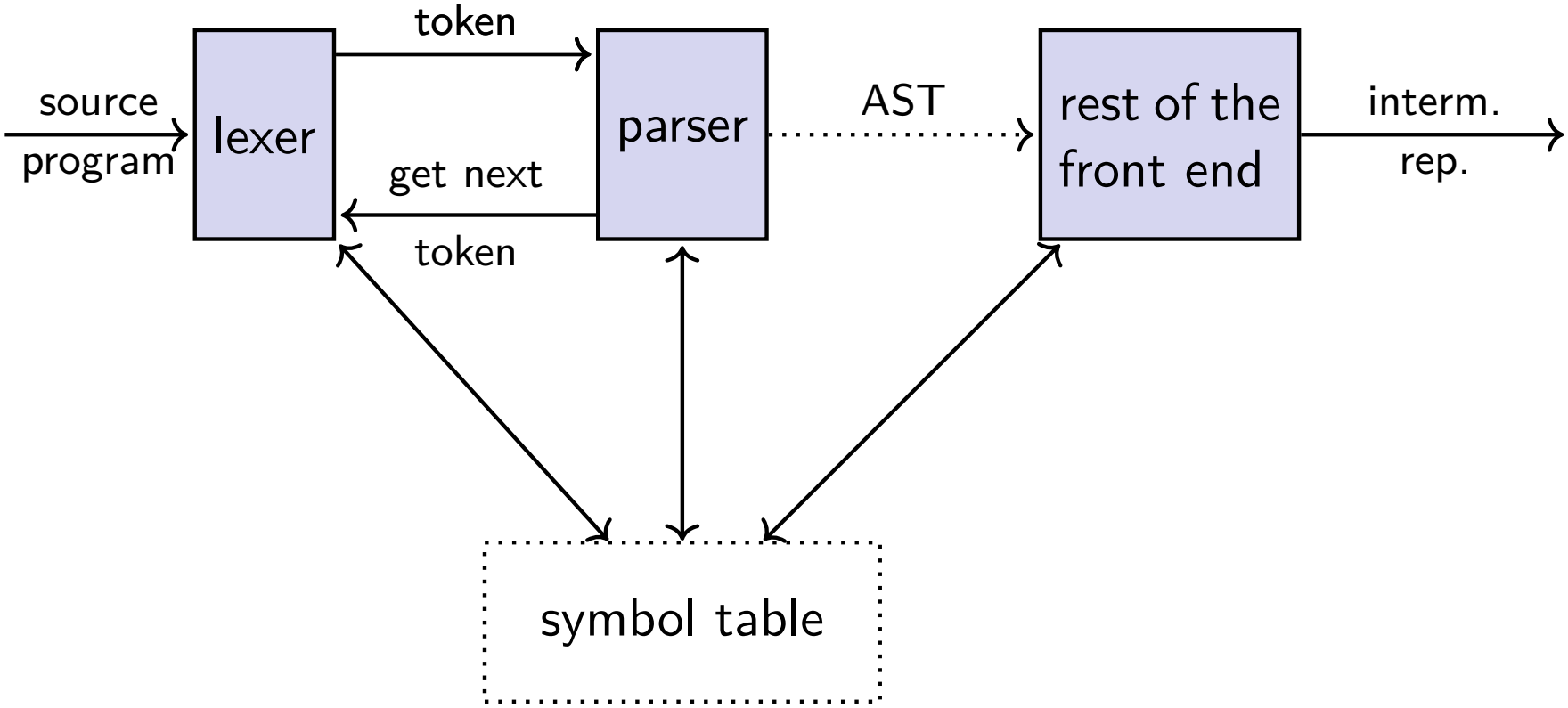
task of parser = syntax analysis

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing


source
program
lexer
token
get next
token
parser
AST
rest of the
front end
interm.
rep.
symbol table

Top-down vs. bottom-up

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

Bottom-up	**Top-down**
Parse tree is being grown from the leaves to the root.	Parse tree is being grown from the root to the leaves.



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Parsing restricted classes of CFGs

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Classes of CFG grammars/languages



- *maaaany* have been proposed & studied, including their relationships, the lecture concentrates on

top-down parsing, in particular

- LL(1)
- recursive descent

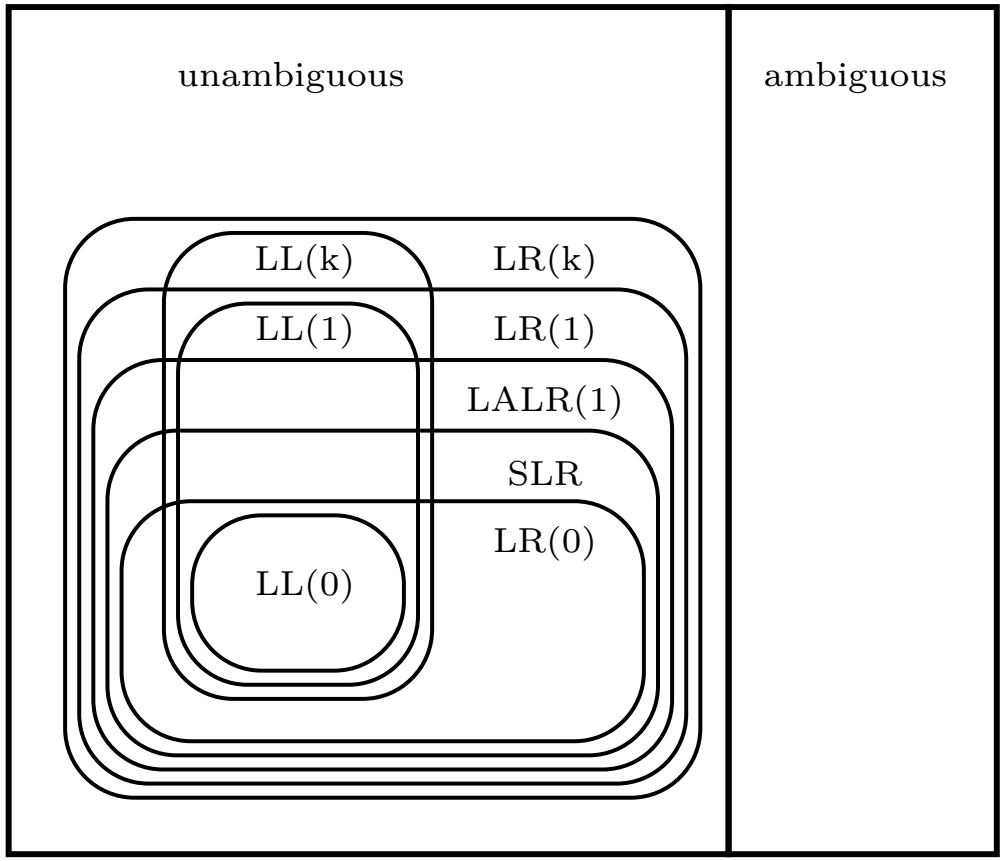
bottom-up parsing

- LR(1)
- SLR
- LALR(1) (the class covered by yacc-style tools)

- grammars typically written in *pure* BNF

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Relationship of some grammar (not language) classes



taken from [1]





Section

Top-down parsing

Chapter 4 “Parsing”
Course “Compiler Construction”
Martin Steffen
Spring 2023

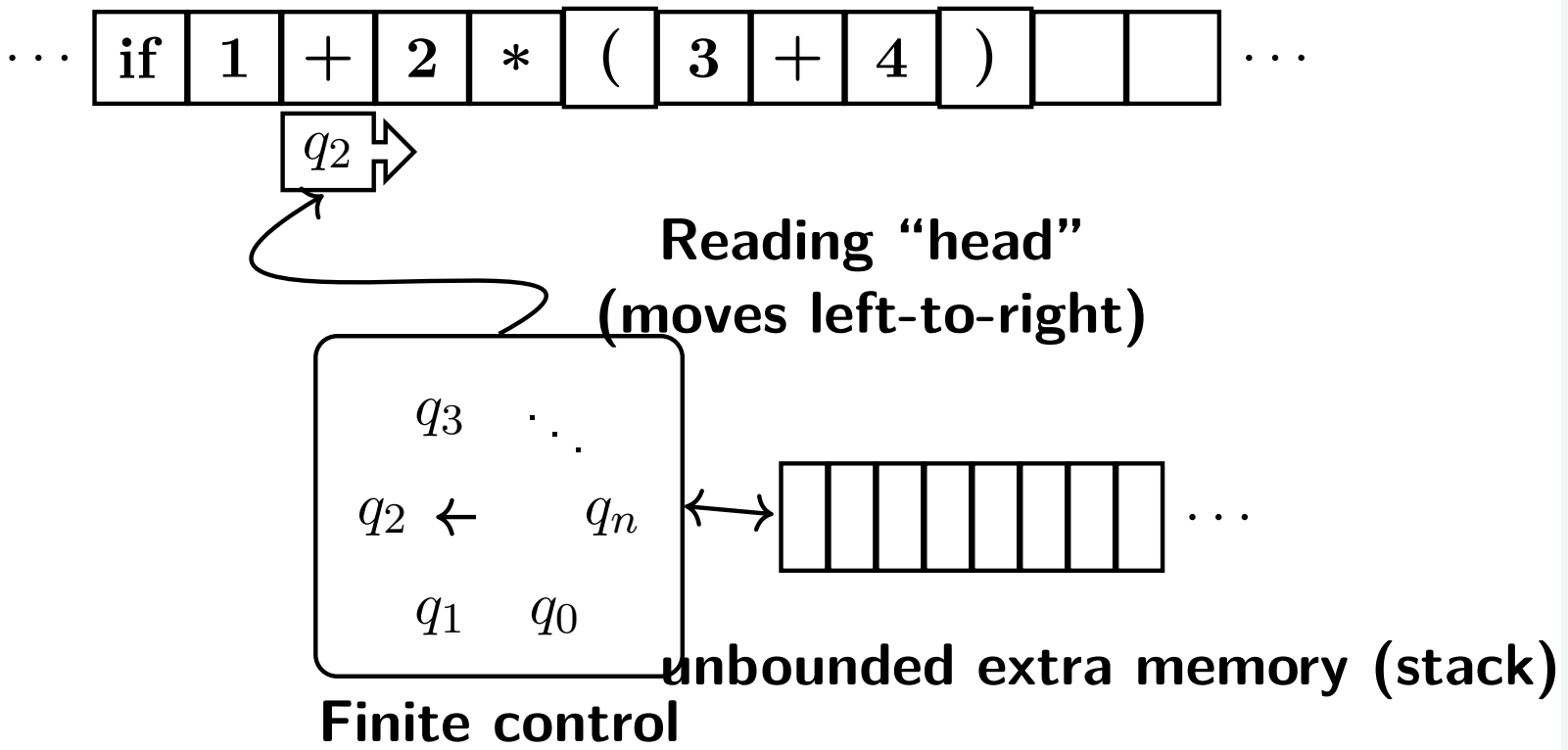
General task (once more)

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



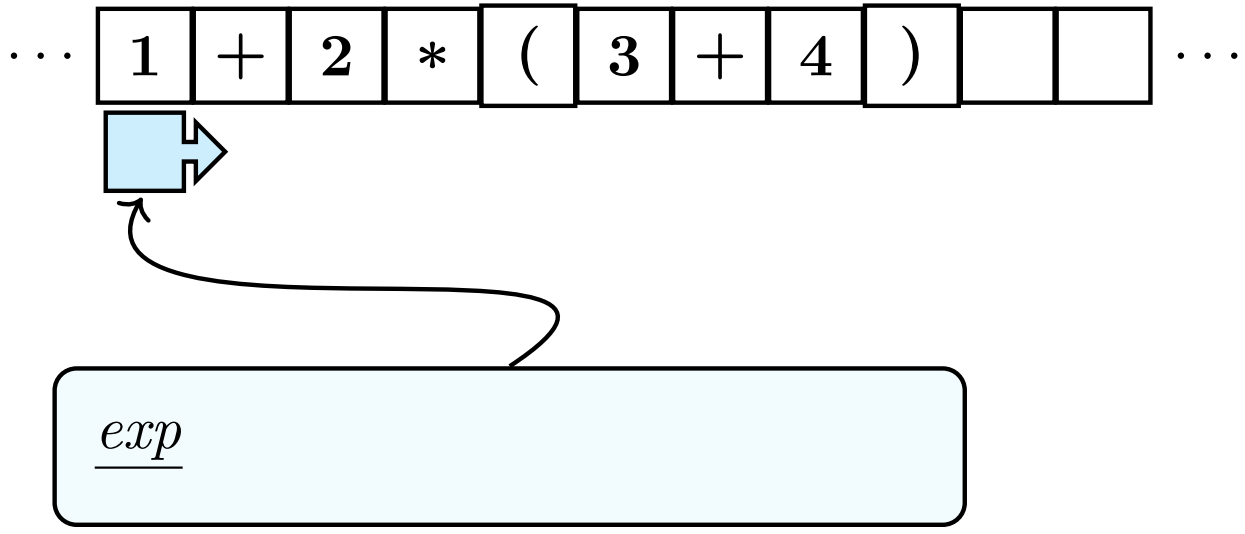
Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Schematic view on "parser machine"



Note: sequence of *tokens* (not characters)

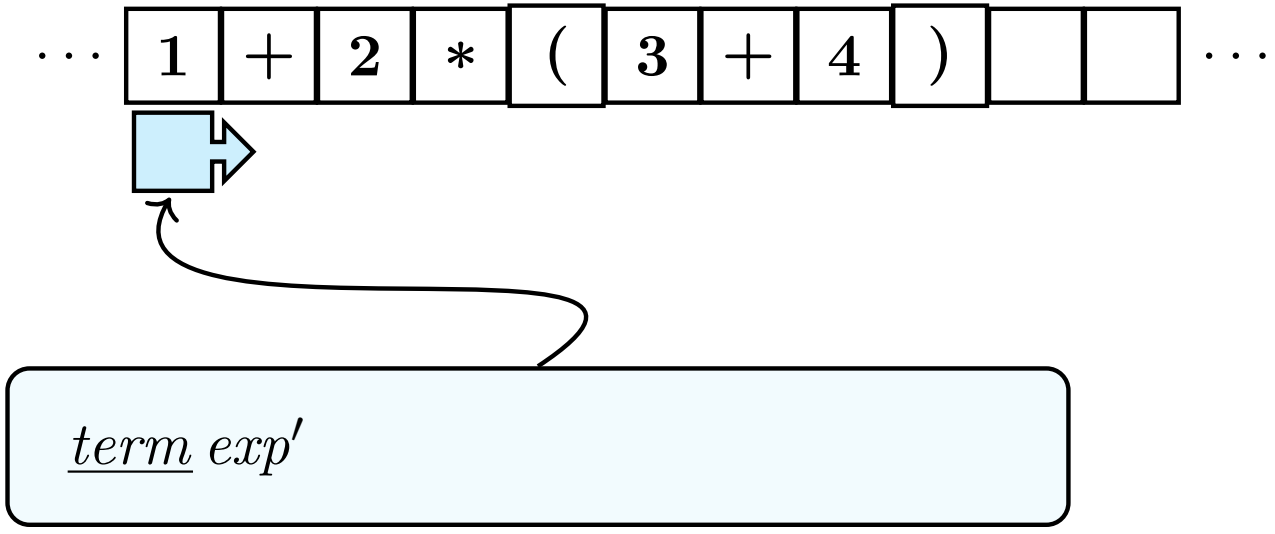
Derivation of an expression



factors and terms

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

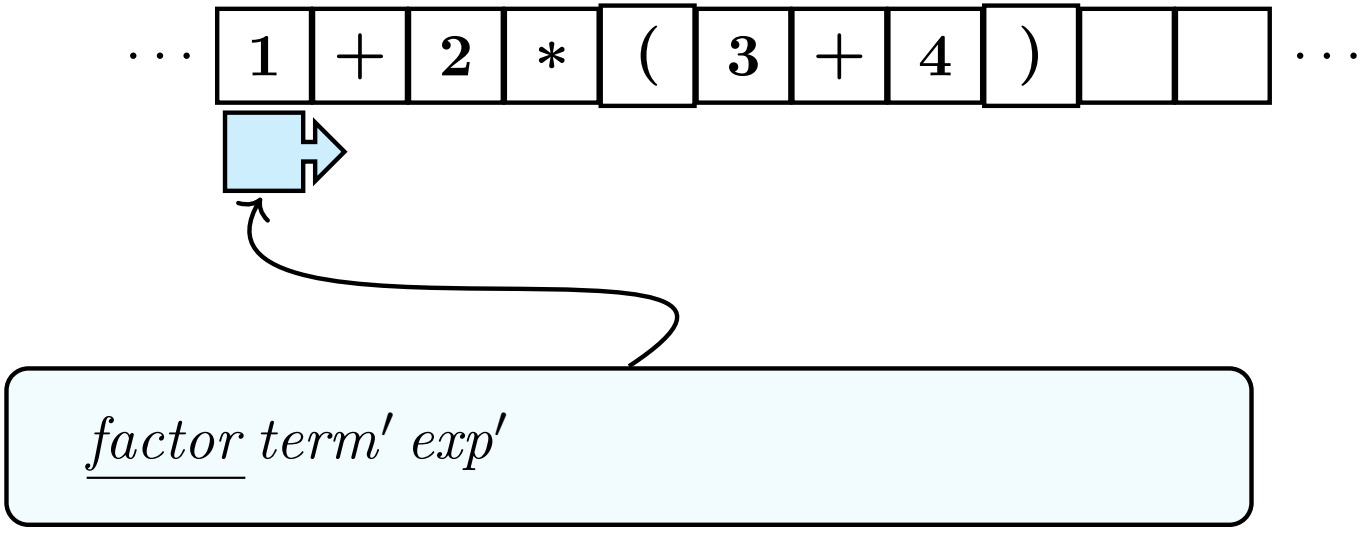
Derivation of an expression



factors and terms

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

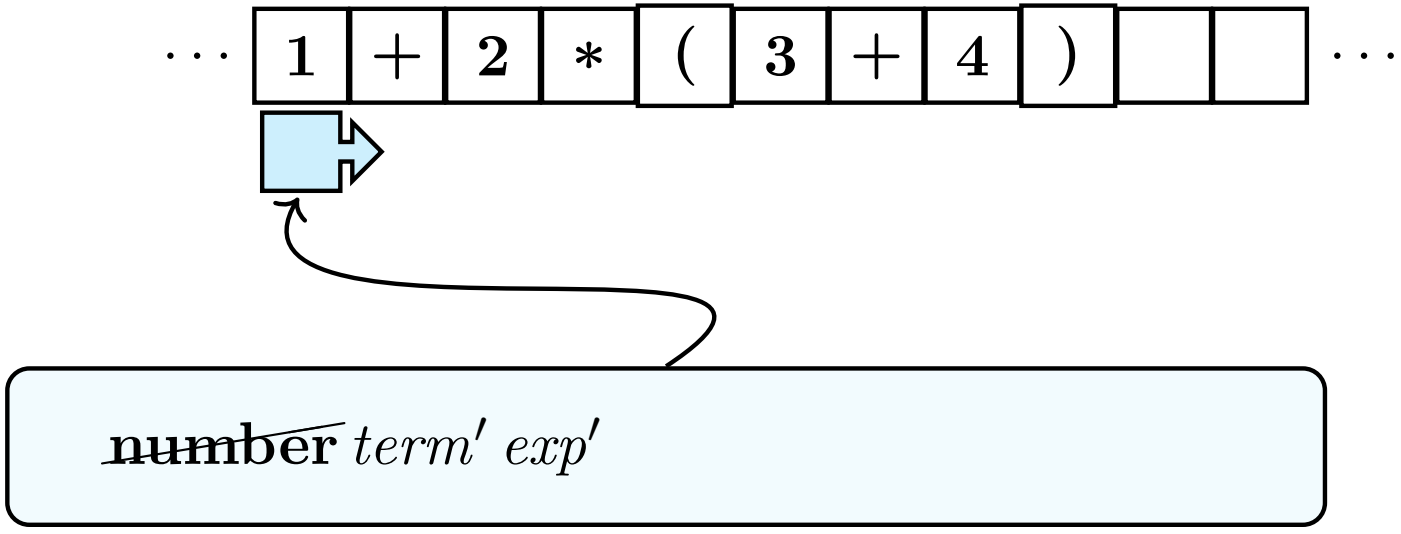
Derivation of an expression



factors and terms

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

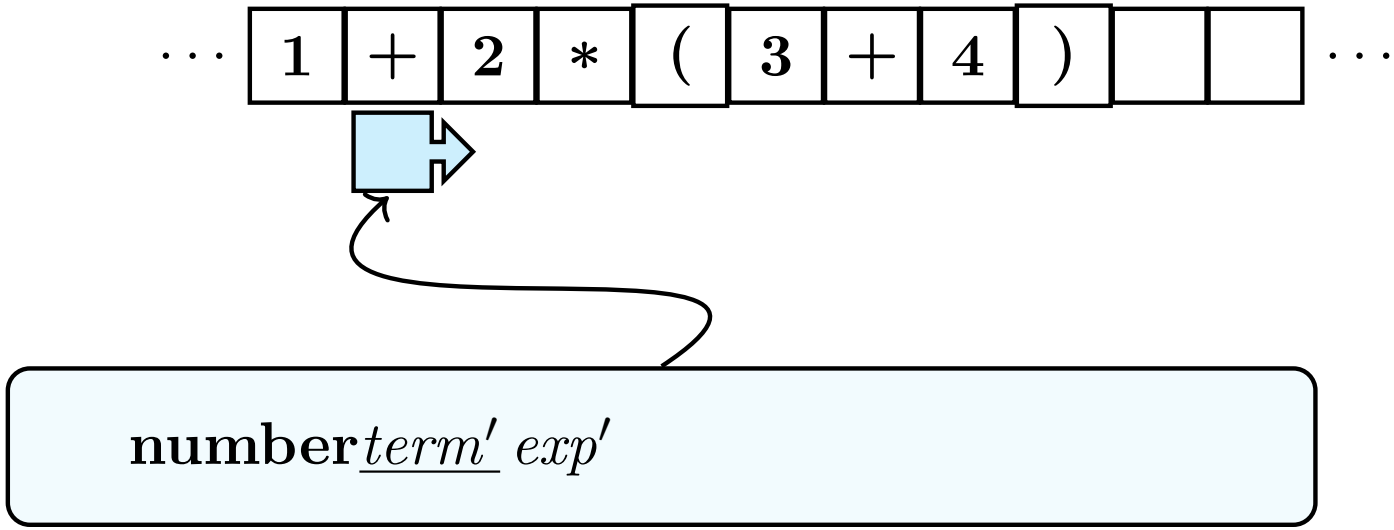
Derivation of an expression



factors and terms

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

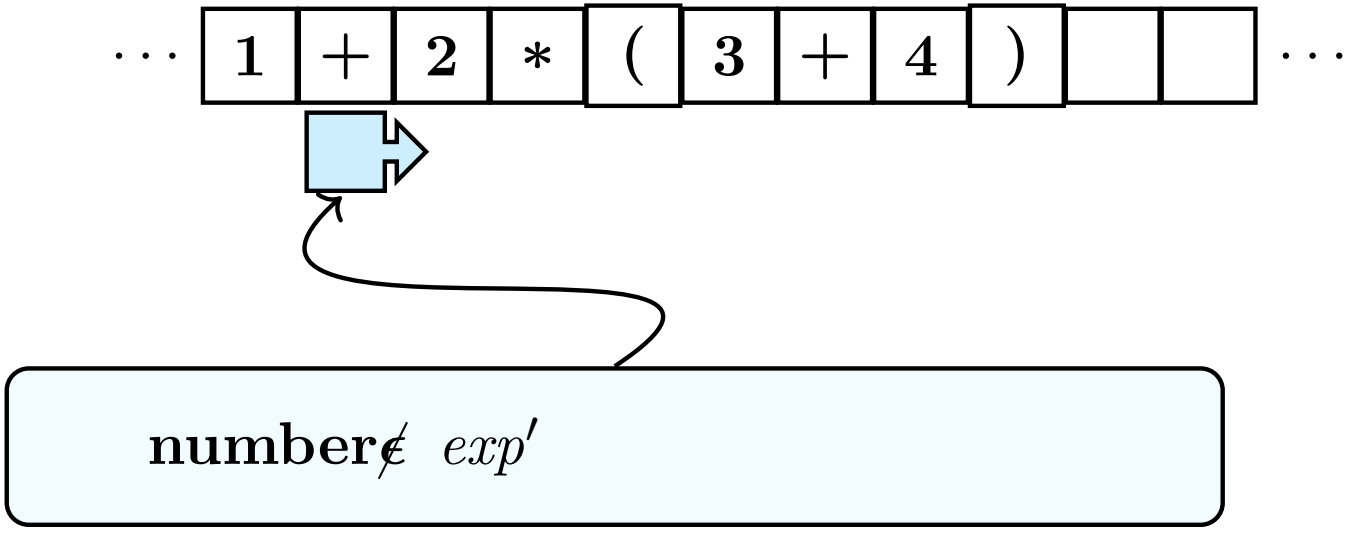
Derivation of an expression



factors and terms

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

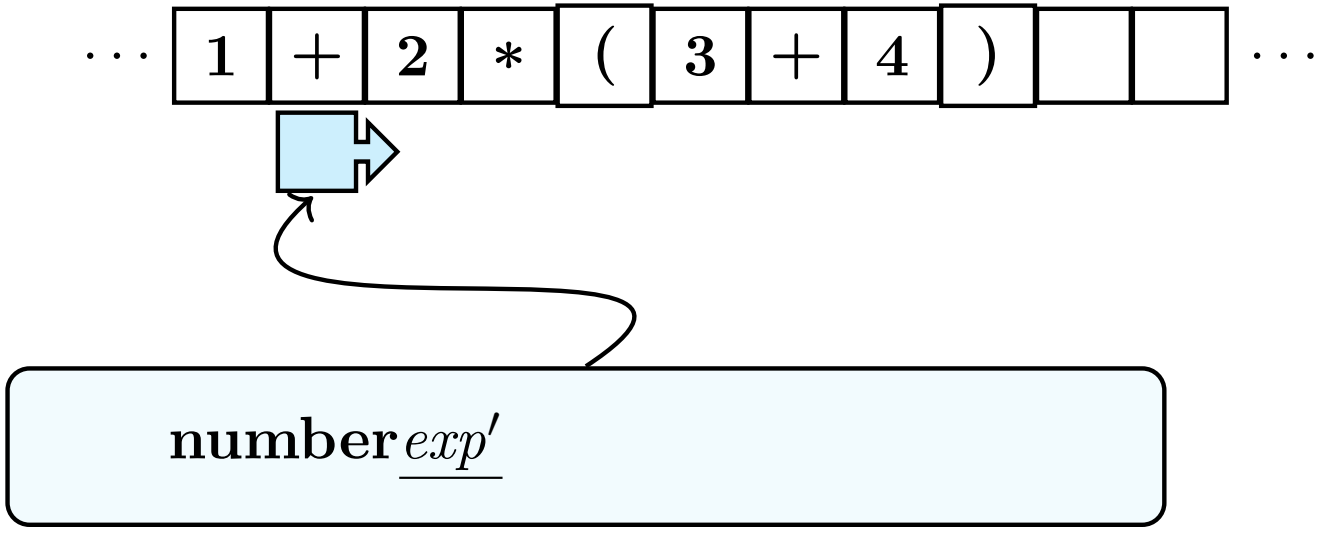
Derivation of an expression



factors and terms

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

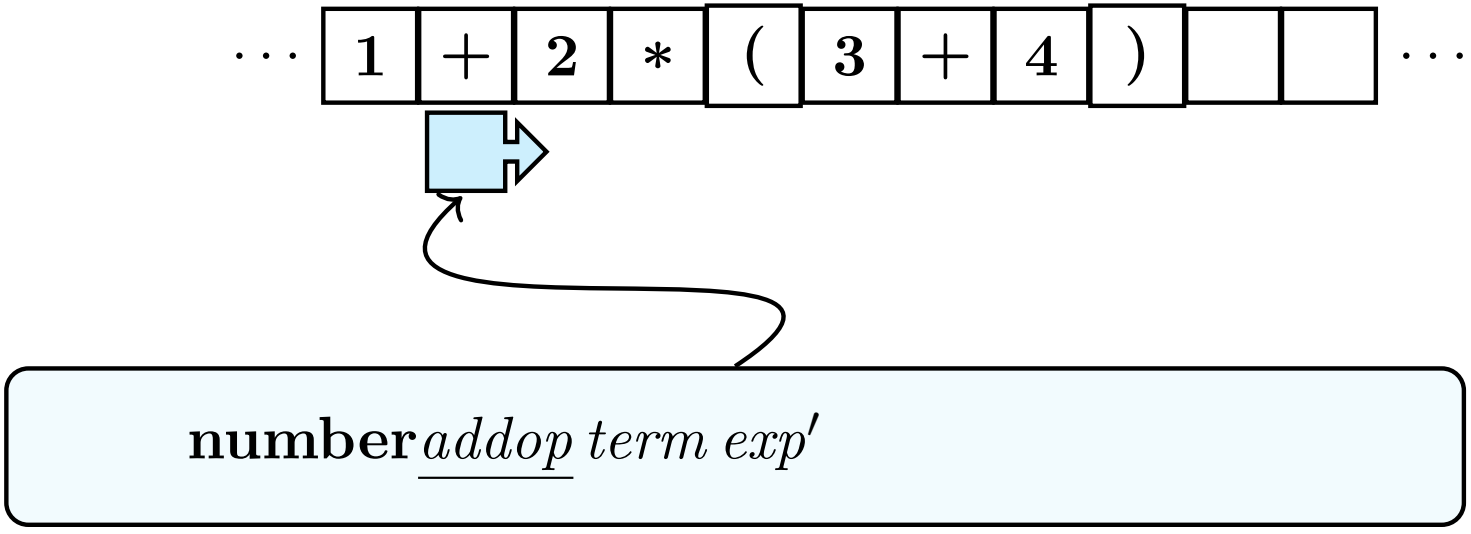
Derivation of an expression



factors and terms

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

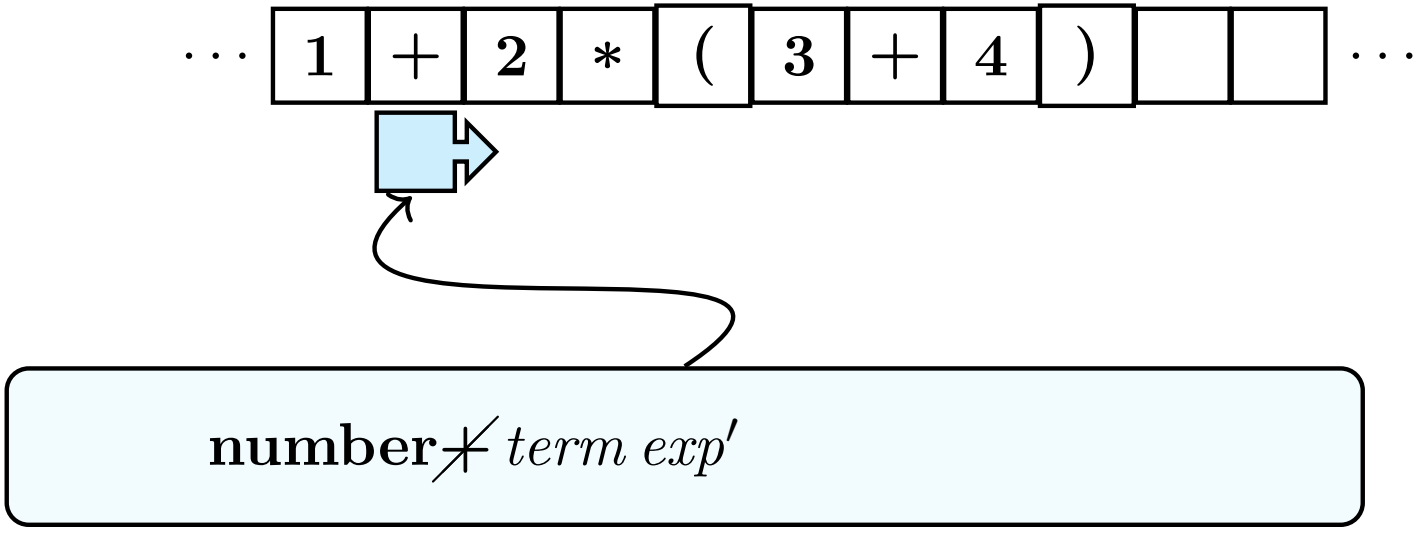
Derivation of an expression



factors and terms

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

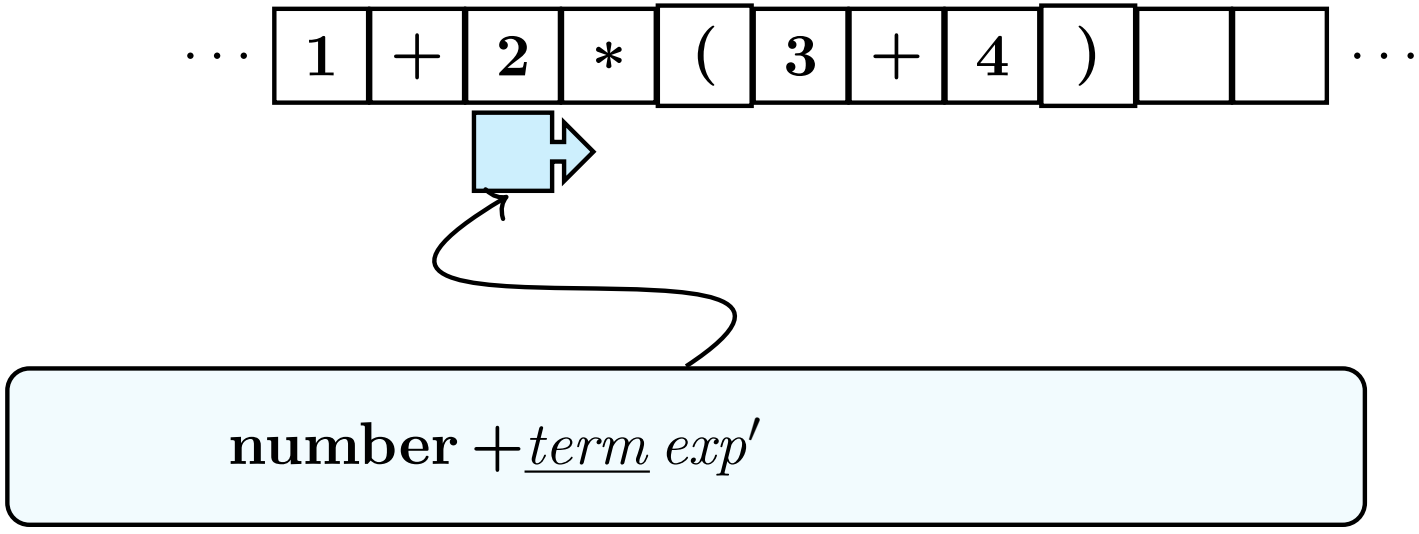
Derivation of an expression



factors and terms

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

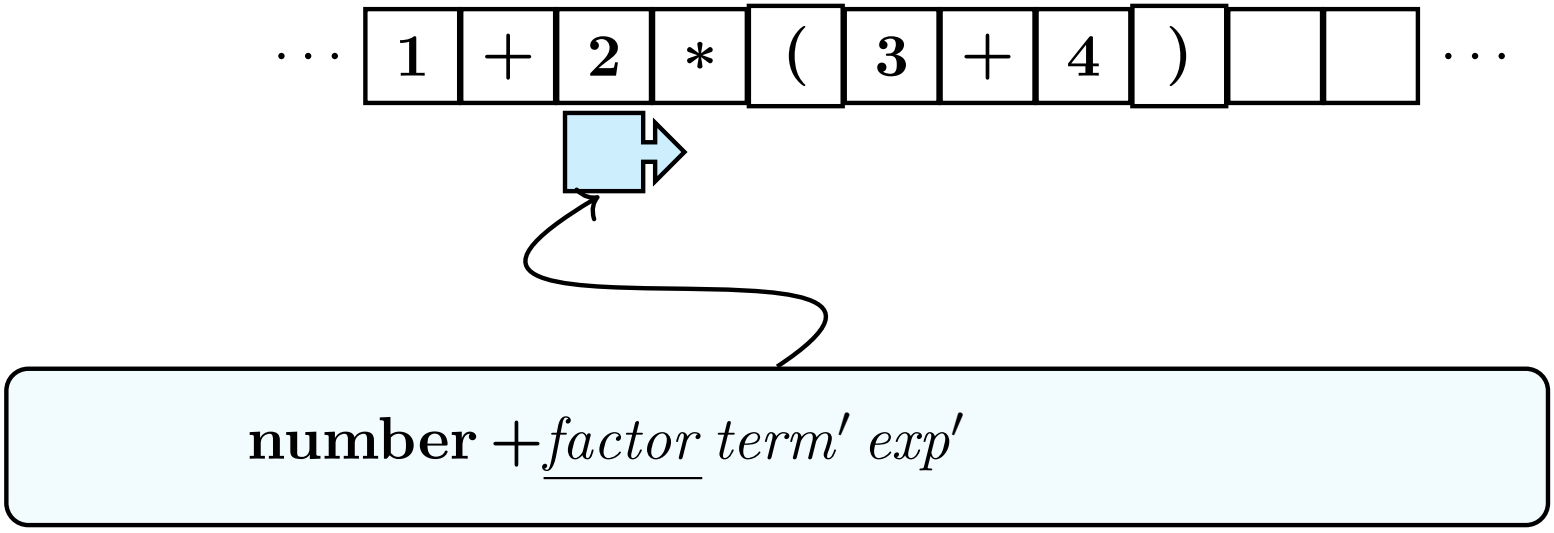
Derivation of an expression



factors and terms

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

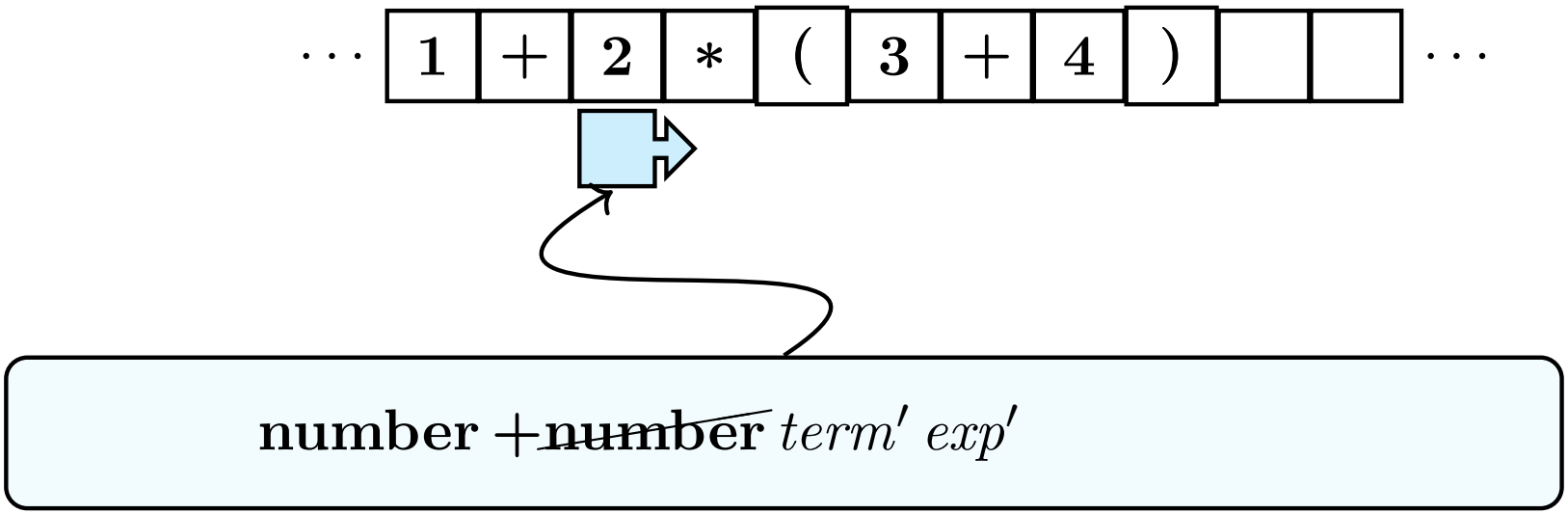
Derivation of an expression



factors and terms

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

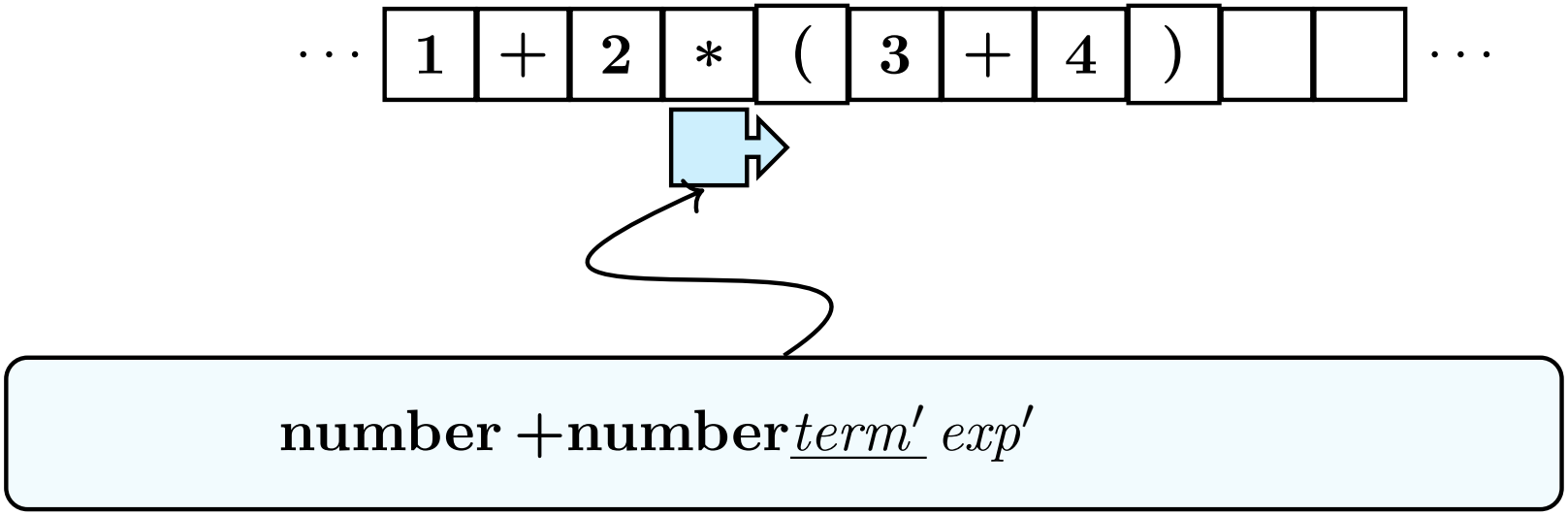
Derivation of an expression



factors and terms

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

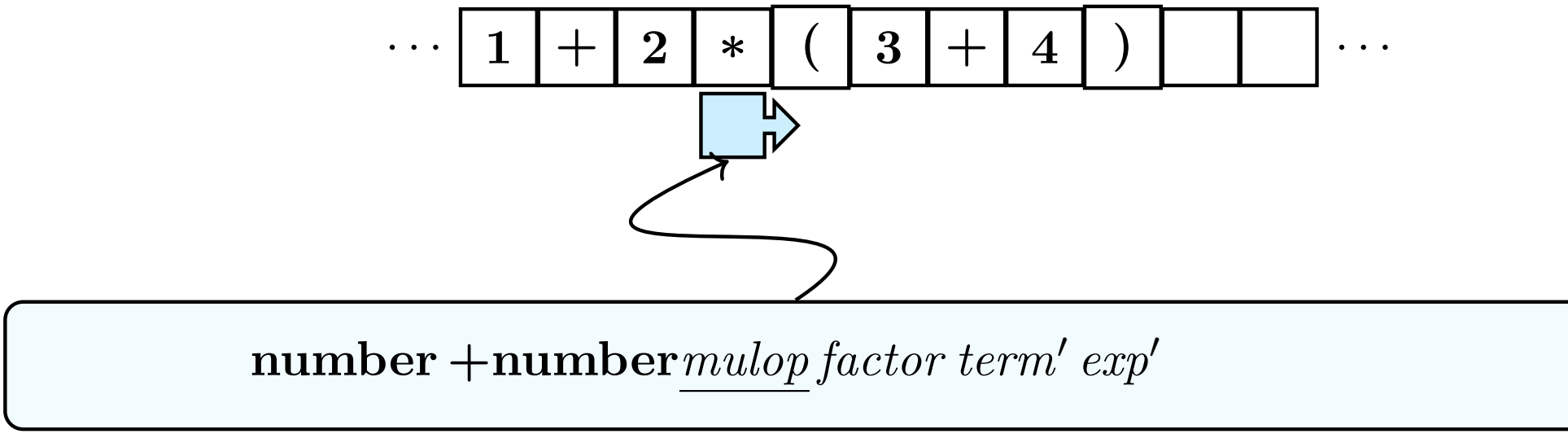
Derivation of an expression



factors and terms

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

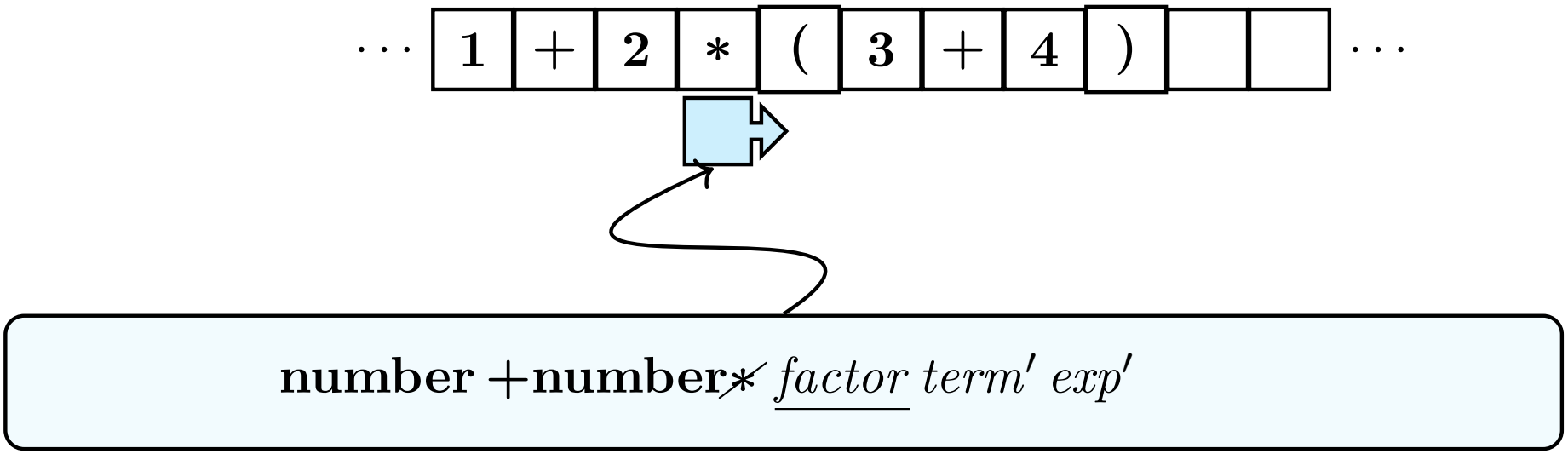
Derivation of an expression



factors and terms

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

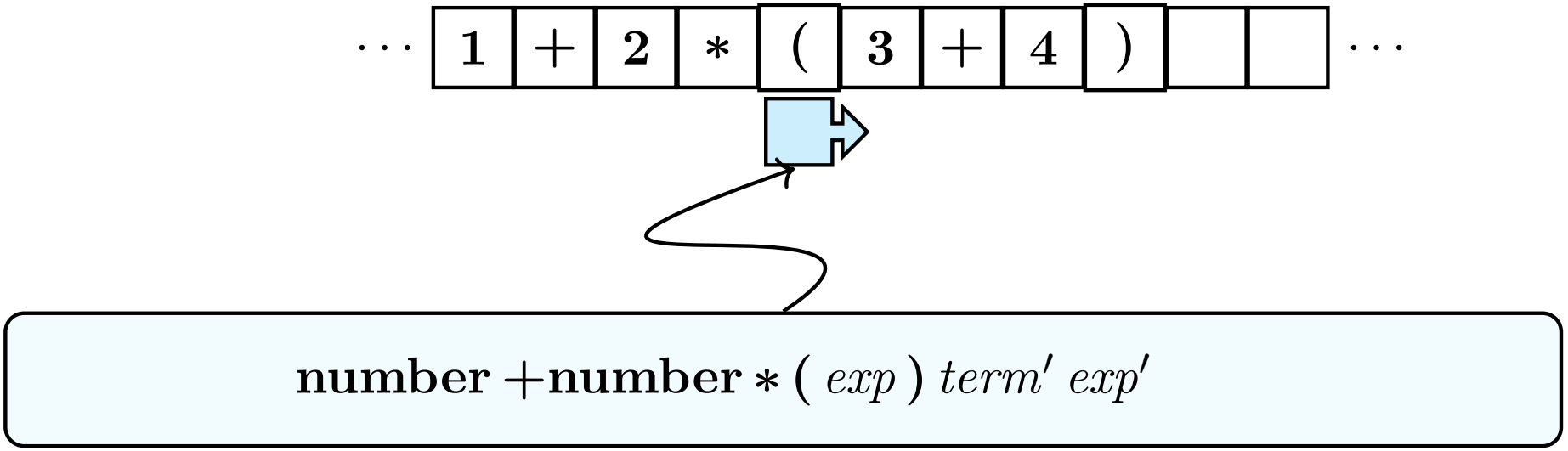
Derivation of an expression



factors and terms

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

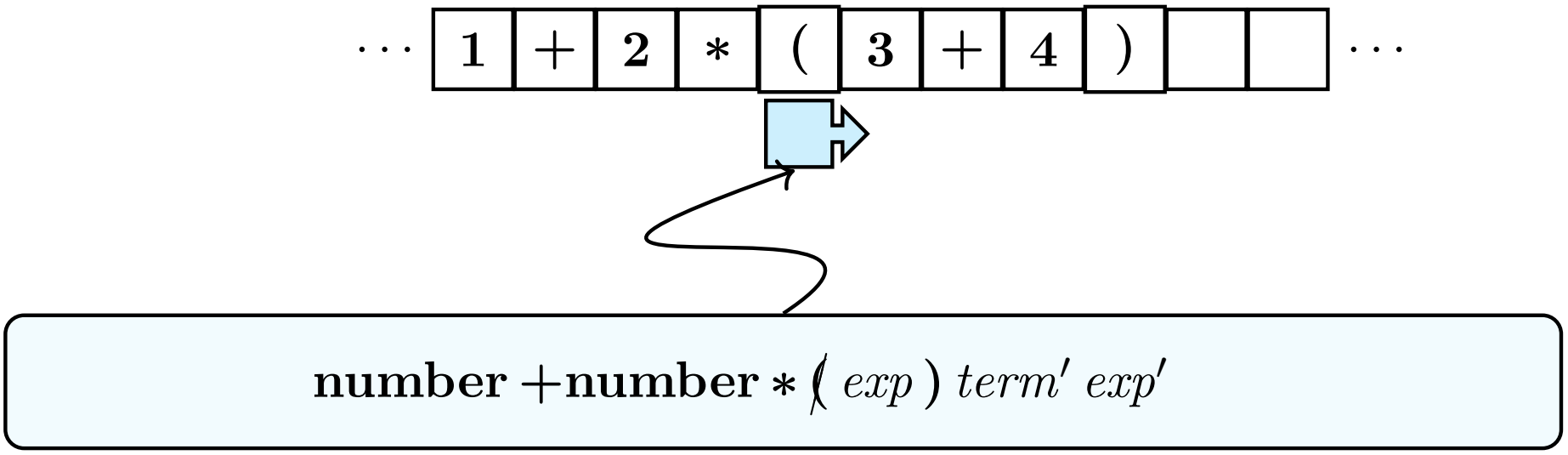
Derivation of an expression



factors and terms

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

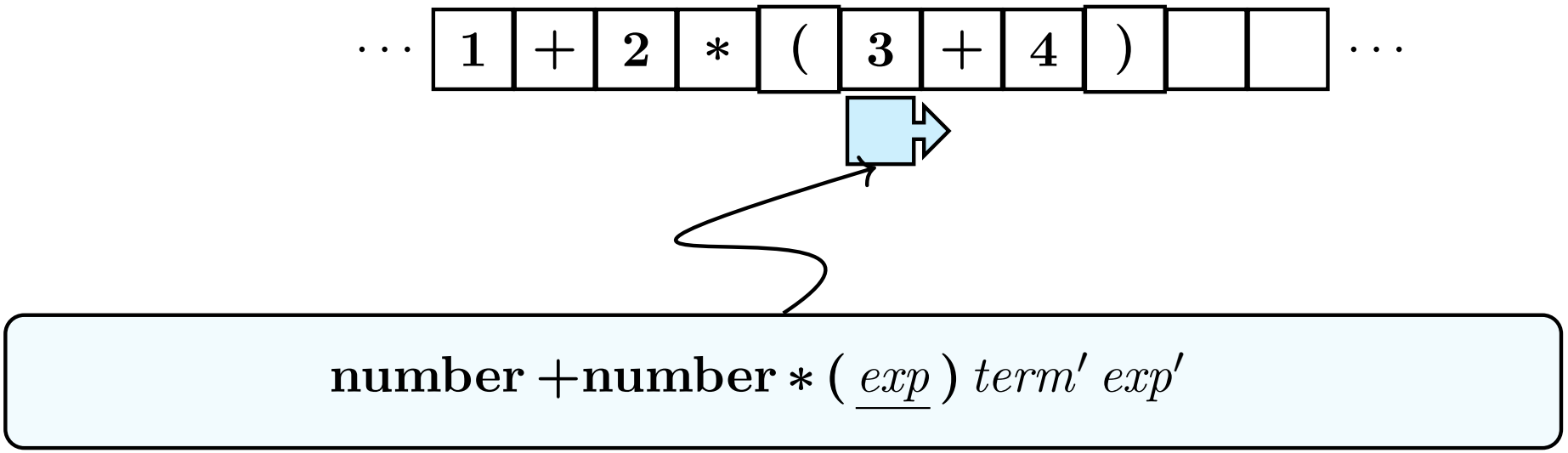
Derivation of an expression



factors and terms

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

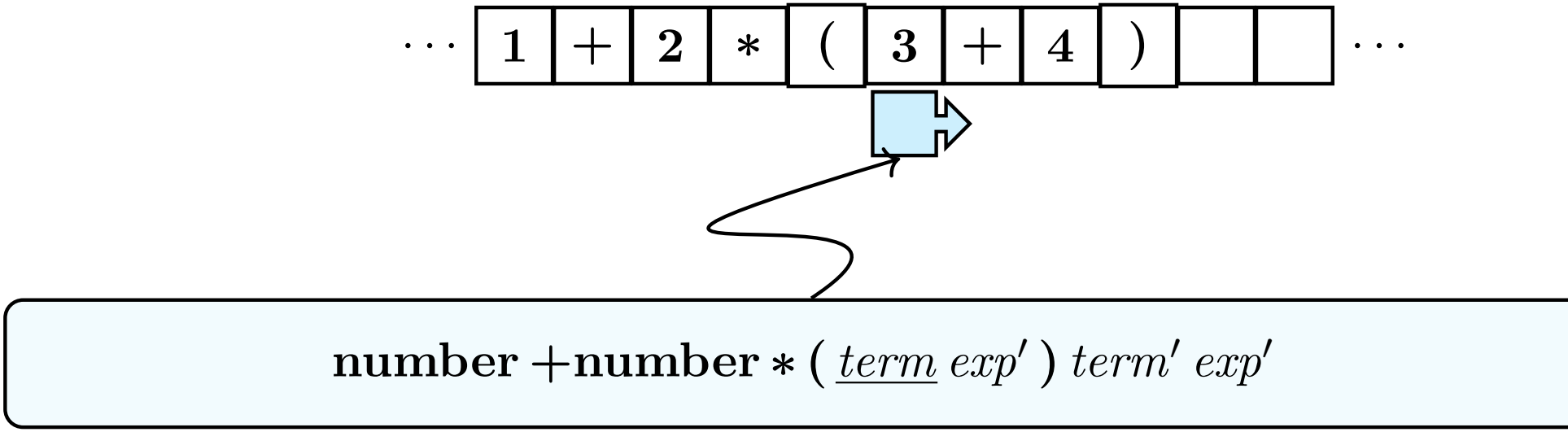
Derivation of an expression



factors and terms

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

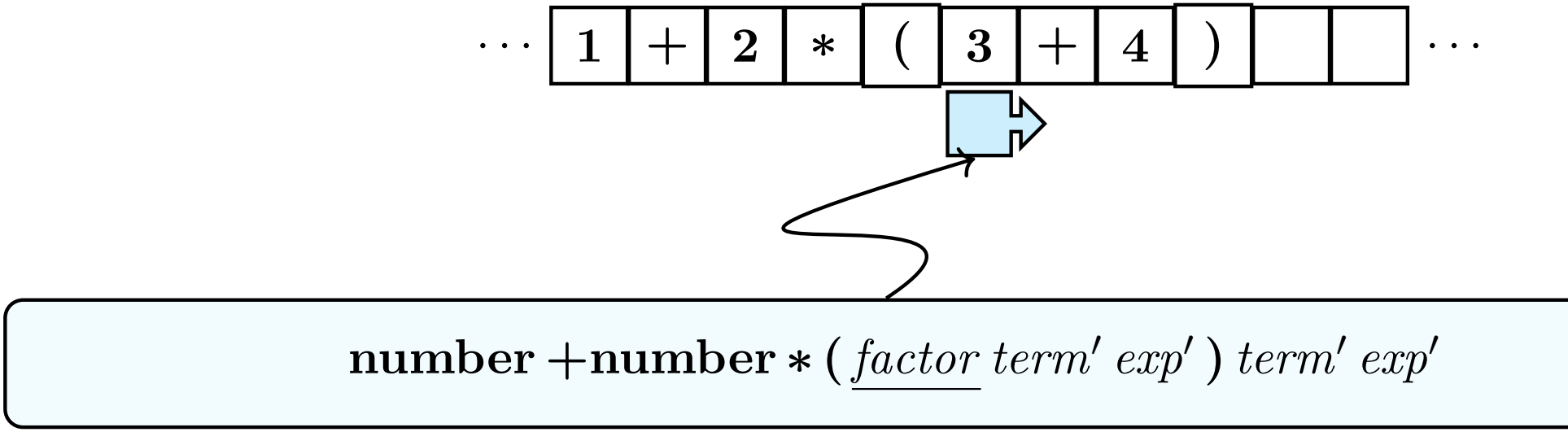
Derivation of an expression



factors and terms

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

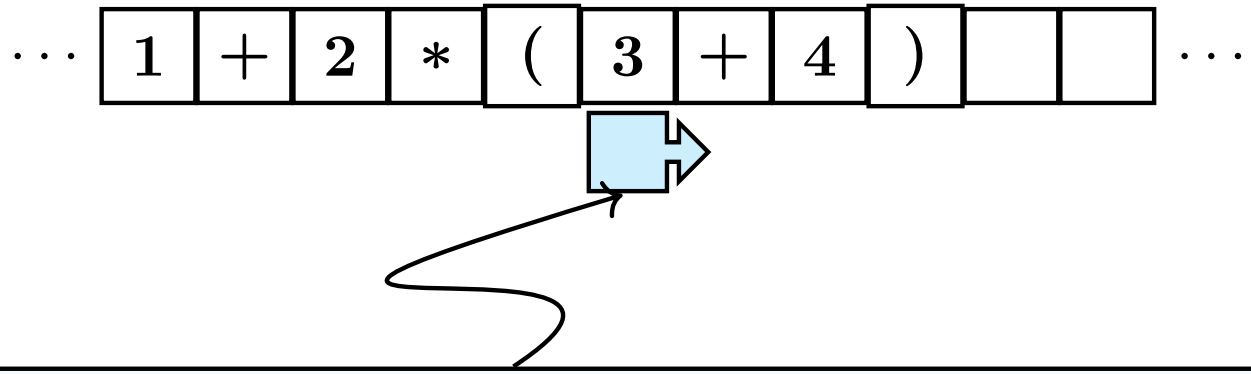
Derivation of an expression



factors and terms

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

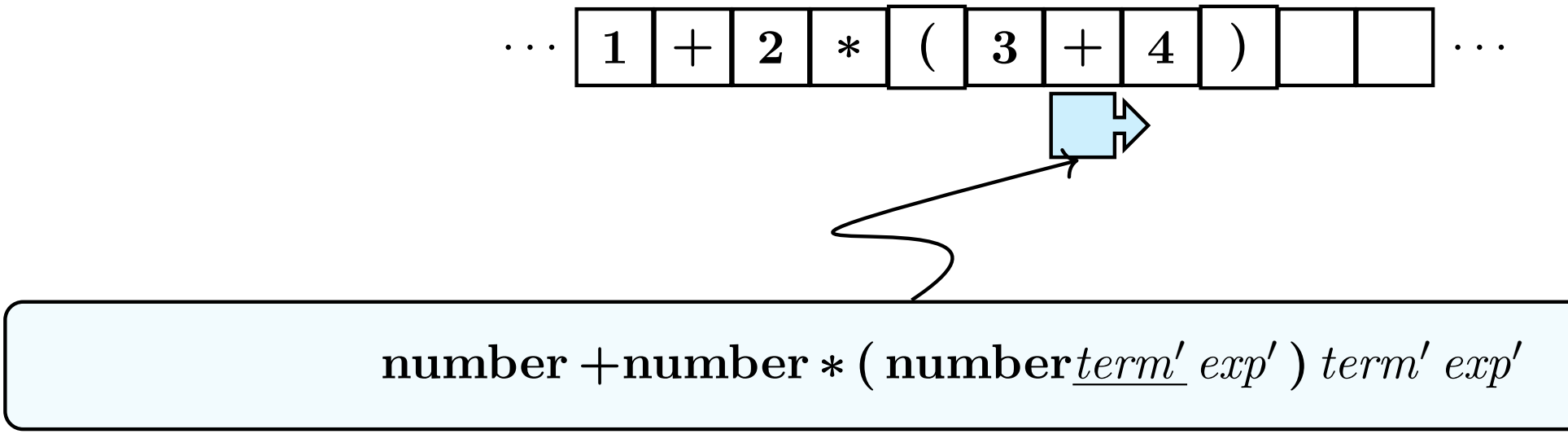
Derivation of an expression



factors and terms

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

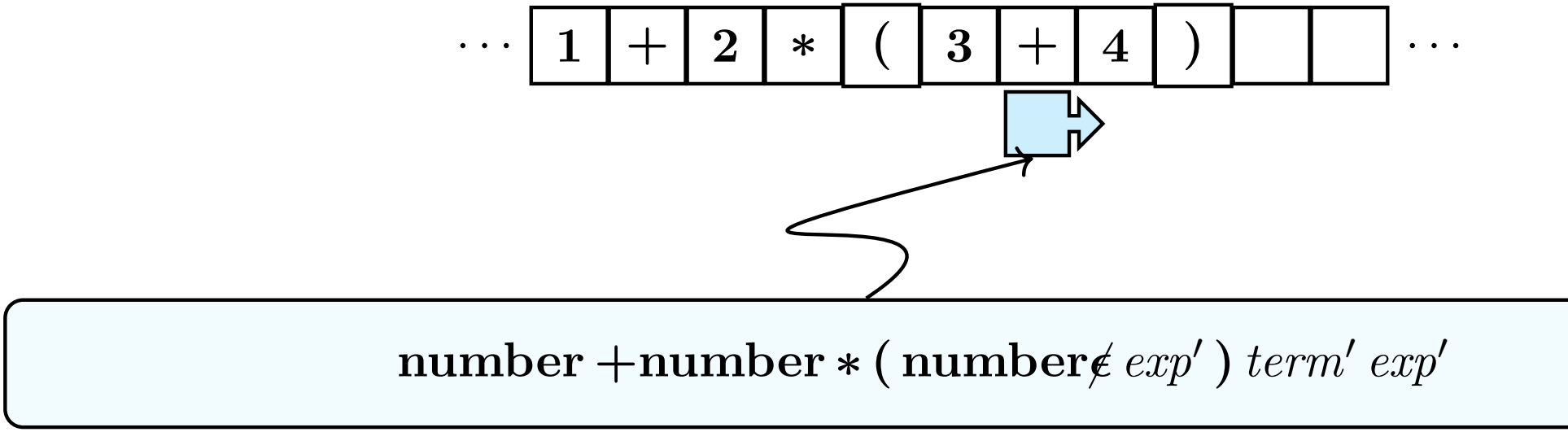
Derivation of an expression



factors and terms

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

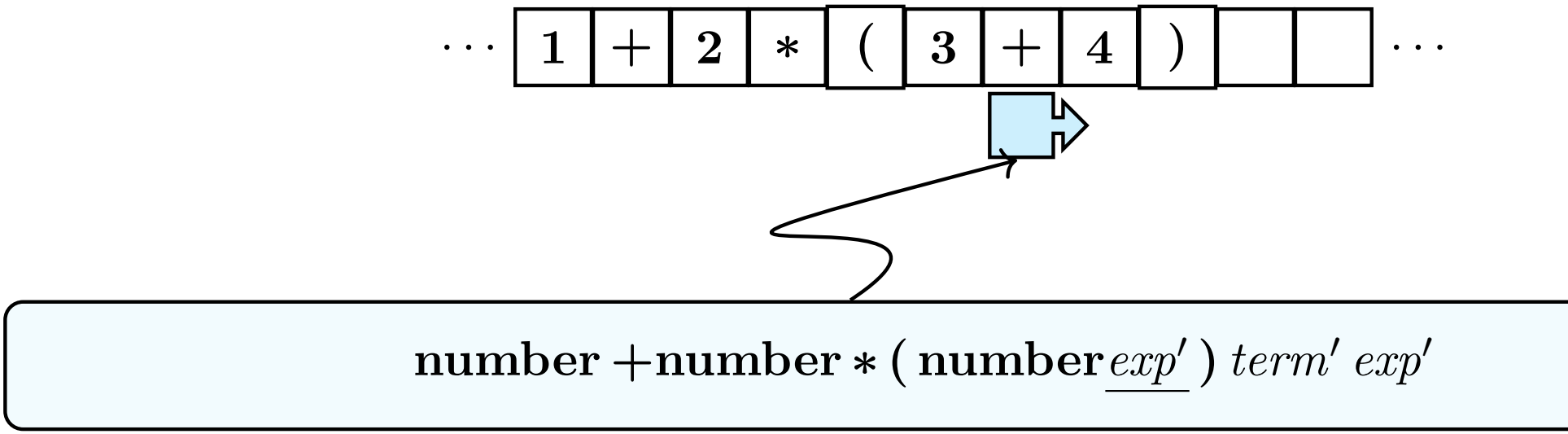
Derivation of an expression



factors and terms

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

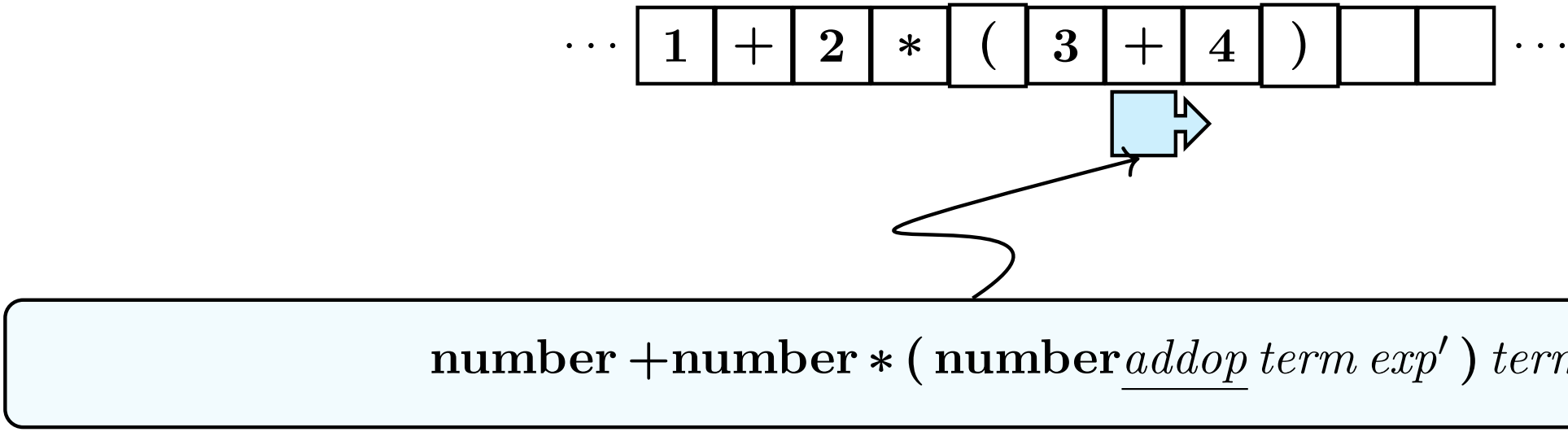
Derivation of an expression



factors and terms

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

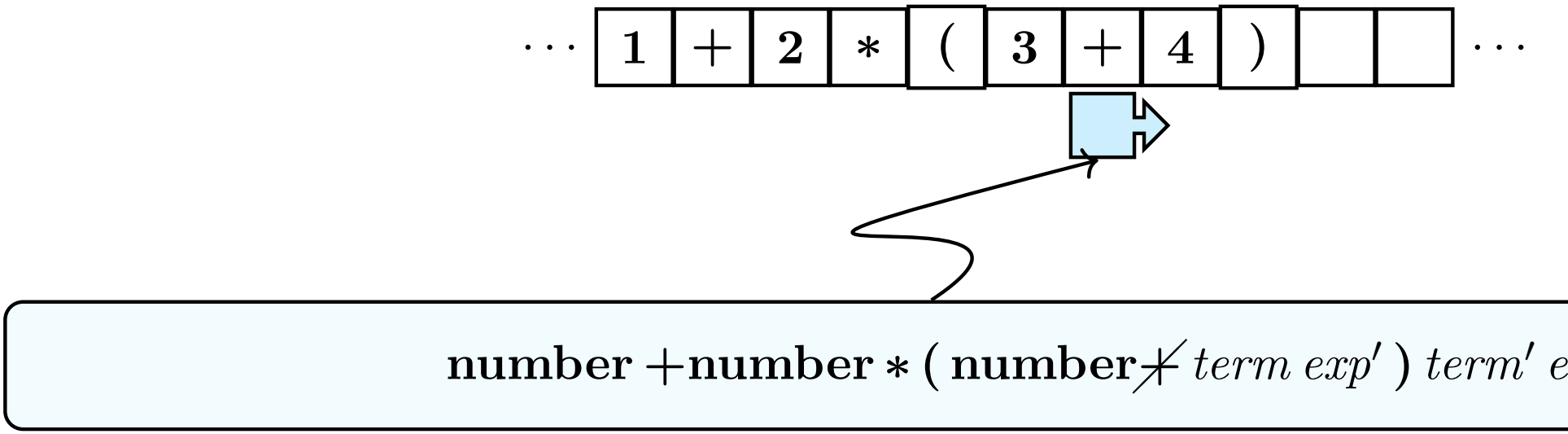
Derivation of an expression



factors and terms

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

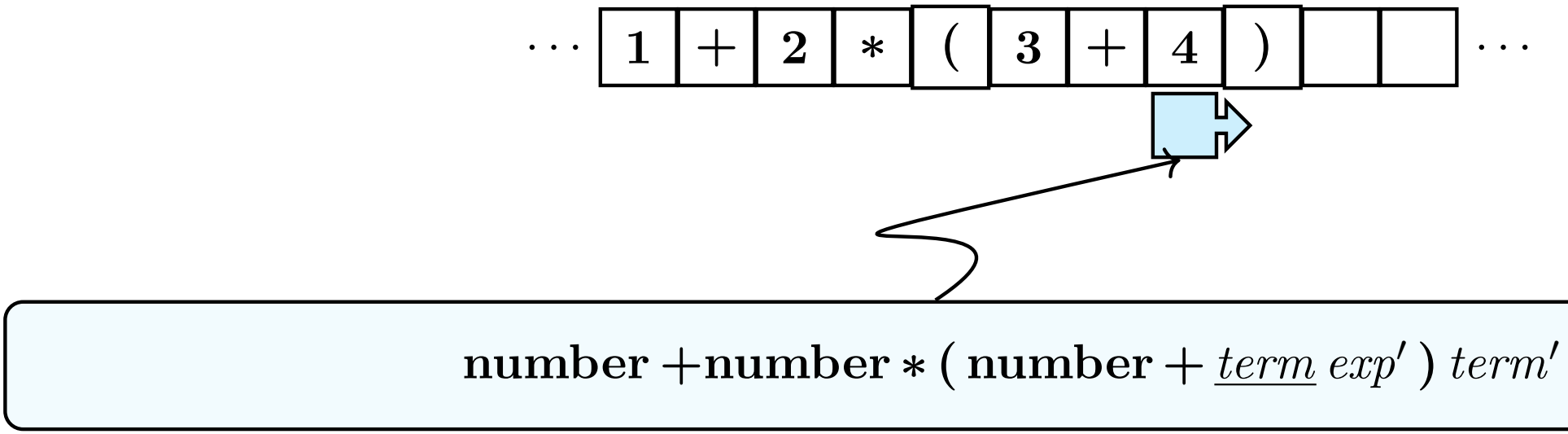
Derivation of an expression



factors and terms

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

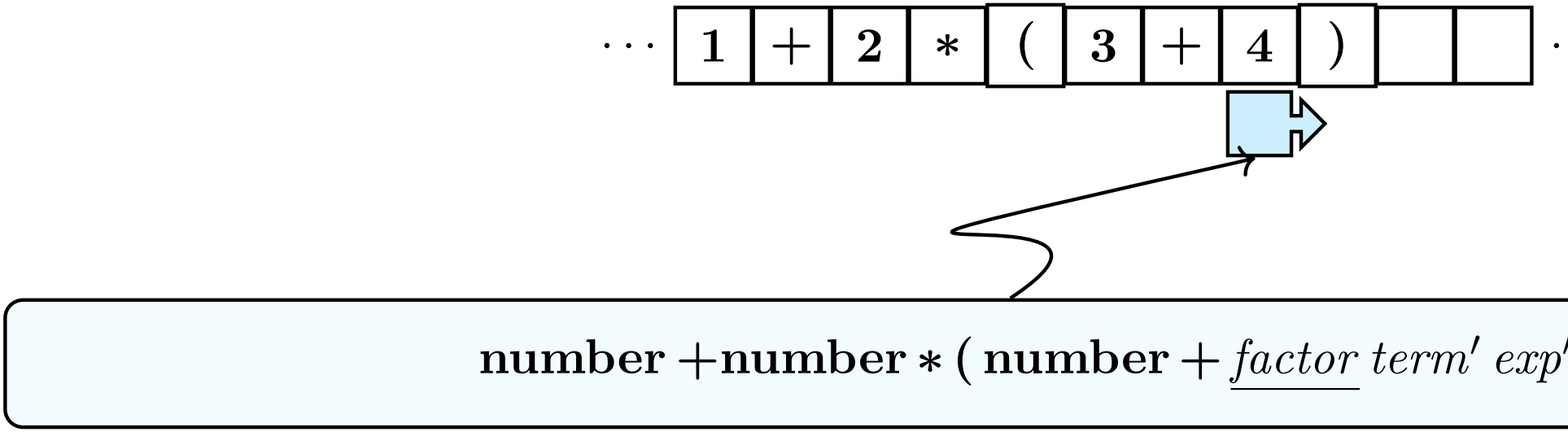
Derivation of an expression



factors and terms

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

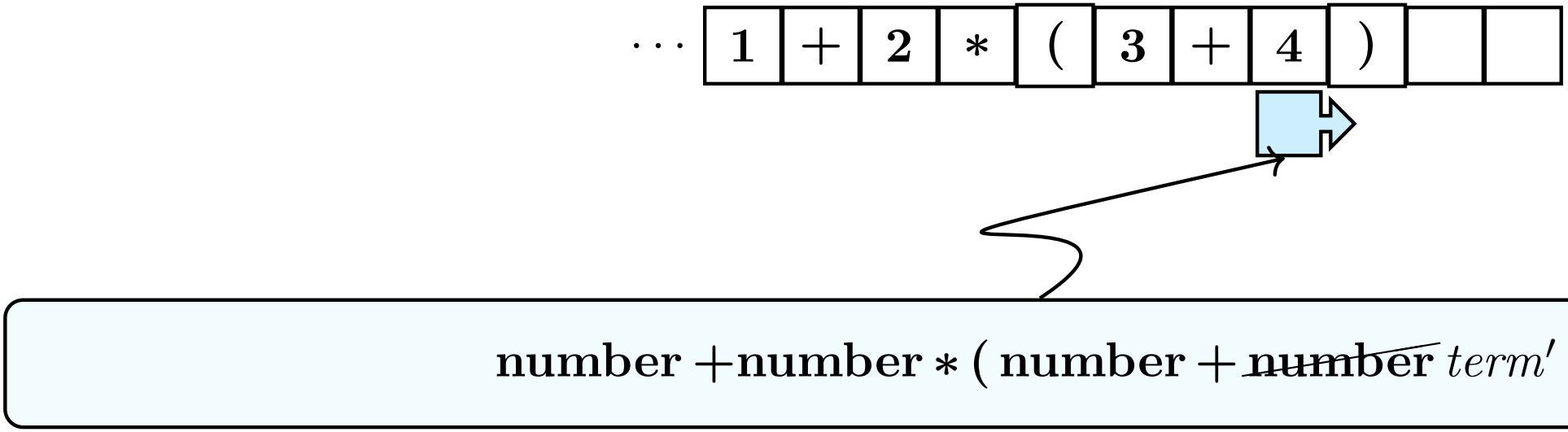
Derivation of an expression



factors and terms

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

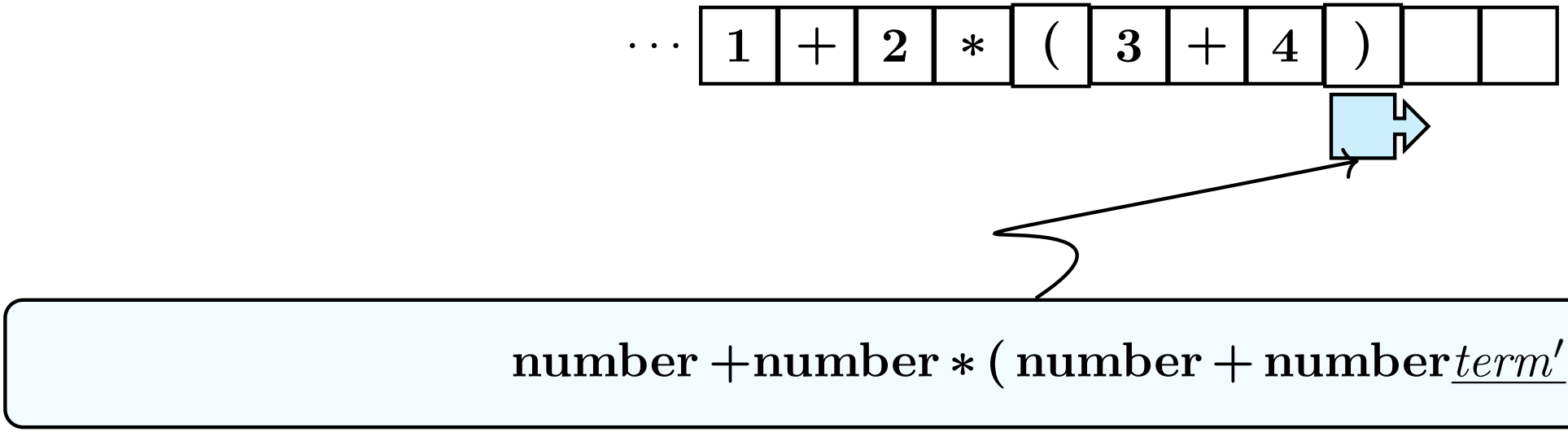
Derivation of an expression



factors and terms

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

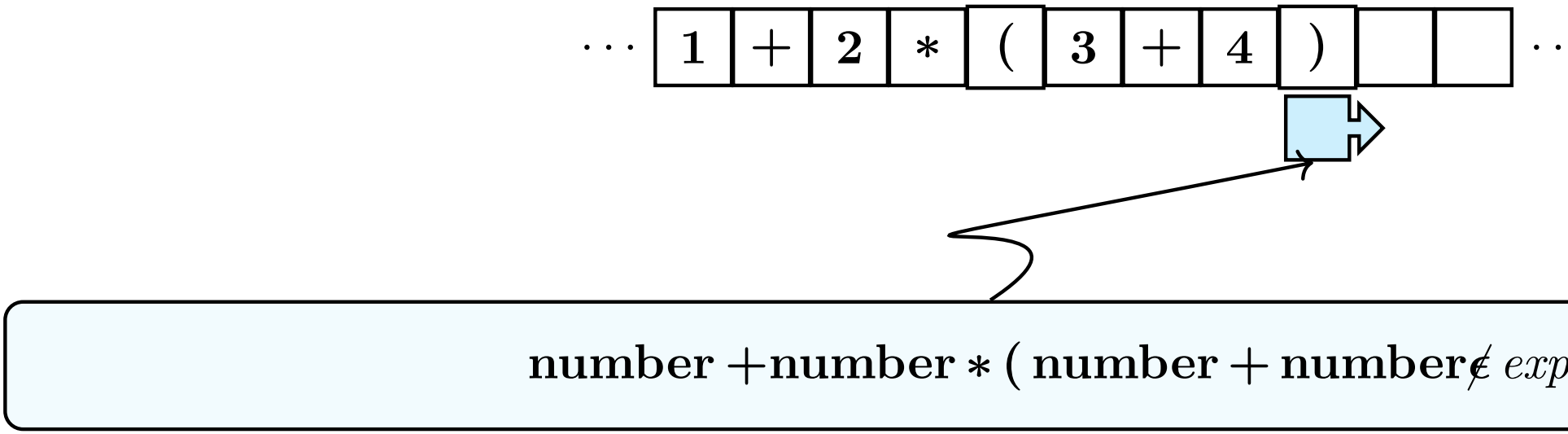
Derivation of an expression



factors and terms

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

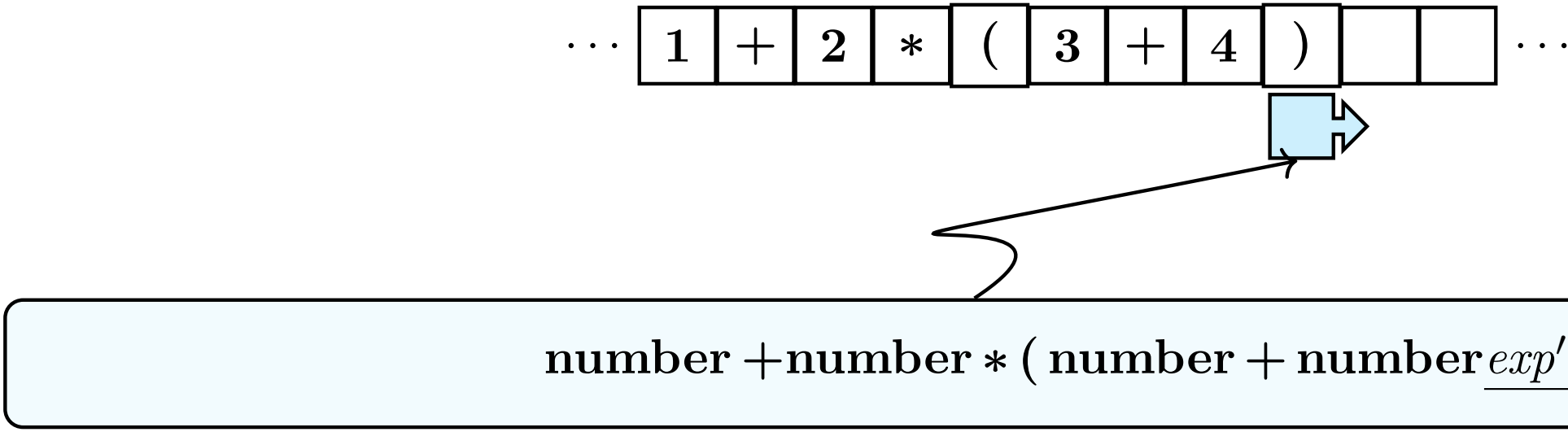
Derivation of an expression



factors and terms

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

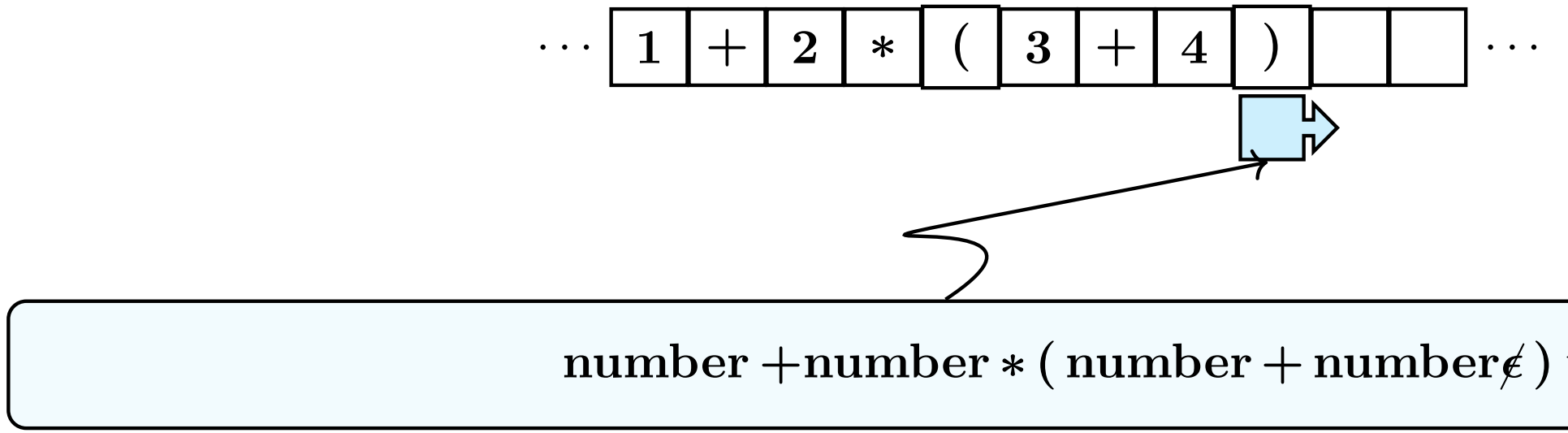
Derivation of an expression



factors and terms

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

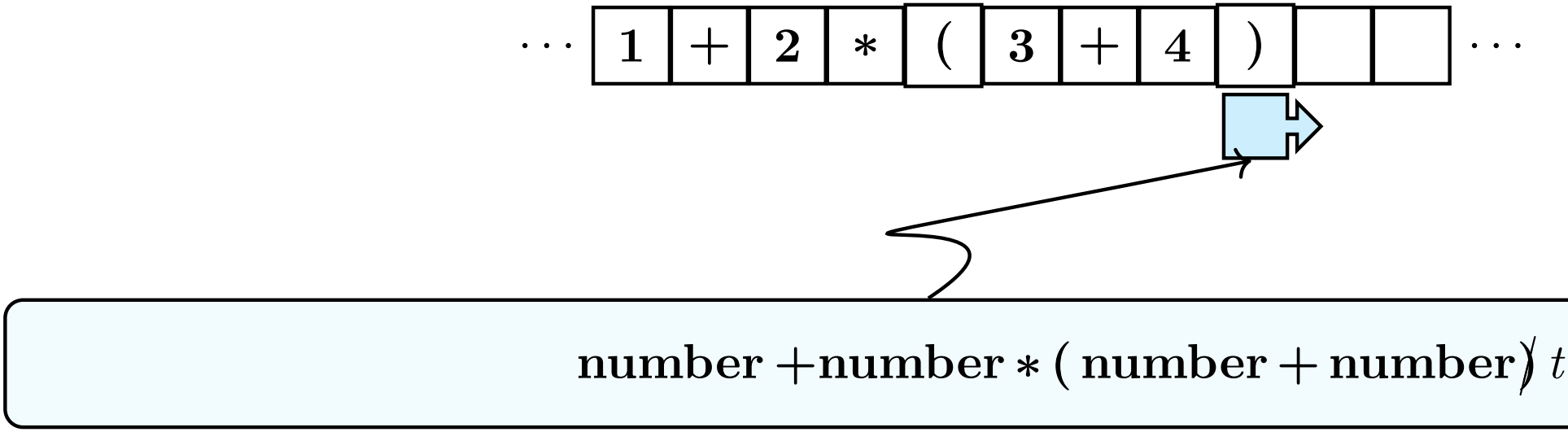
Derivation of an expression



factors and terms

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

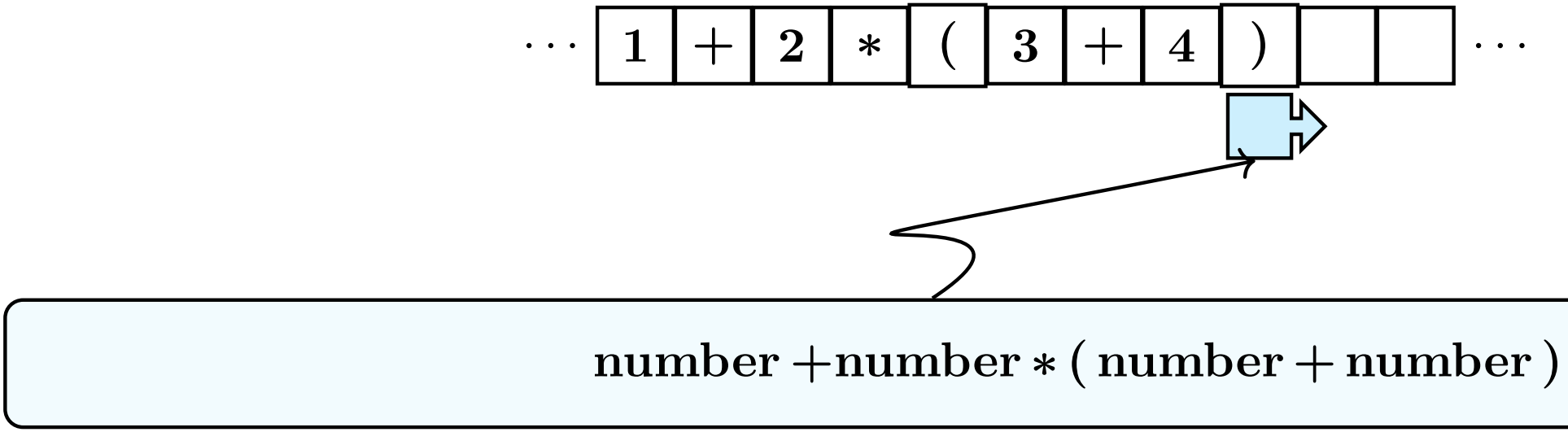
Derivation of an expression



factors and terms

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

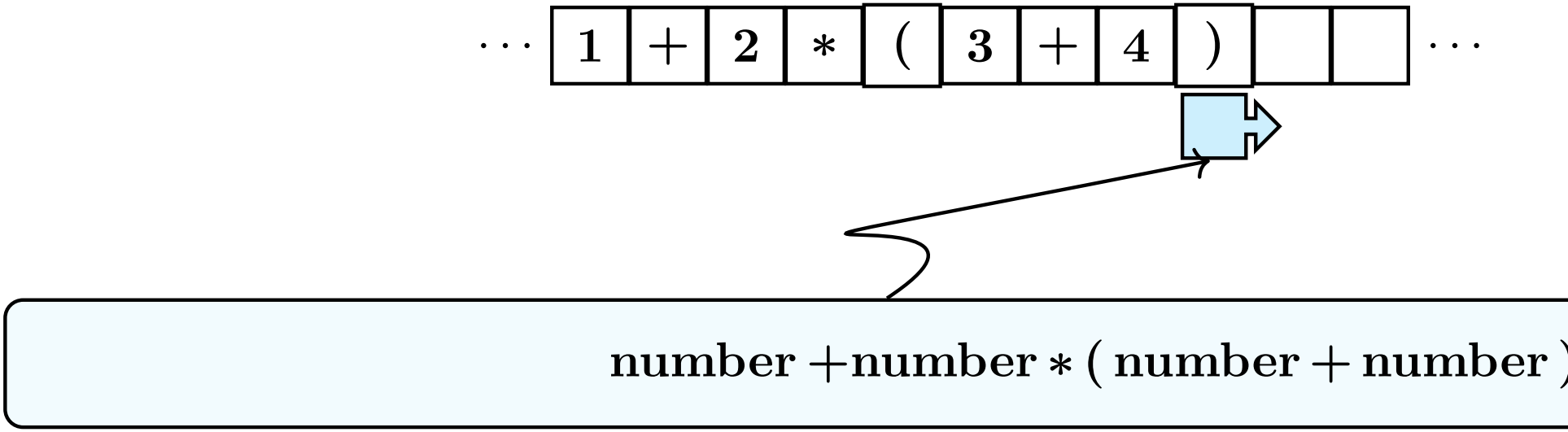
Derivation of an expression



factors and terms

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

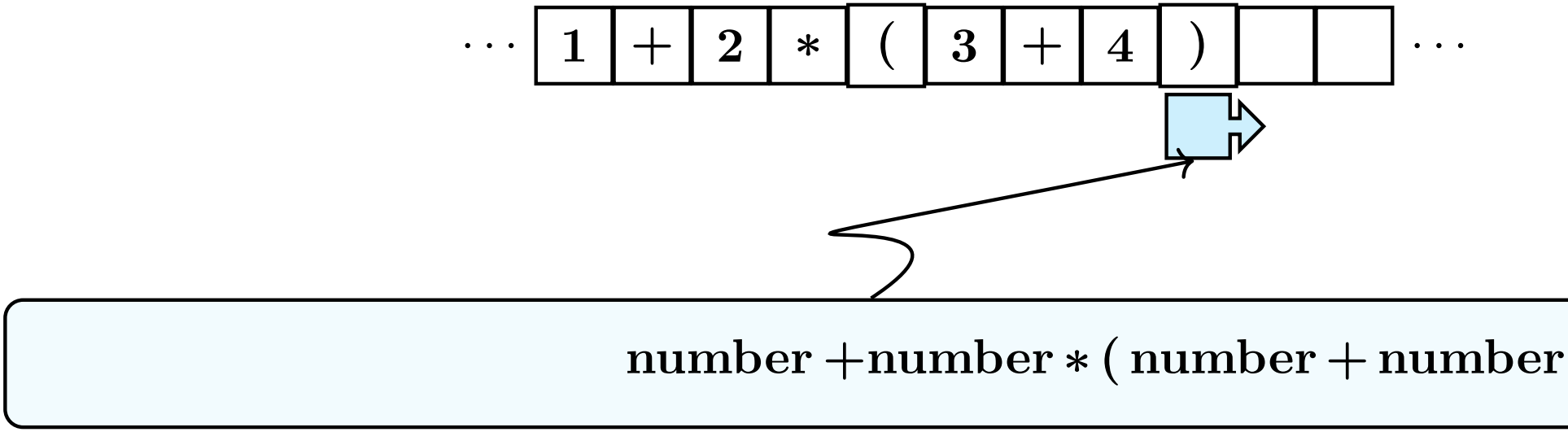
Derivation of an expression



factors and terms

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

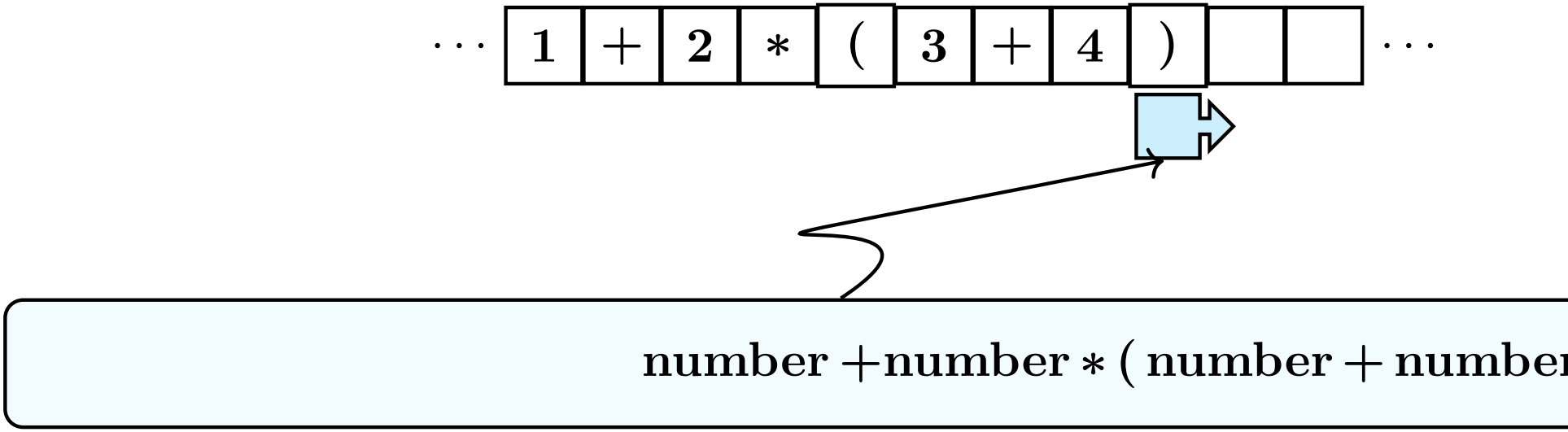
Derivation of an expression



factors and terms

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

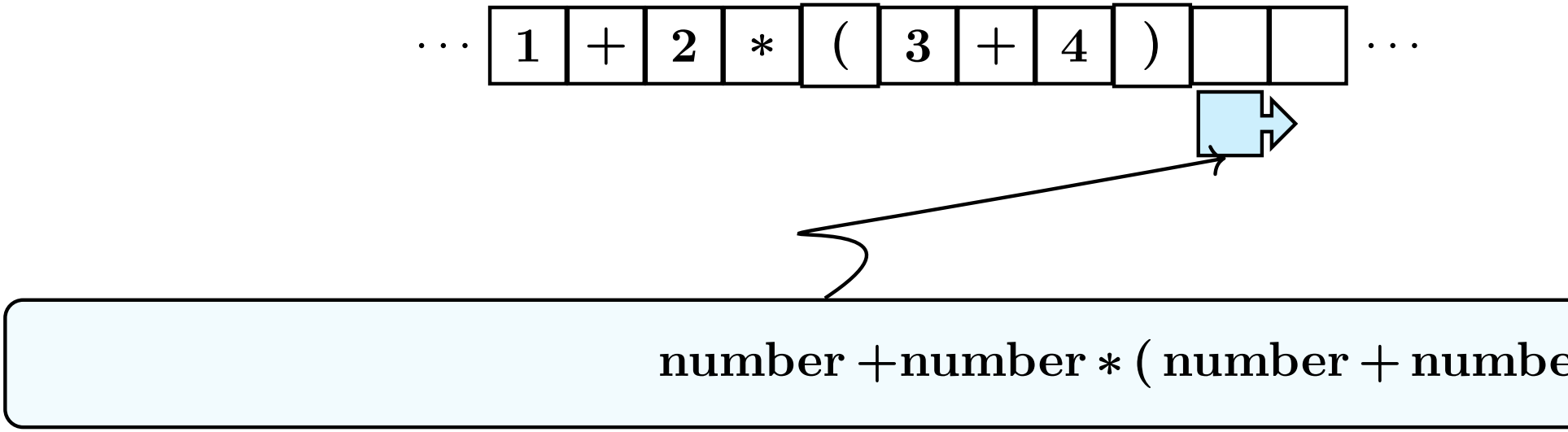
Derivation of an expression



factors and terms

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

Derivation of an expression



factors and terms

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

Remarks concerning the derivation

Note:

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

Notation:

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Not as a "film" but at a glance: reduction sequence

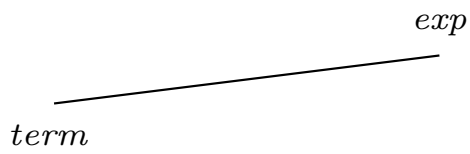
$$
\begin{aligned}
&\underline{exp} && \Rightarrow \\
&\underline{term}\ exp' && \Rightarrow \\
&\underline{factor}\ term'\ exp' && \Rightarrow \\
&\cancel{\mathbf{number}}\ term'\ exp' && \Rightarrow \\
&\mathbf{number}\underline{term'}\ exp' && \Rightarrow \\
&\mathbf{number}\cancel{\epsilon}\ exp' && \Rightarrow \\
&\mathbf{number}\underline{exp'} && \Rightarrow \\
&\mathbf{number}\underline{addop}\ term\ exp' && \Rightarrow \\
&\mathbf{number}\cancel{+}\ term\ exp' && \Rightarrow \\
&\mathbf{number}\ +\underline{term}\ exp' && \Rightarrow \\
&\mathbf{number}\ +\underline{factor}\ term'\ exp' && \Rightarrow \\
&\mathbf{number}\ +\cancel{\mathbf{number}}\ term'\ exp' && \Rightarrow \\
&\mathbf{number}\ +\mathbf{number}\underline{term'}\ exp' && \Rightarrow \\
&\mathbf{number}\ +\mathbf{number}\underline{mulop}\ factor\ term'\ exp' && \Rightarrow \\
&\mathbf{number}\ +\mathbf{number}\cancel{*}\ \underline{factor}\ term'\ exp' && \Rightarrow \\
&\mathbf{number}\ +\mathbf{number}\ *\ (\ exp\ )\ term'\ exp' && \Rightarrow \\
&\mathbf{number}\ +\mathbf{number}\ *\ \cancel{(}\ exp\ )\ term'\ exp' && \Rightarrow \\
&\mathbf{number}\ +\mathbf{number}\ *\ (\ \underline{exp}\ )\ term'\ exp' && \Rightarrow \\
&\ldots
\end{aligned}
$$



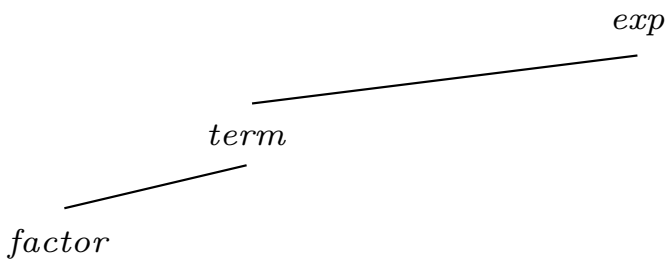
Best viewed as a tree

exp

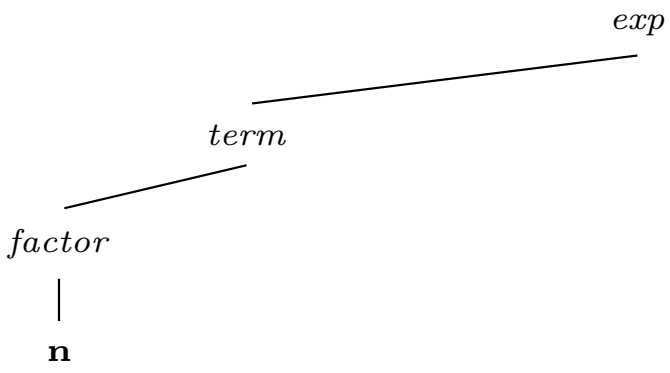
Best viewed as a tree



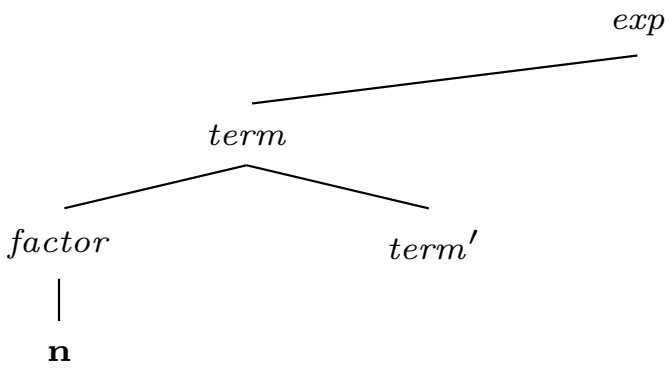
Best viewed as a tree



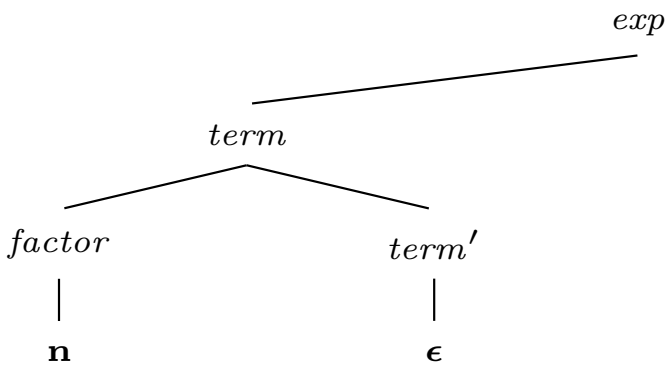
Best viewed as a tree



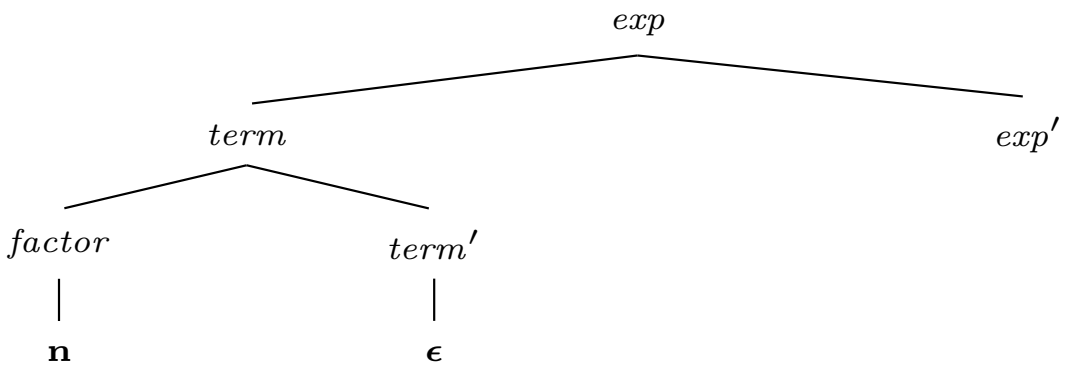
Best viewed as a tree



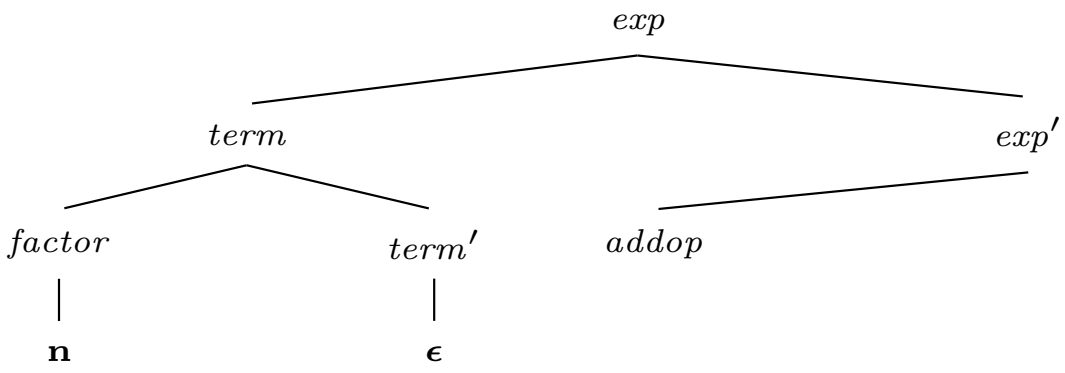
Best viewed as a tree



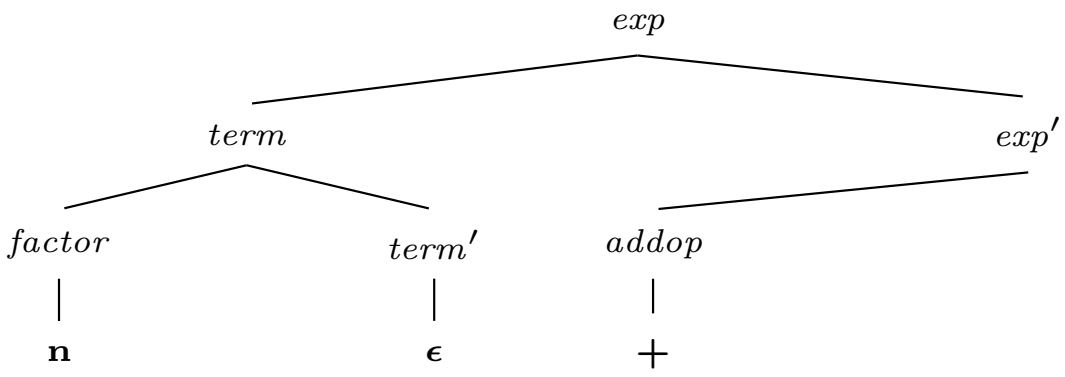
Best viewed as a tree



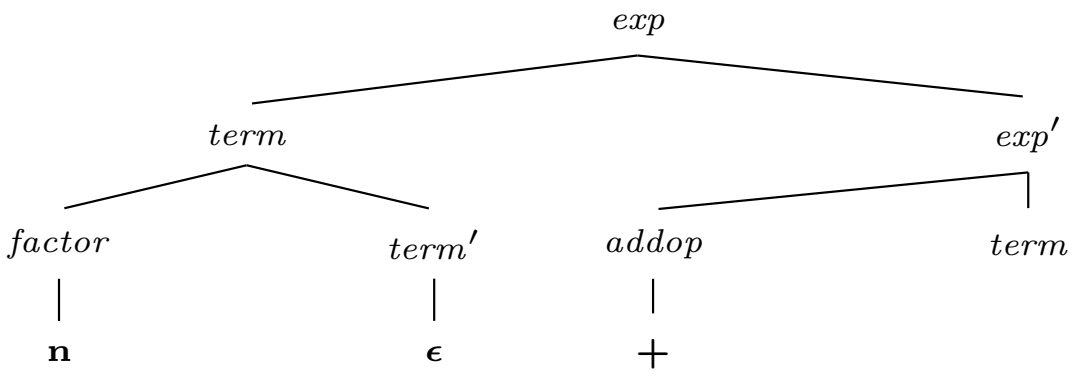
Best viewed as a tree



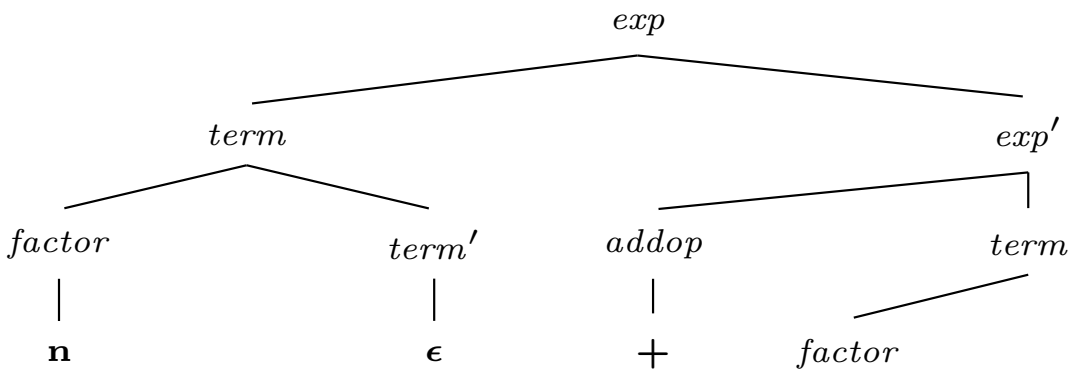
Best viewed as a tree



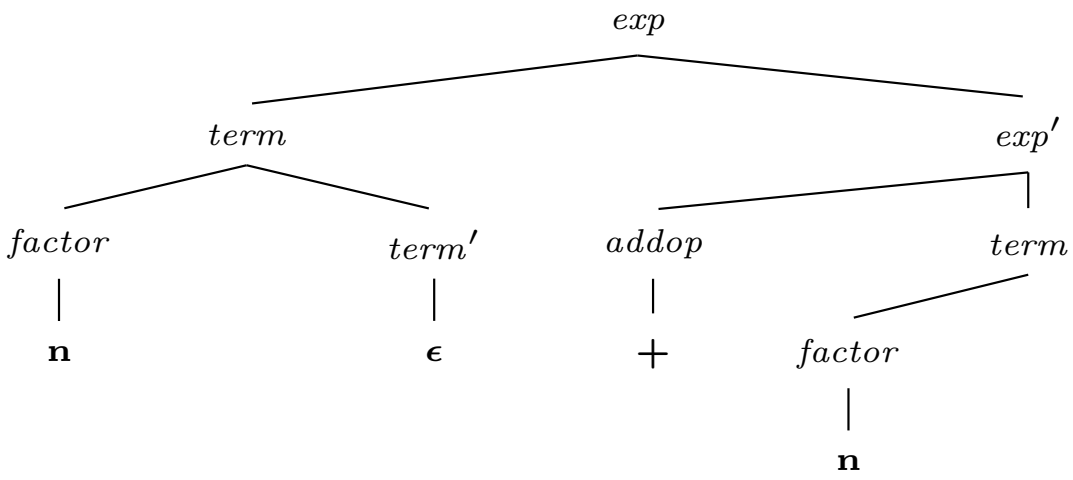
Best viewed as a tree



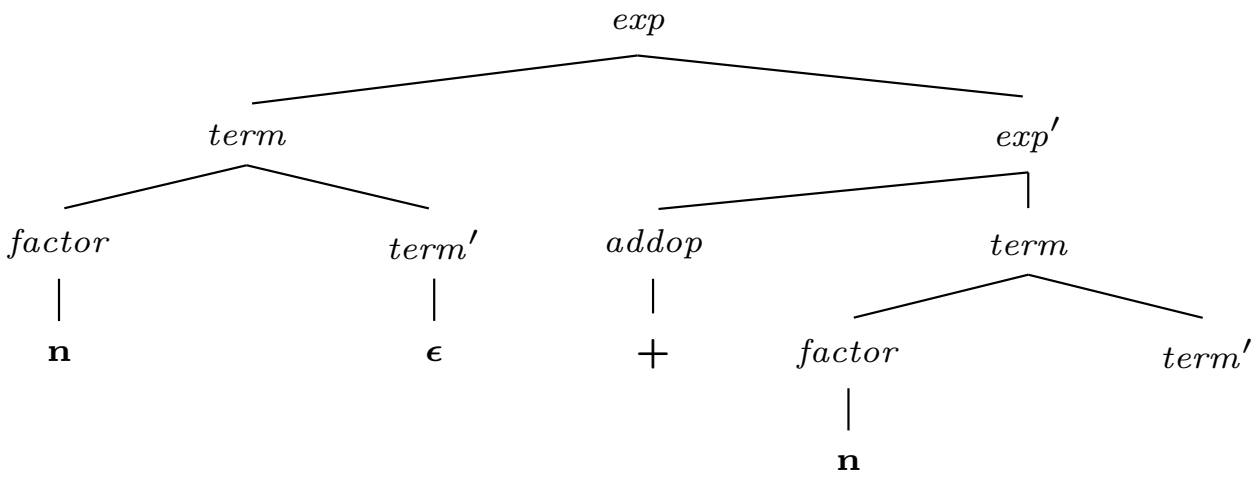
Best viewed as a tree



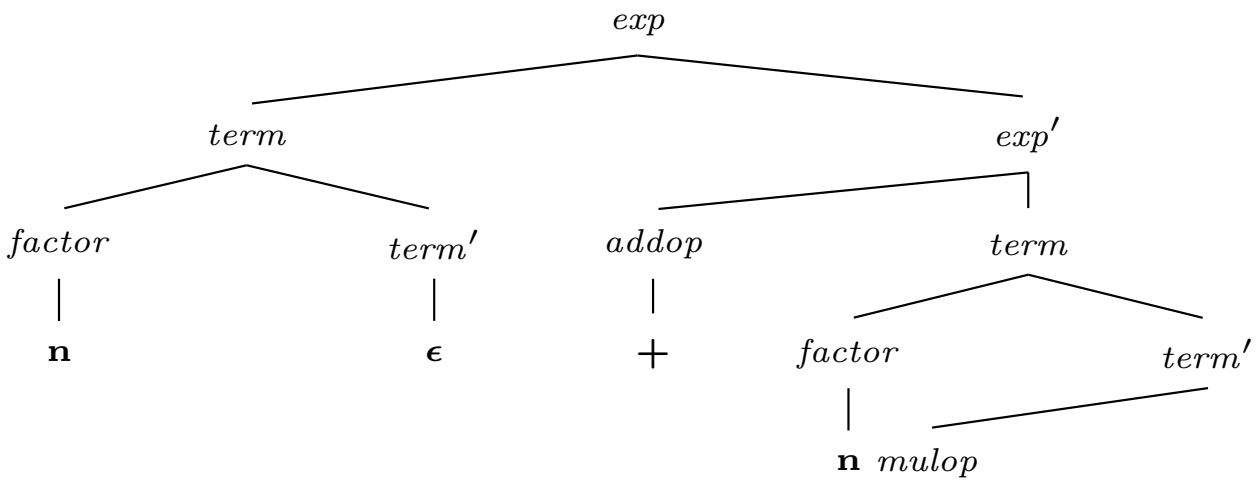
Best viewed as a tree



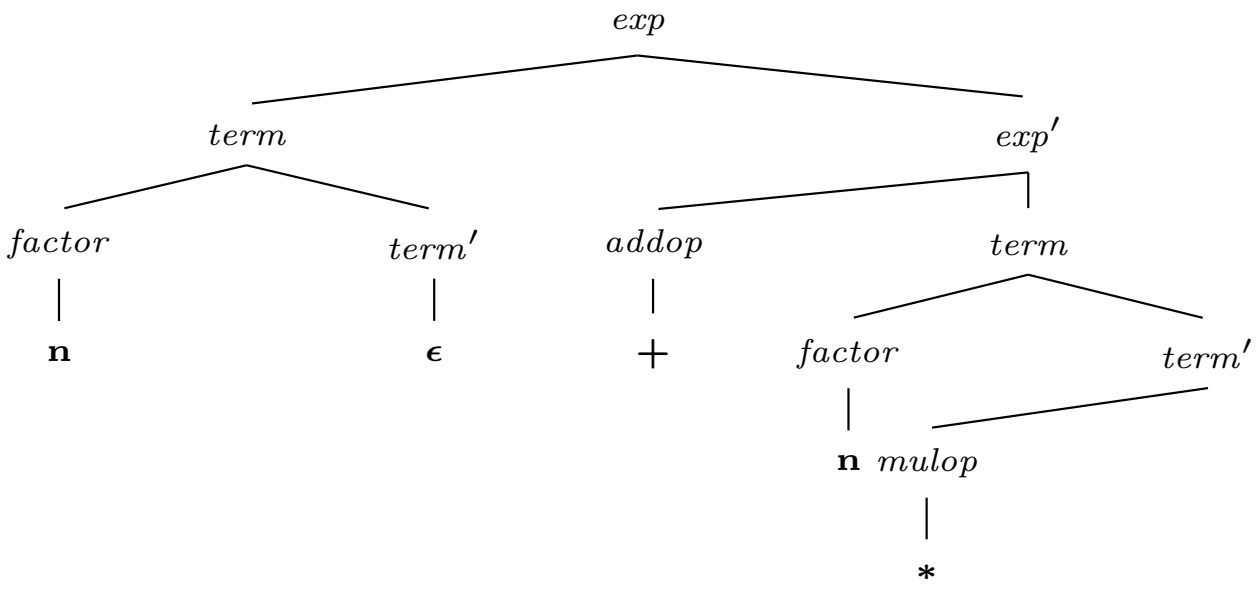
Best viewed as a tree



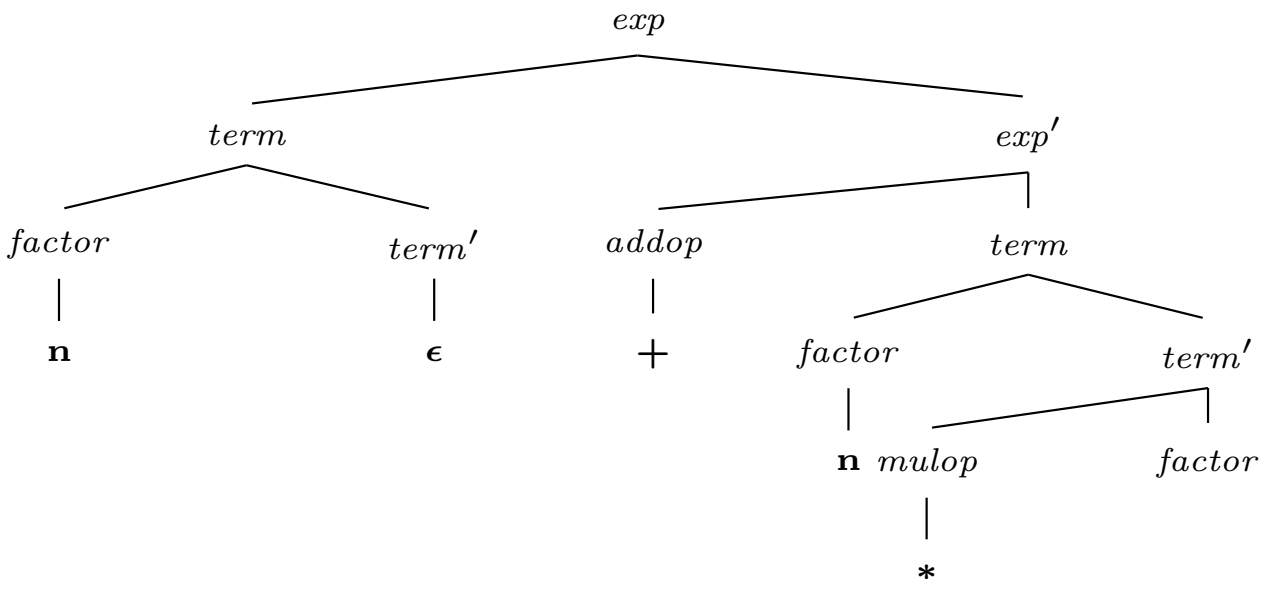
Best viewed as a tree



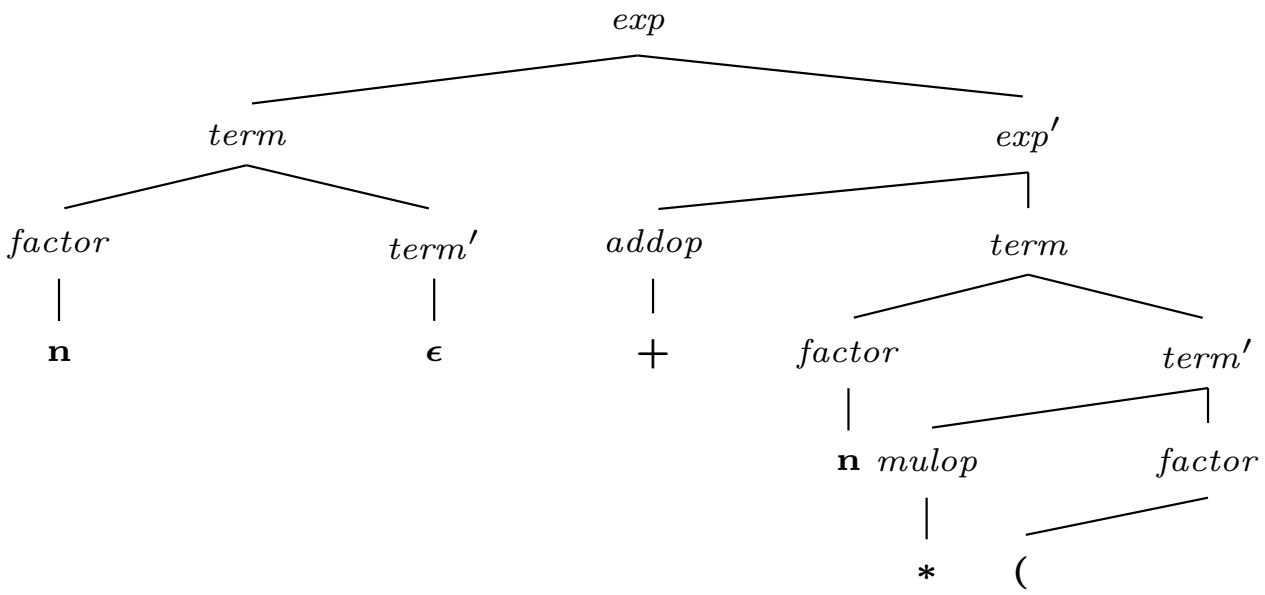
Best viewed as a tree



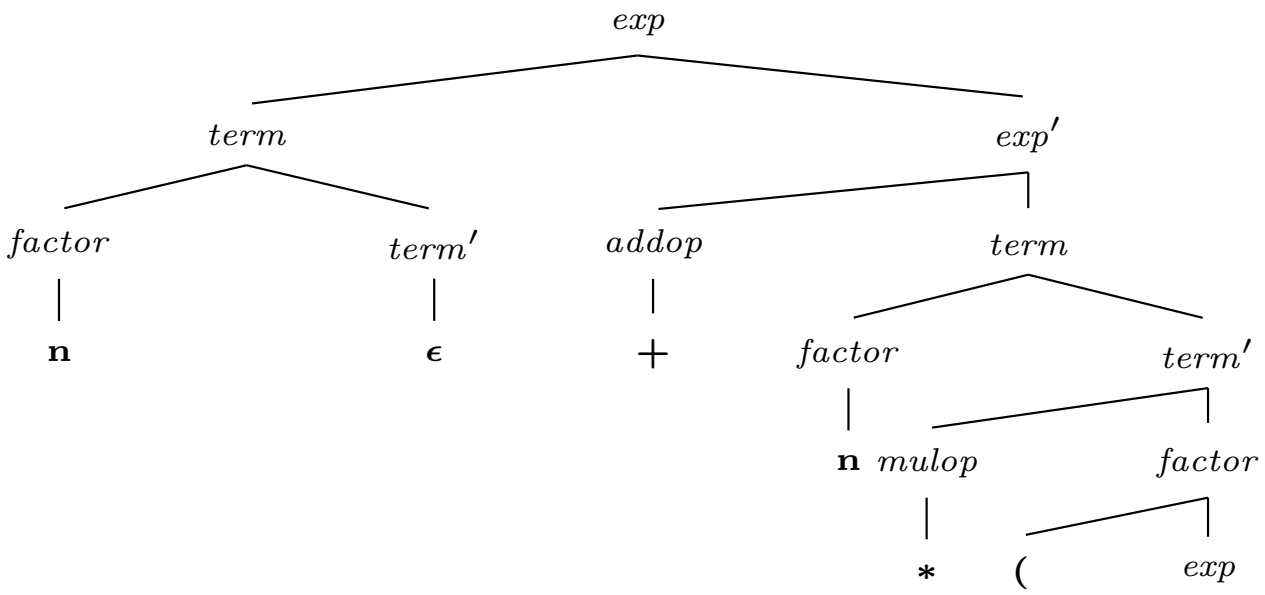
Best viewed as a tree



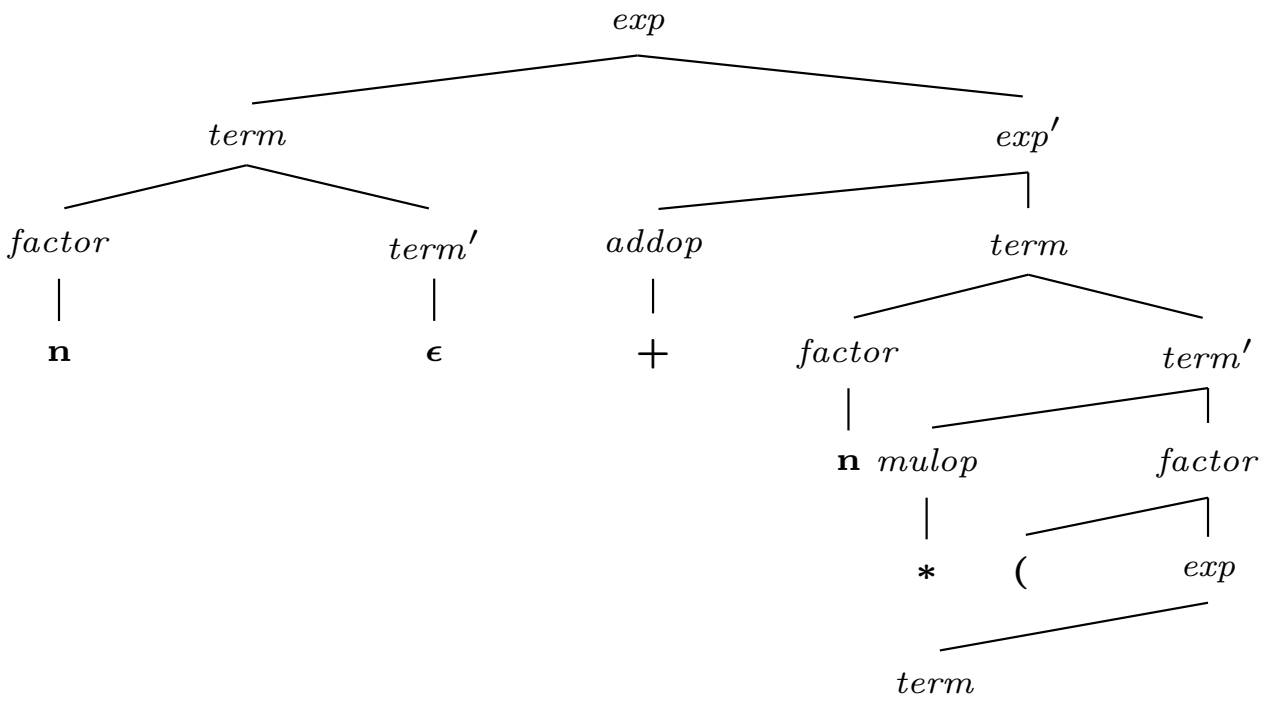
Best viewed as a tree



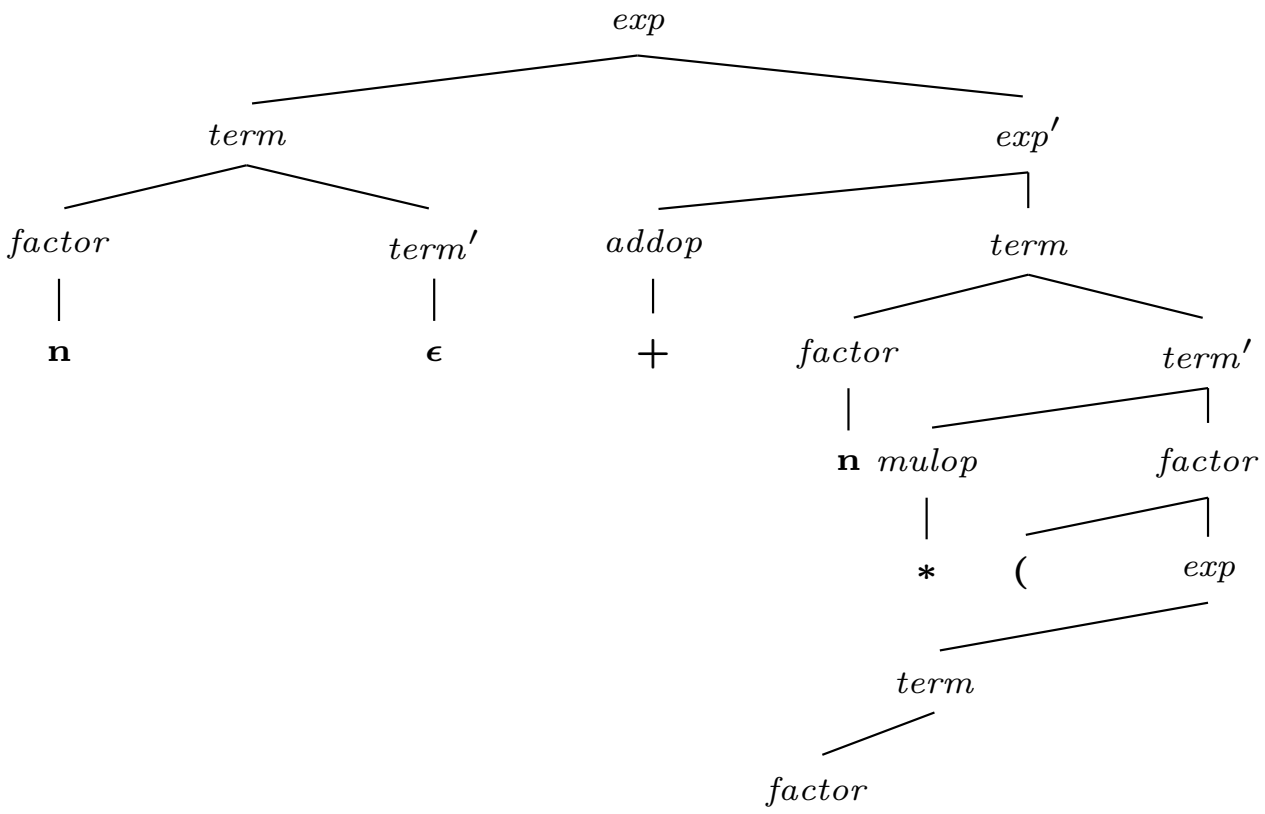
Best viewed as a tree



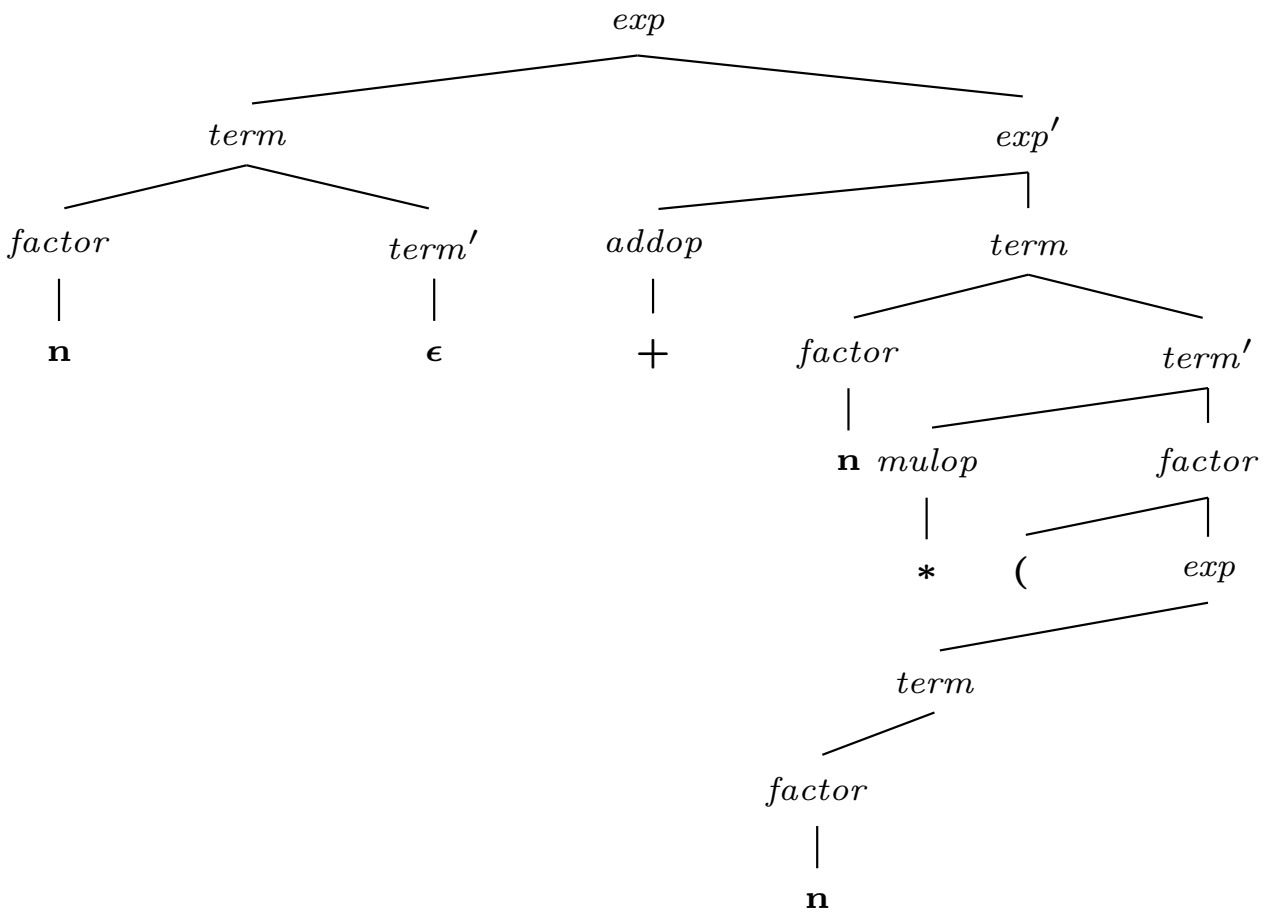
Best viewed as a tree



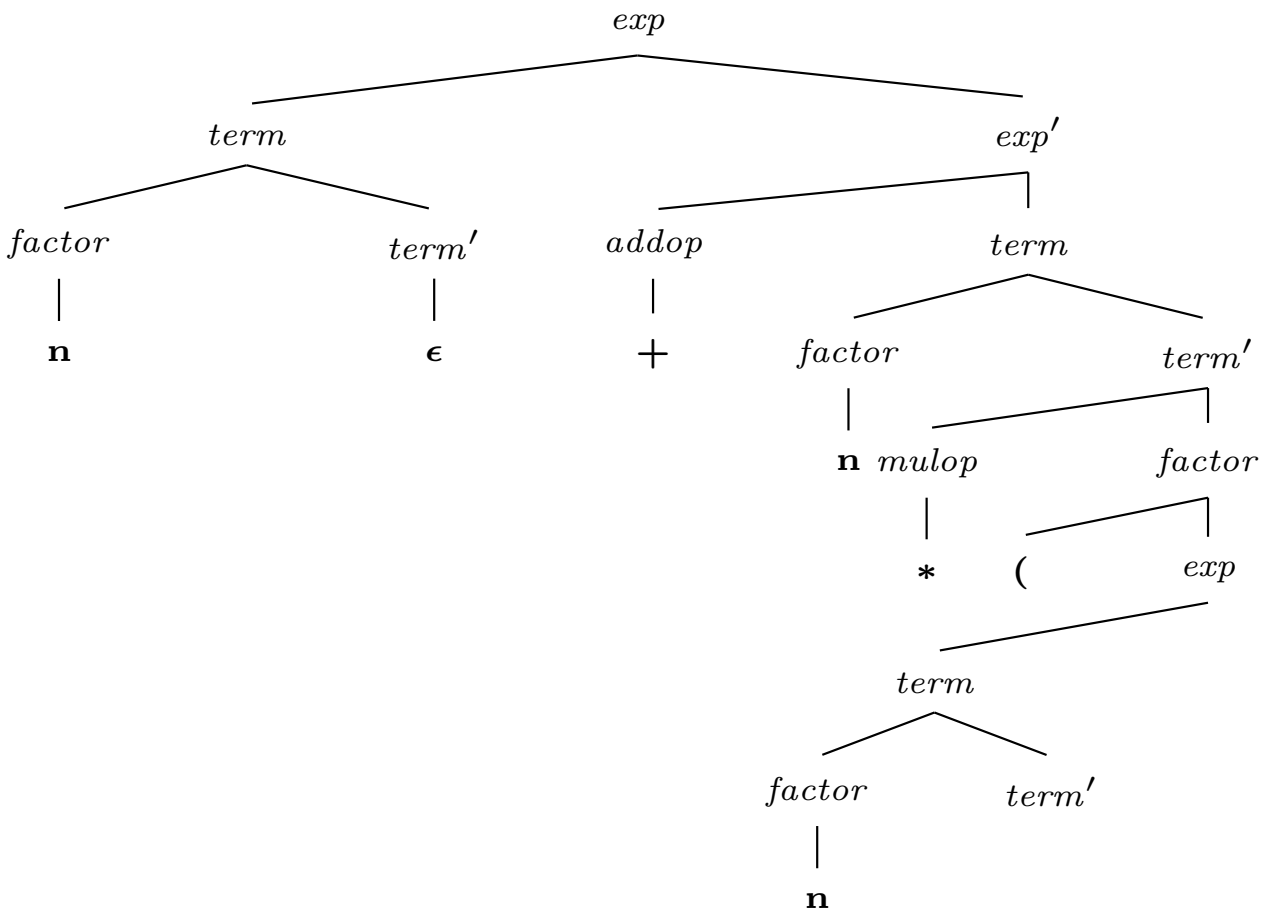
Best viewed as a tree



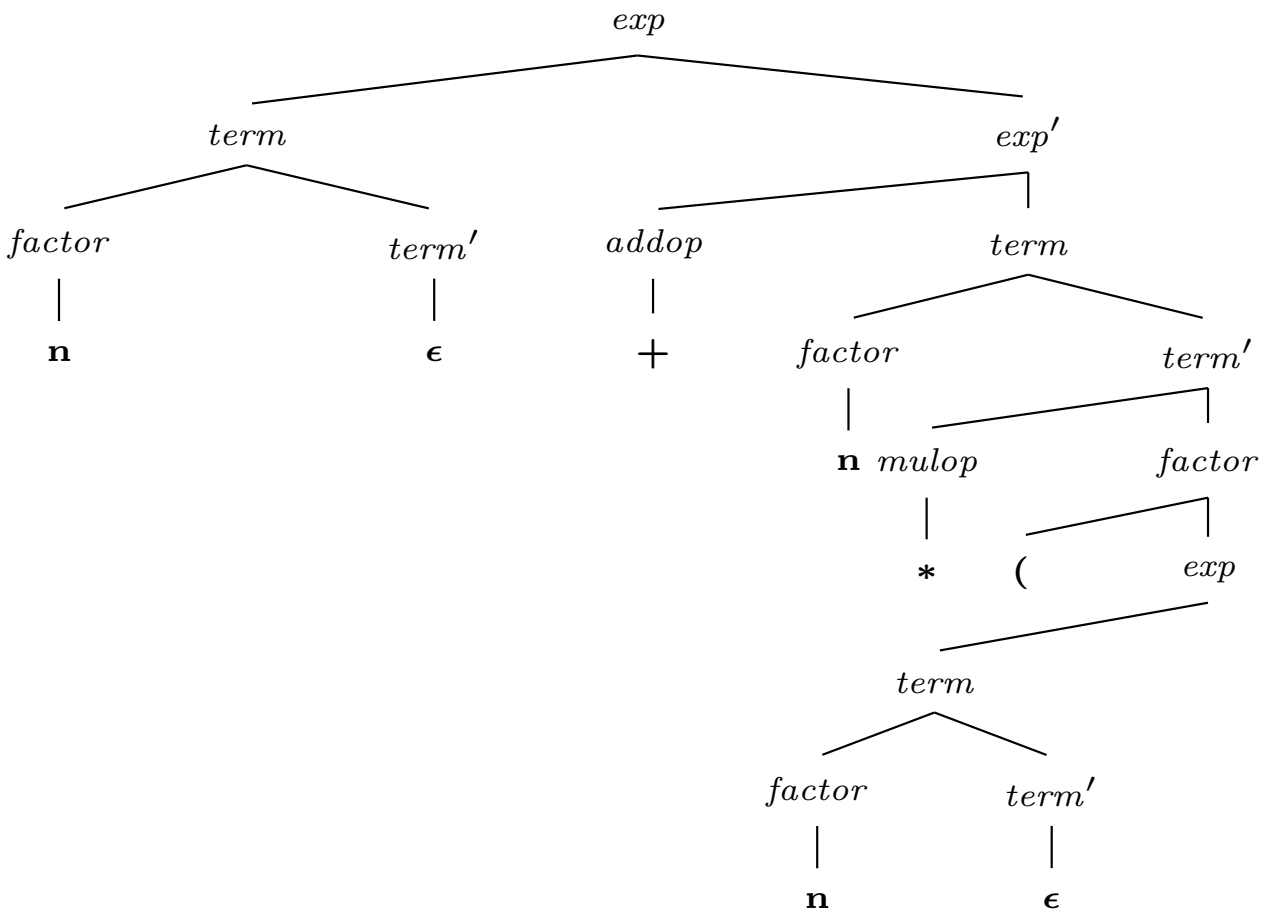
Best viewed as a tree



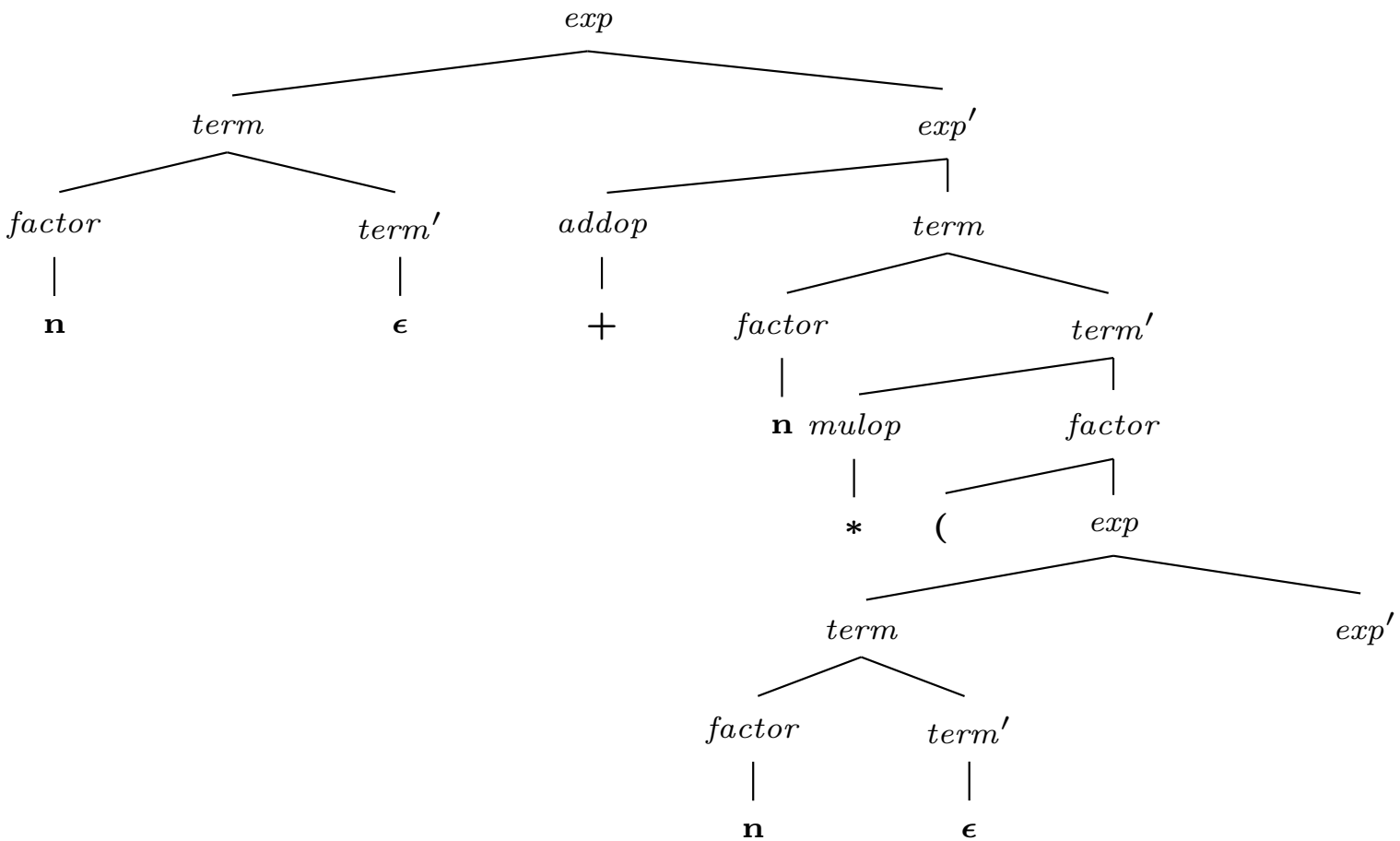
Best viewed as a tree



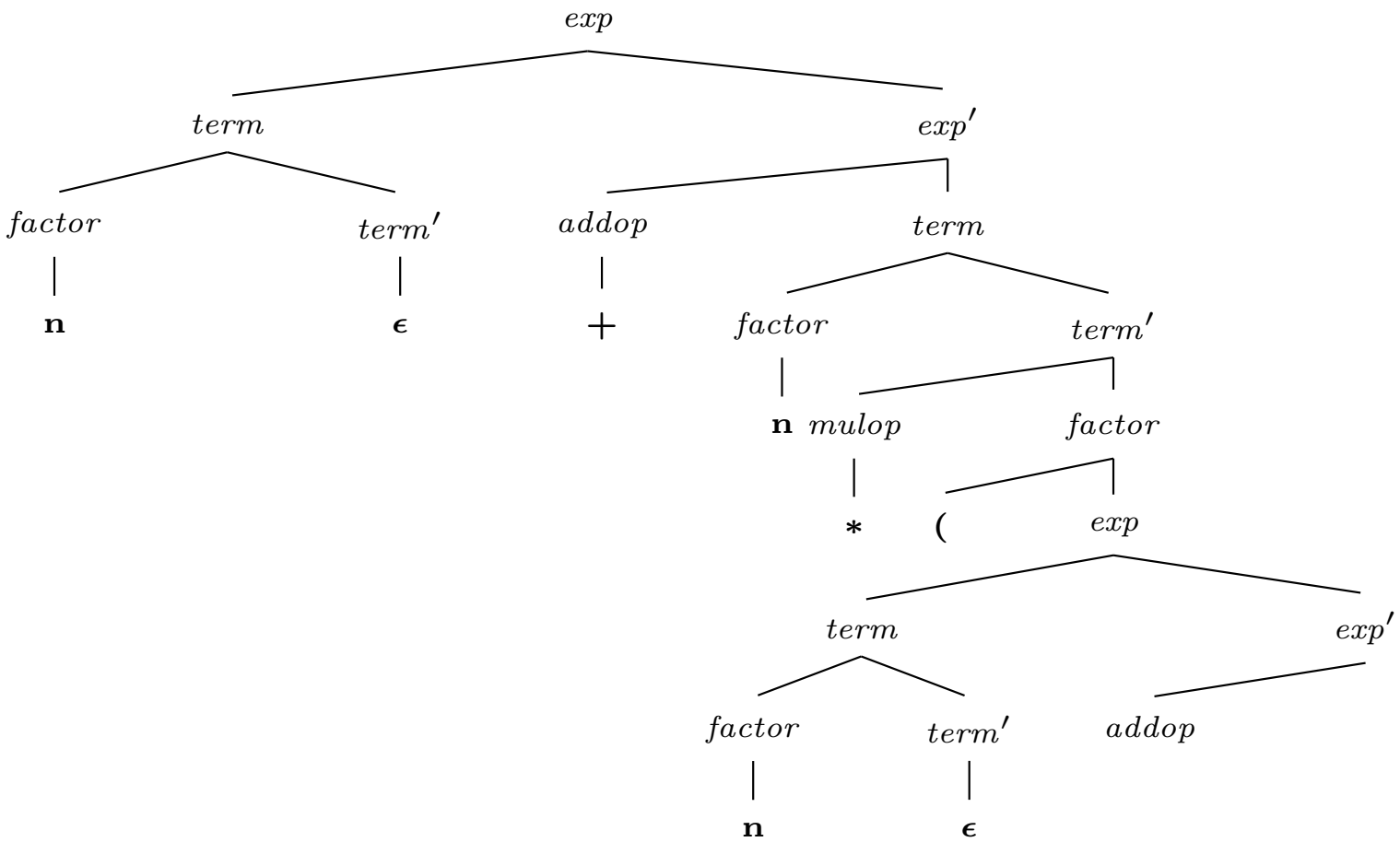
Best viewed as a tree



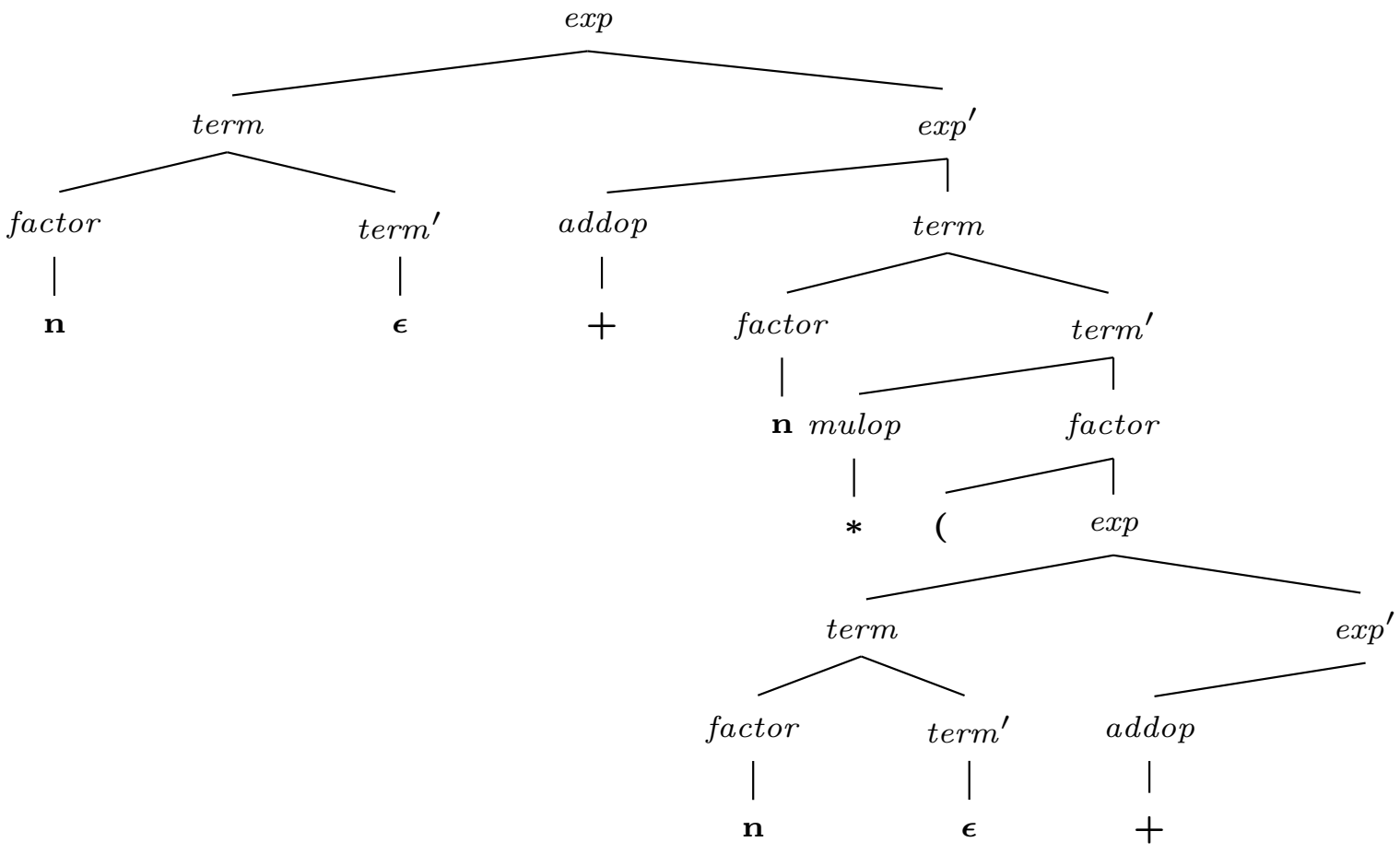
Best viewed as a tree



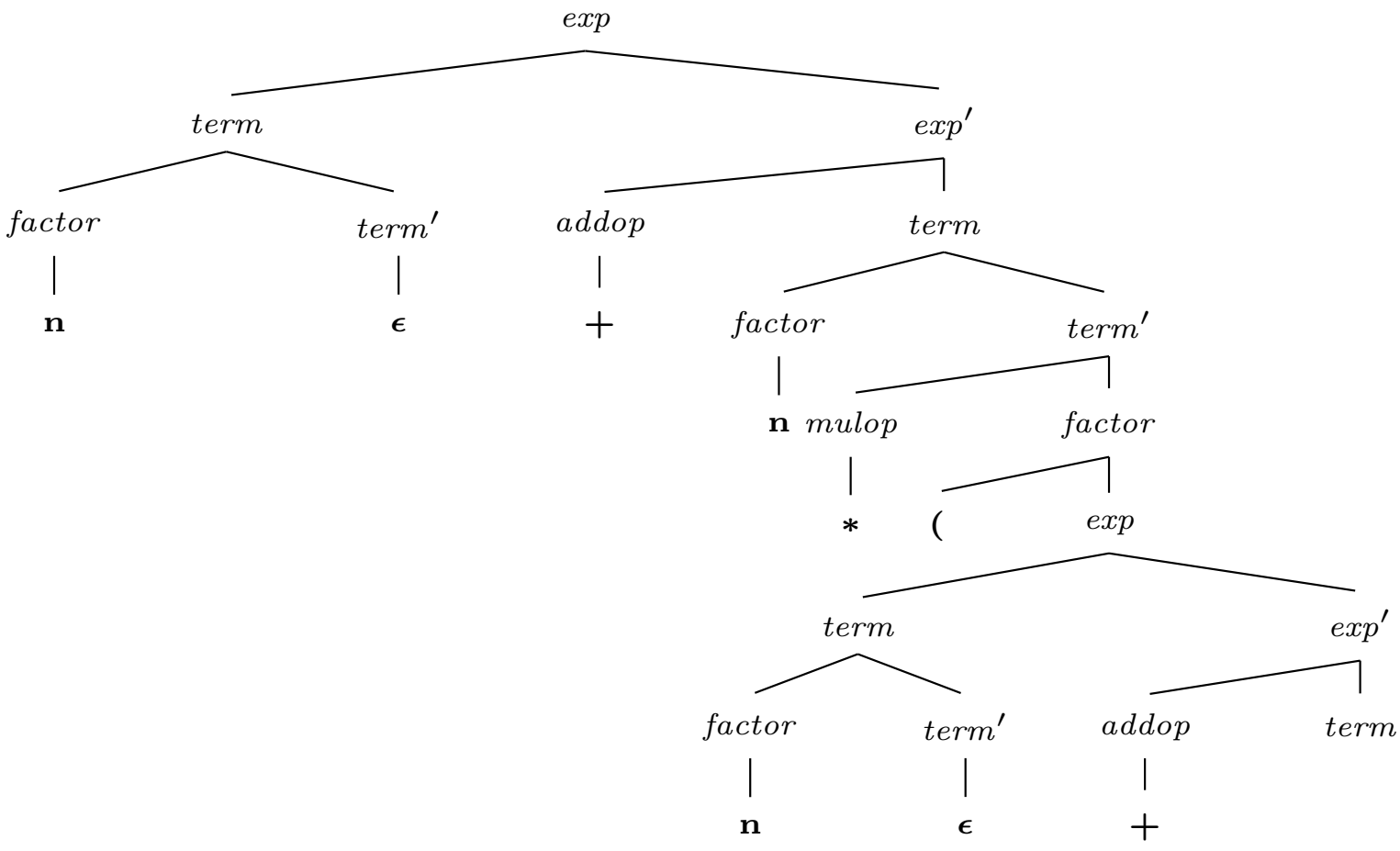
Best viewed as a tree



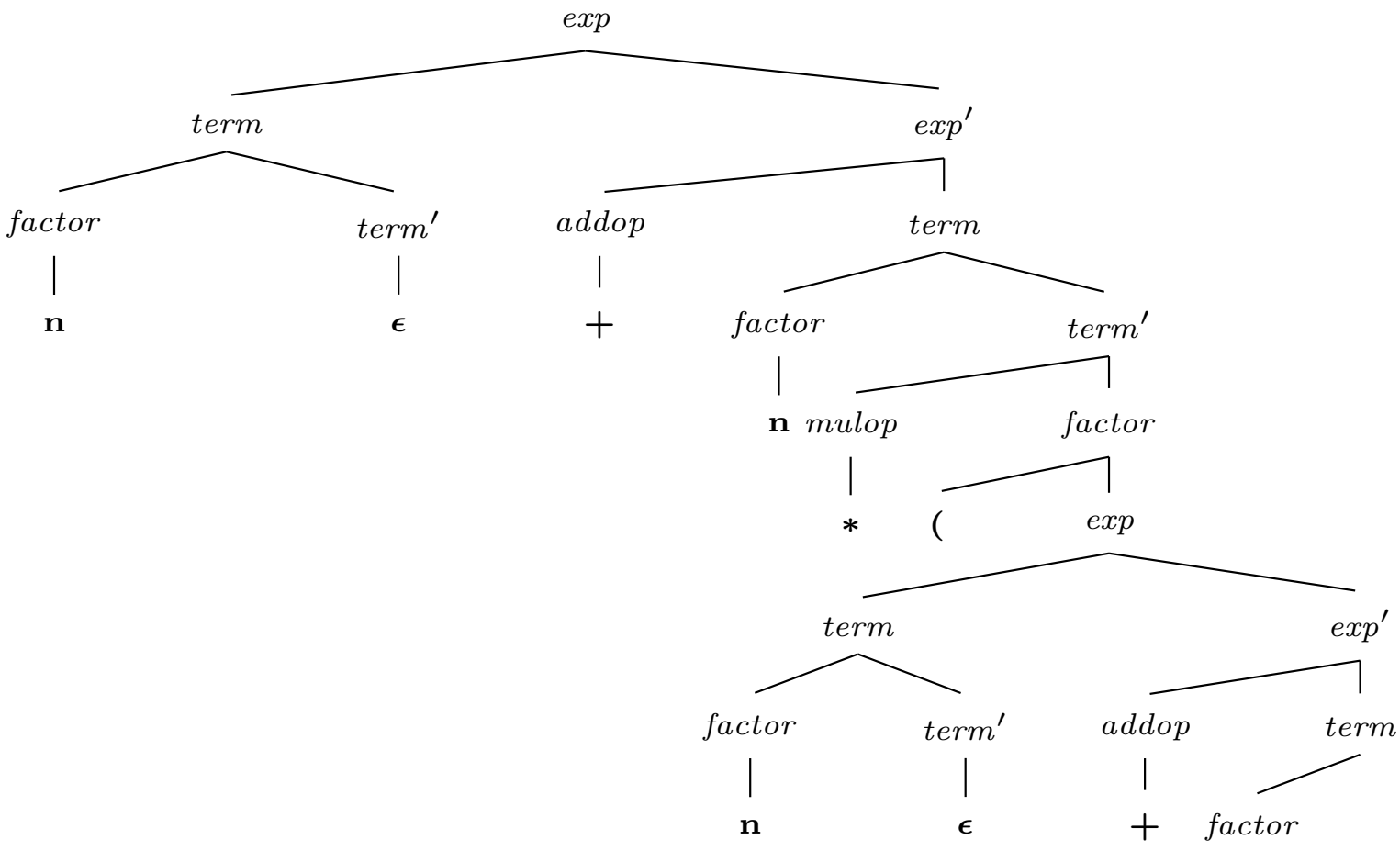
Best viewed as a tree



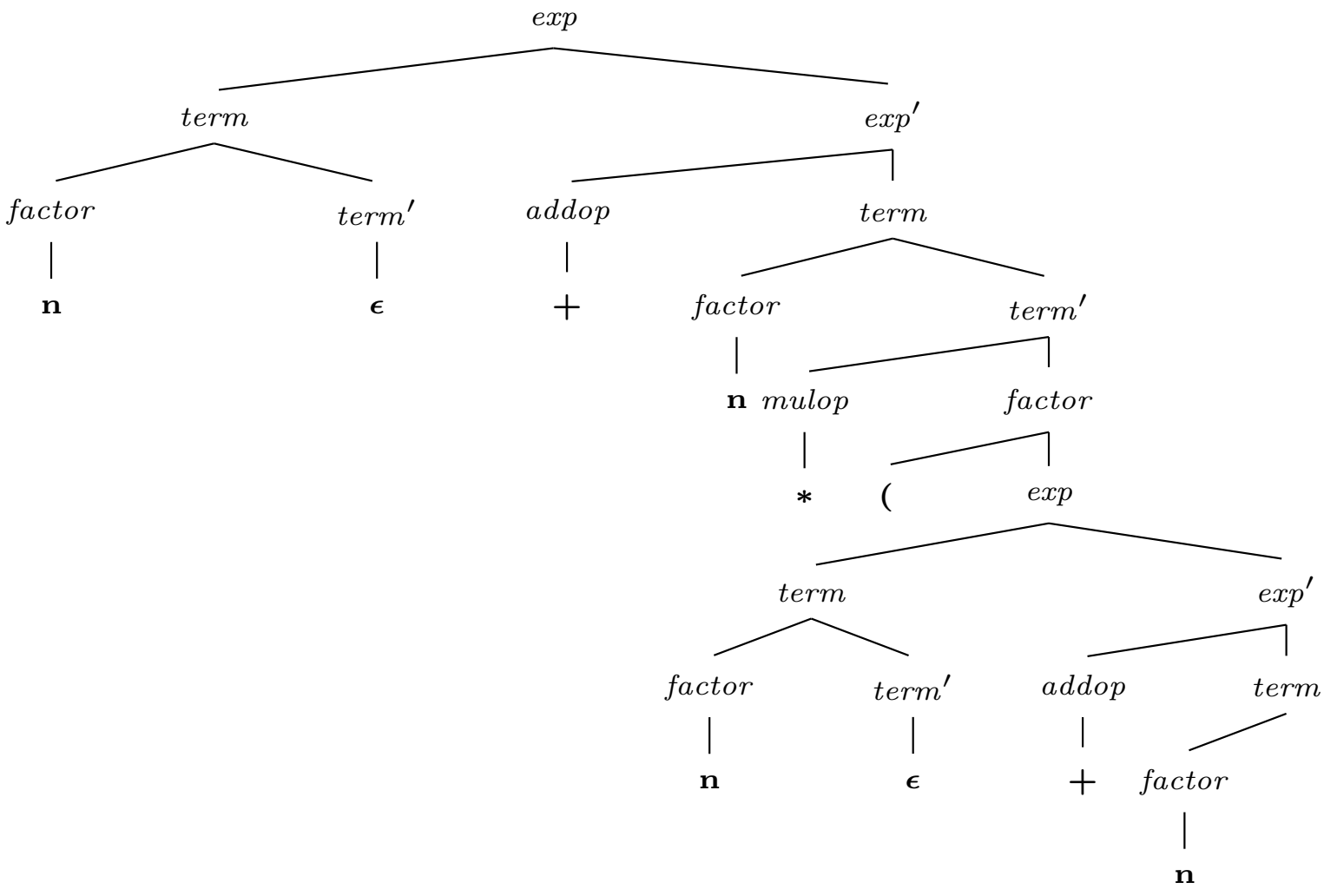
Best viewed as a tree



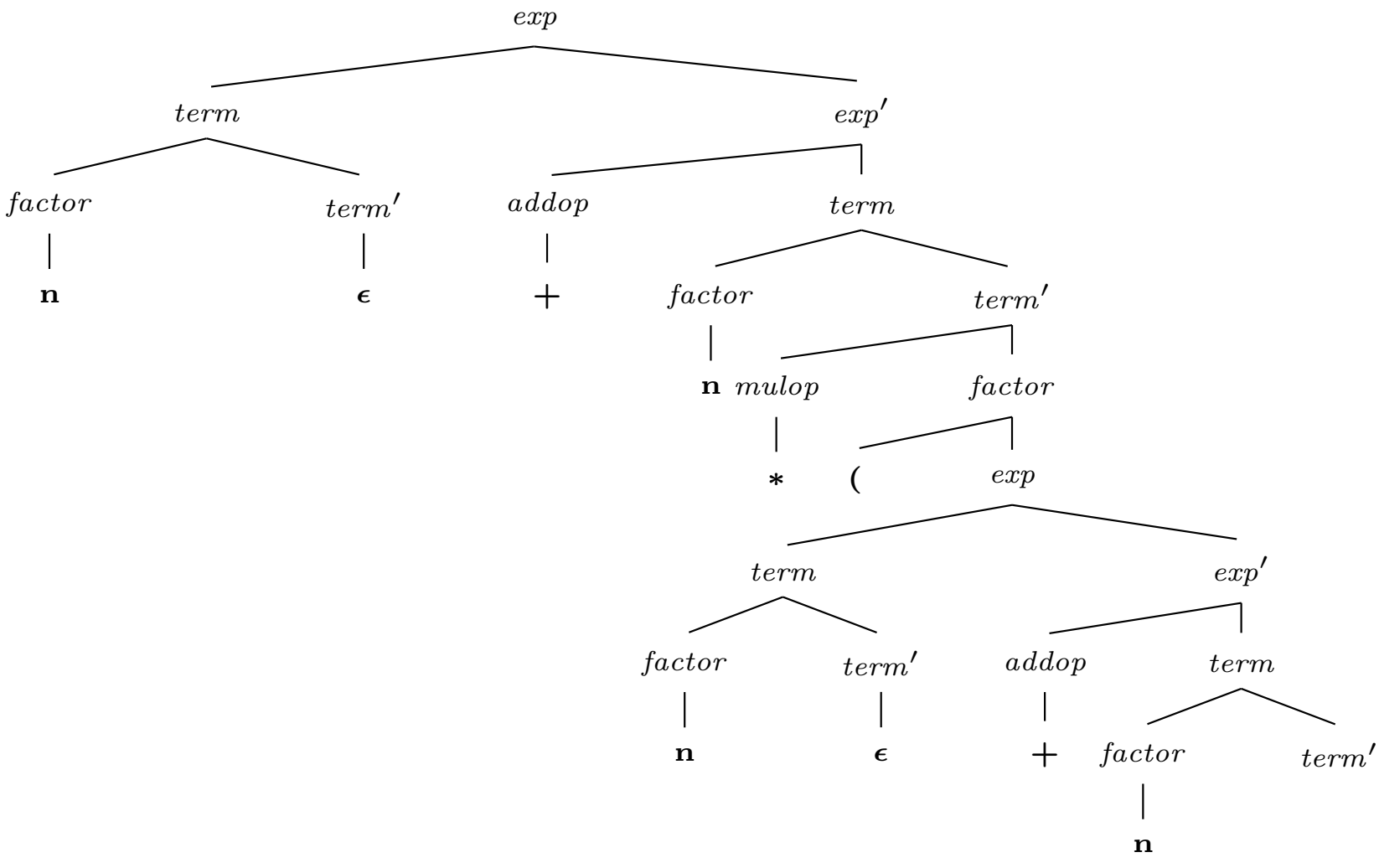
Best viewed as a tree



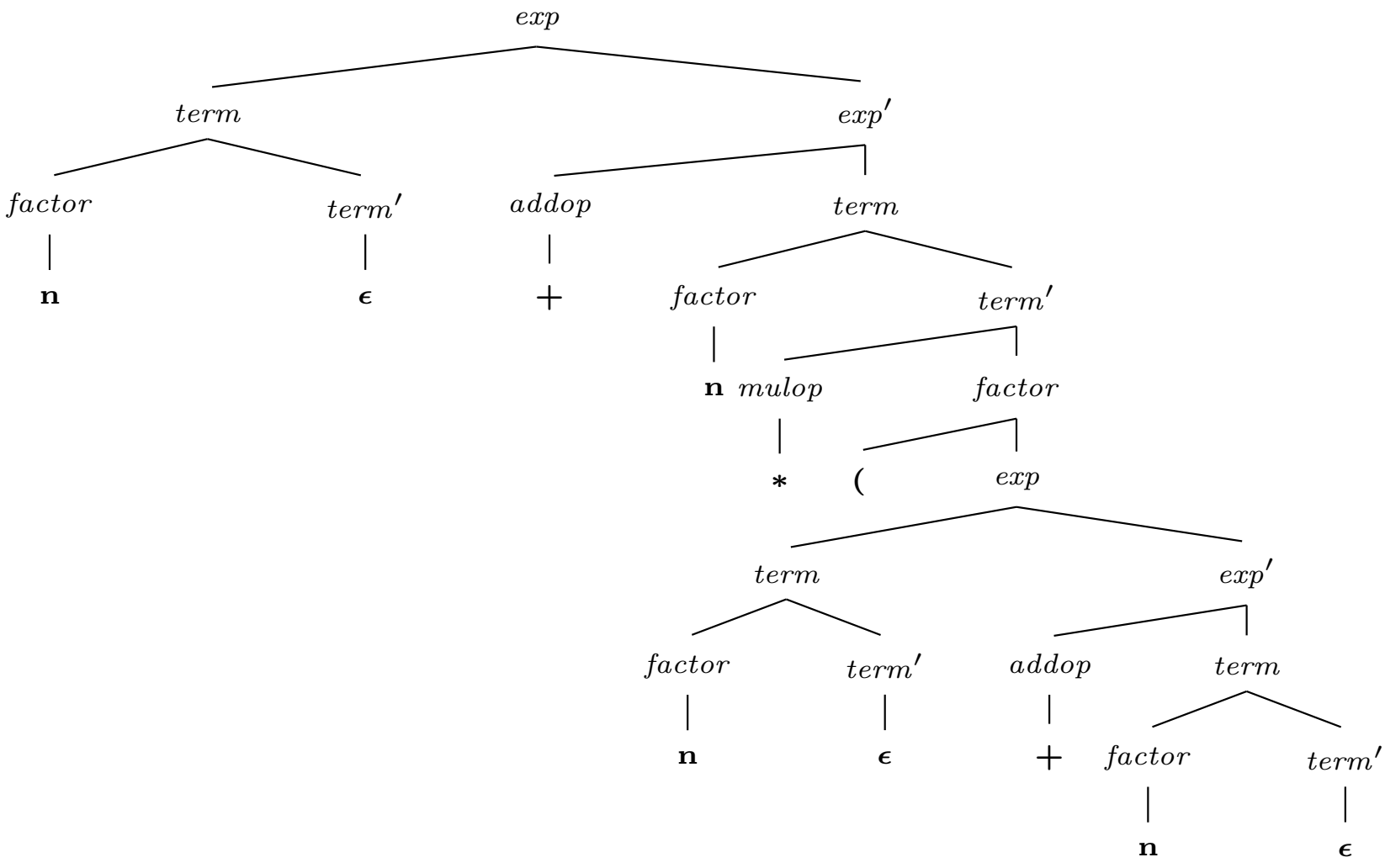
Best viewed as a tree



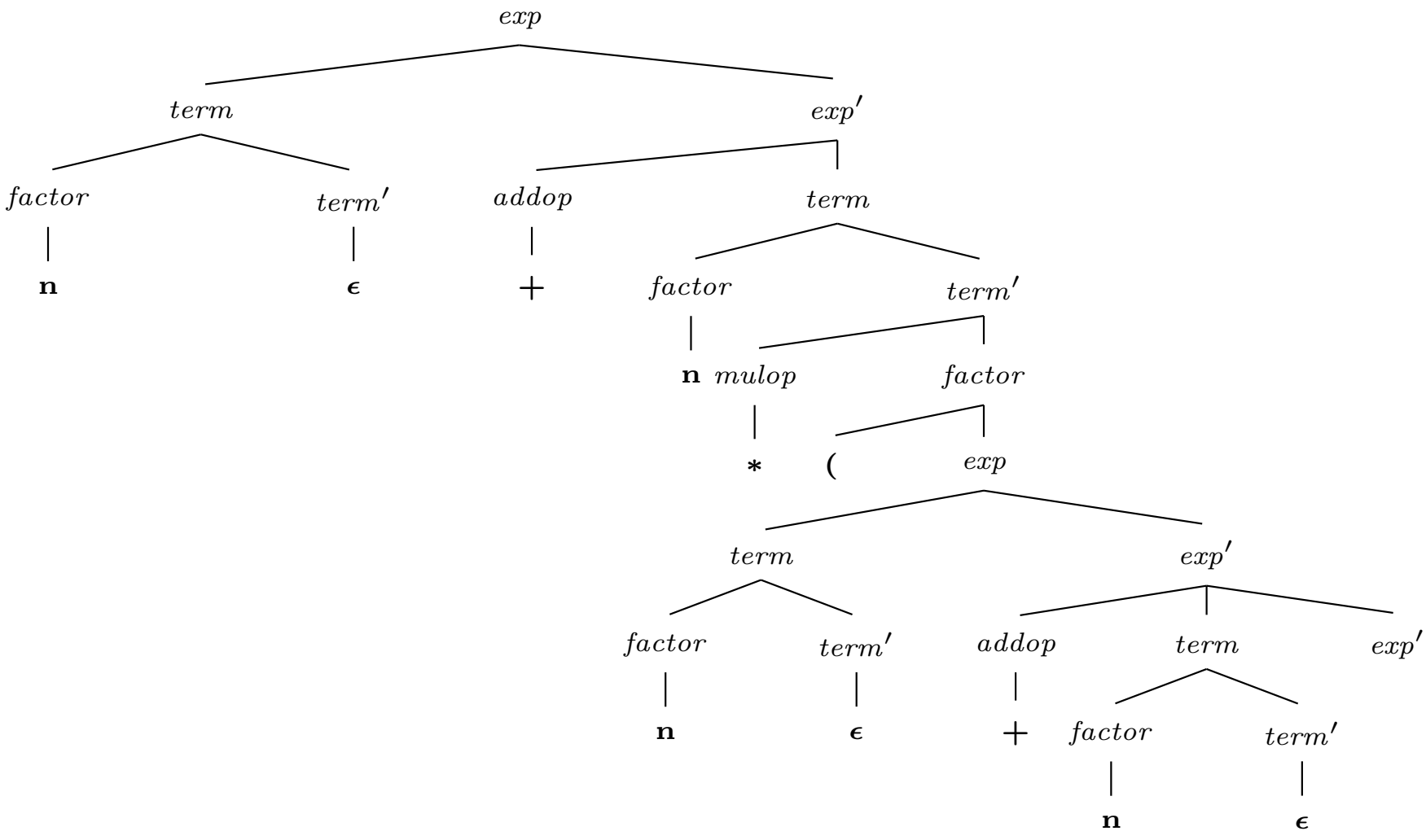
Best viewed as a tree



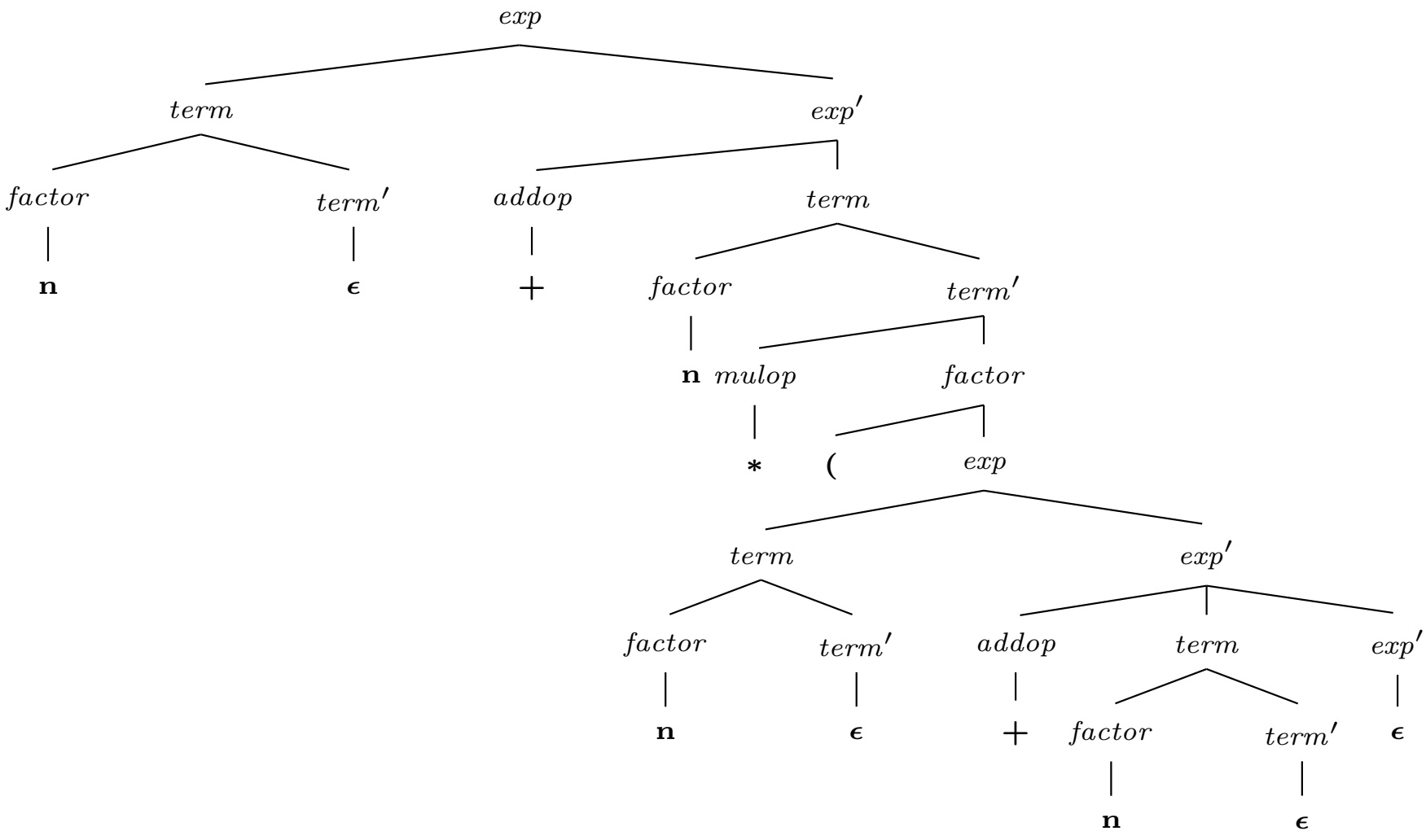
Best viewed as a tree



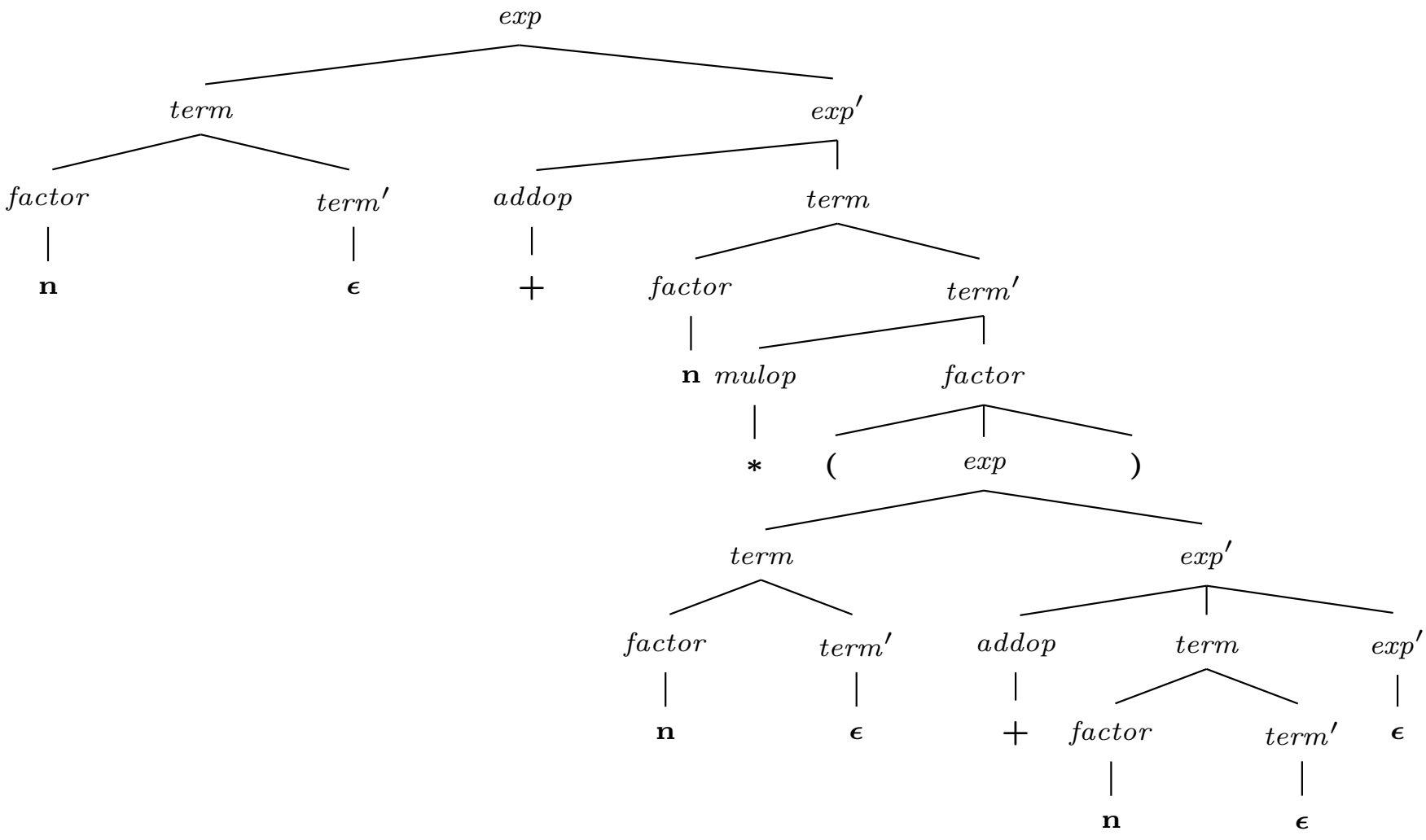
Best viewed as a tree



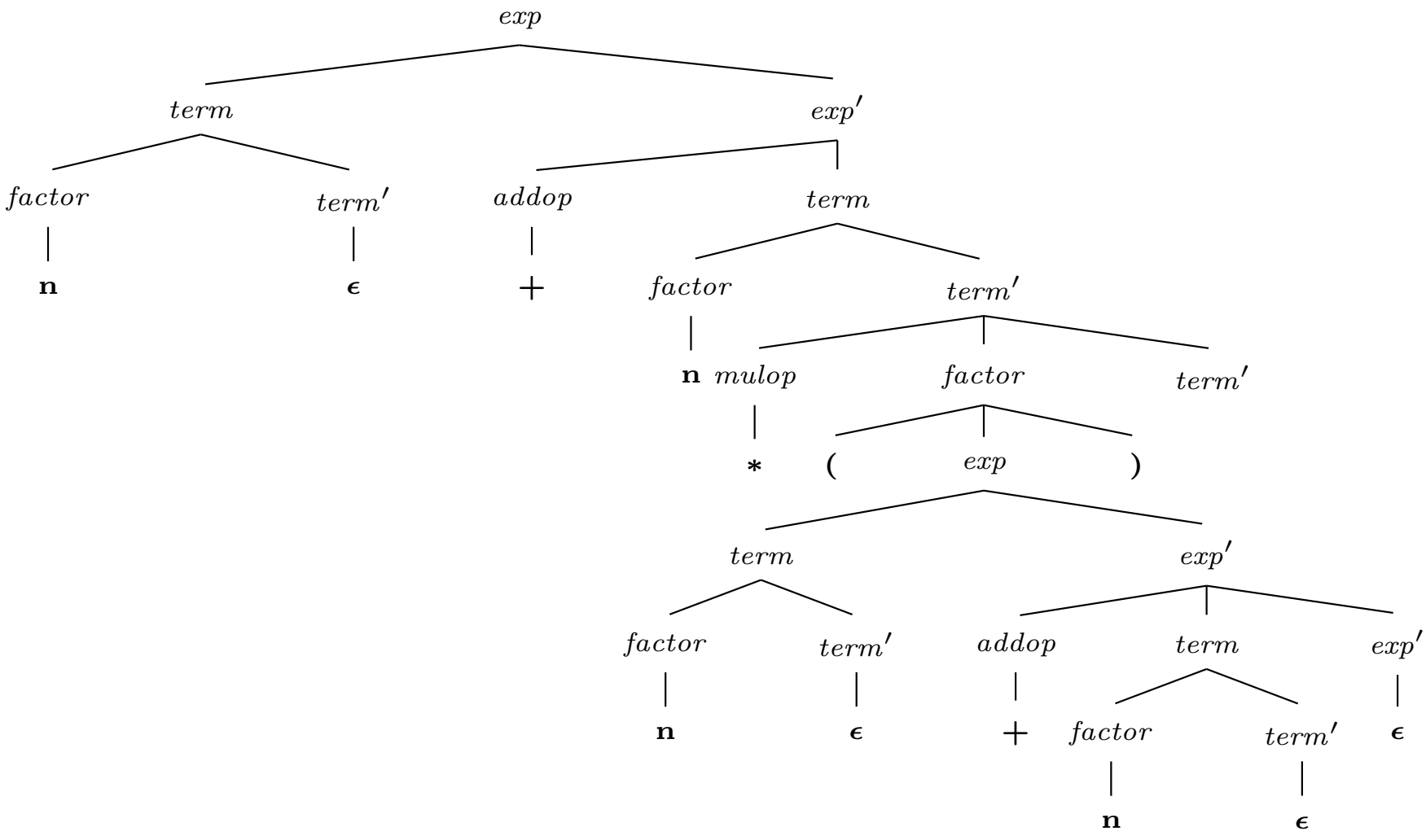
Best viewed as a tree



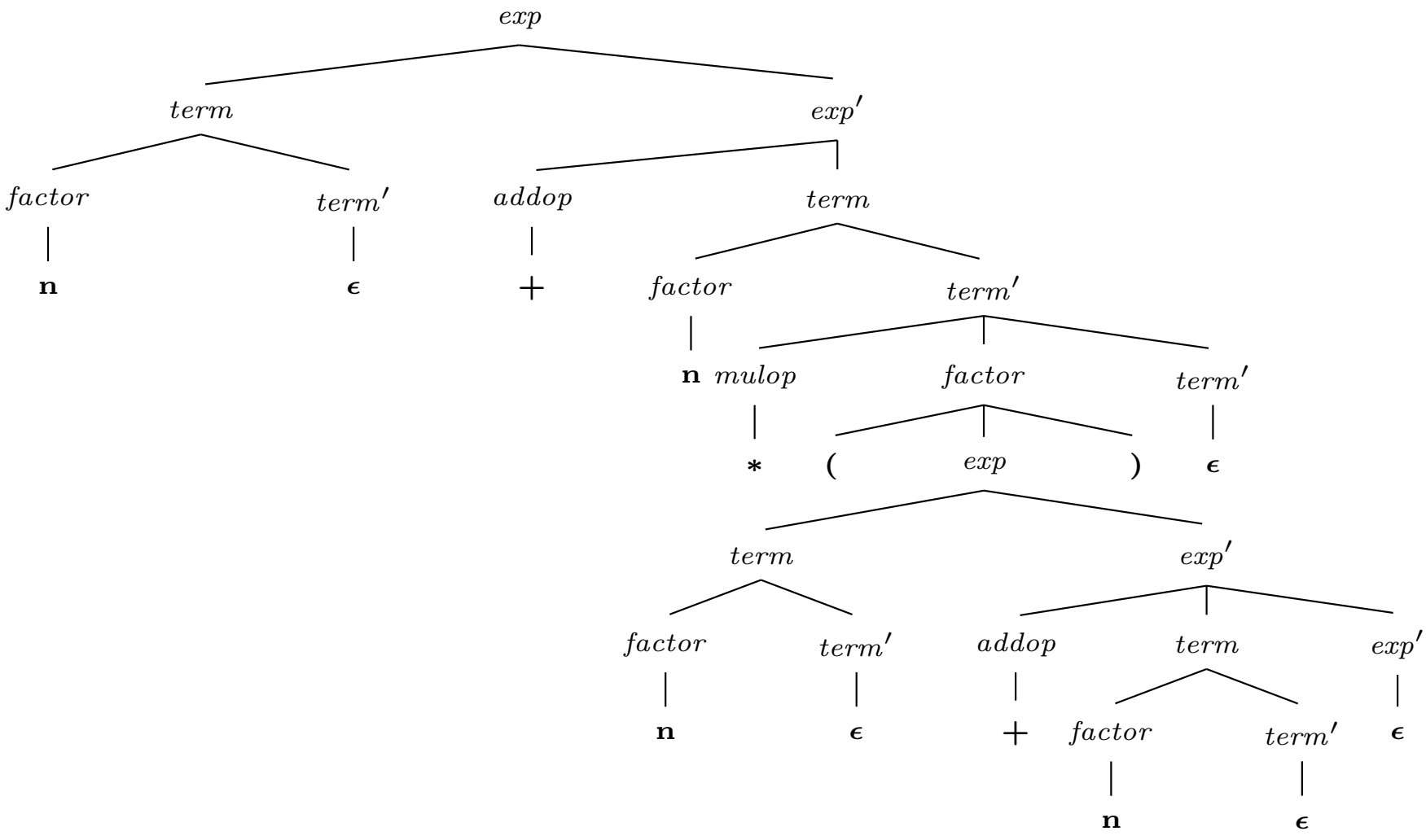
Best viewed as a tree



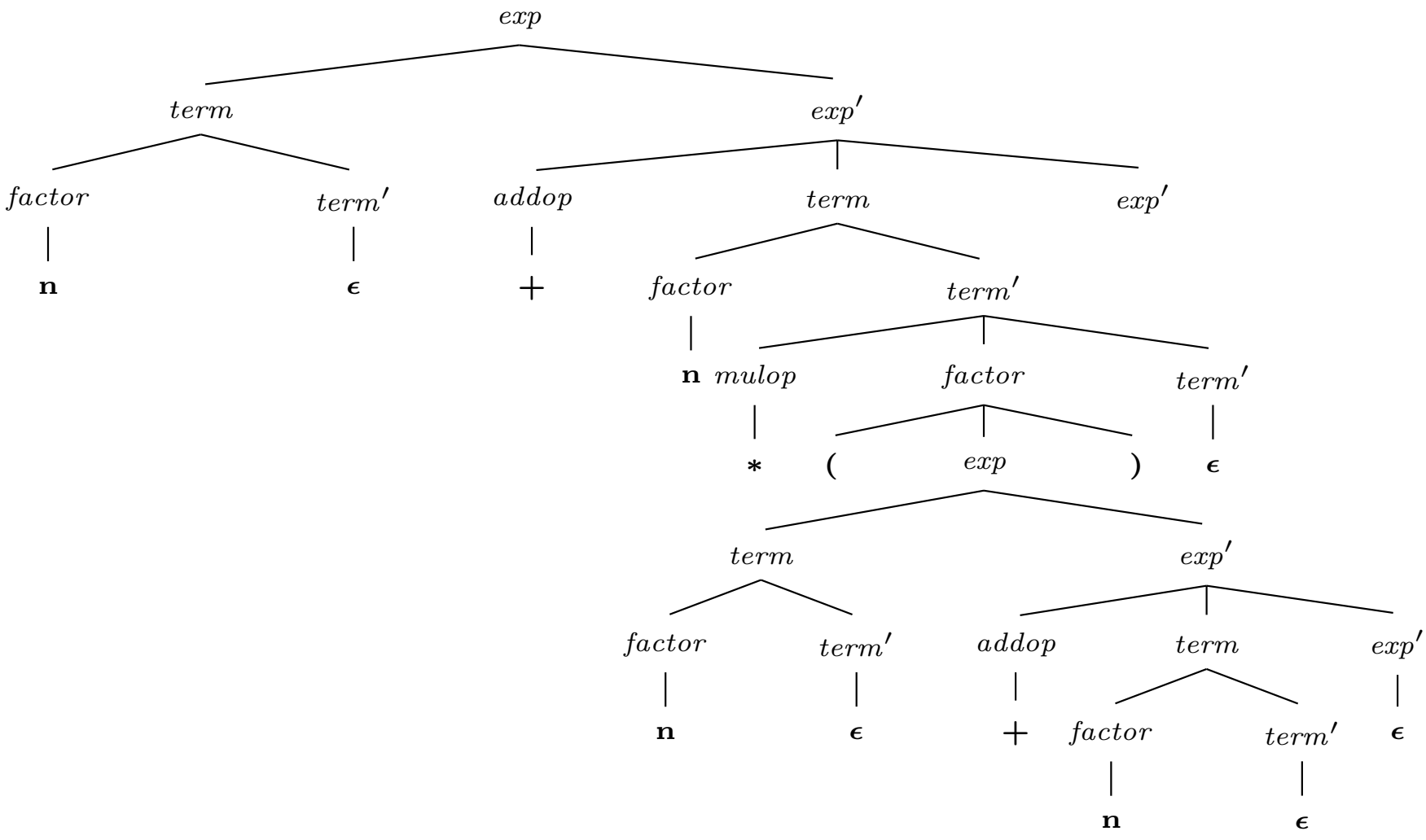
Best viewed as a tree



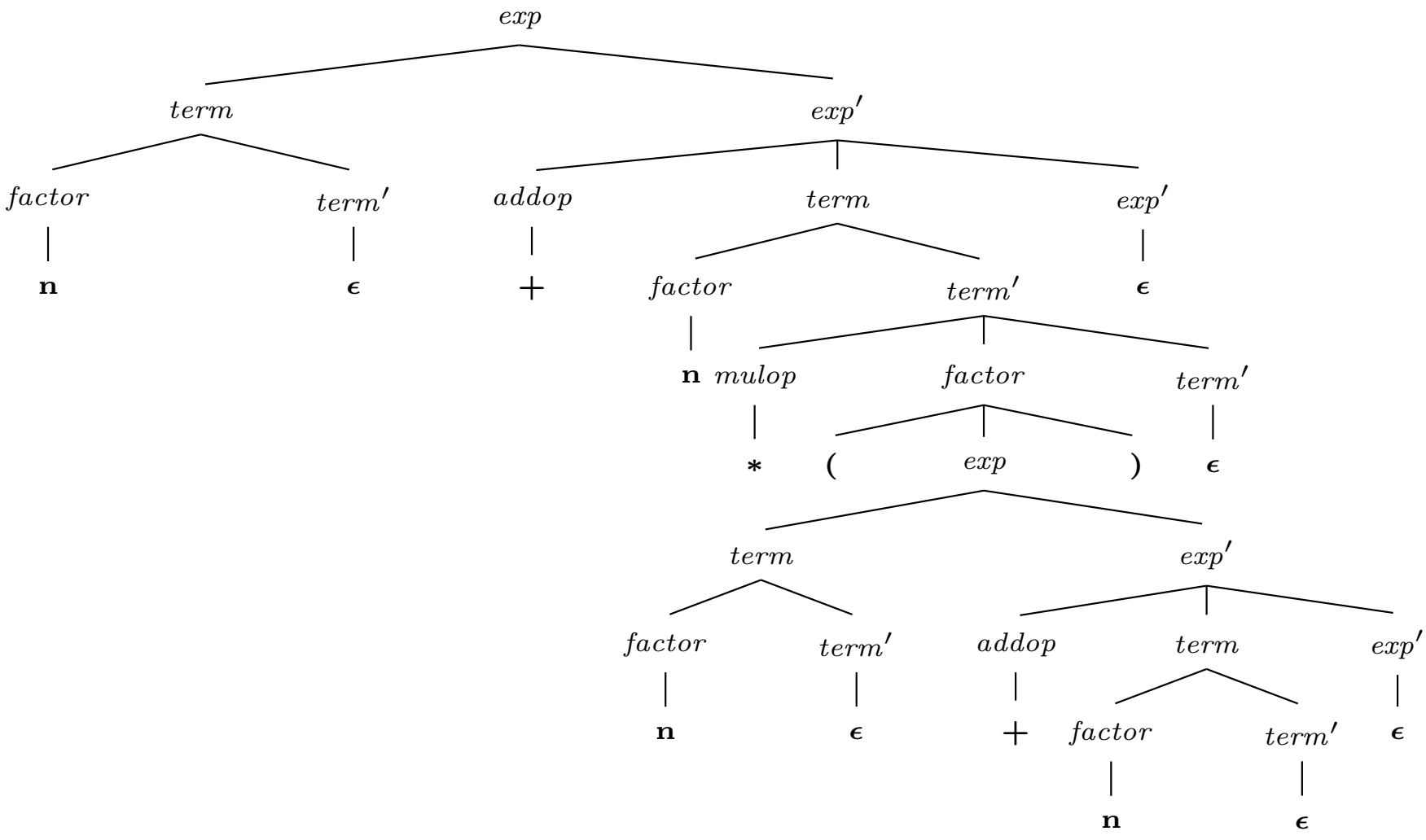
Best viewed as a tree



Best viewed as a tree



Best viewed as a tree



Non-determinism?

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

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Oracular derivation

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

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



Two principle sources of non-determinism

Using production $A \to \beta$

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

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

2 choices to make

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Left-most derivation

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Non-determinism vs. ambiguity

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

Lemma (Left or right, who cares)

$$S \Rightarrow_l^* w \quad \textit{iff} \quad S \Rightarrow_r^* w \quad \textit{iff} \quad S \Rightarrow^* w.$$

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

Using production $A \to \beta$

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



Non-determinism vs. ambiguity

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

Lemma (Left or right, who cares)

$$S \Rightarrow_l^* w \quad \textit{iff} \quad S \Rightarrow_r^* w \quad \textit{iff} \quad S \Rightarrow^* w.$$

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

Using production $A \to \beta$

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



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

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

Is that the correct choice?

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

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

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

Needed (minimal requirement):

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Deterministic, yes, but still impractical

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

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

$\Rightarrow$ impractical, therefore:

Look-ahead of length k

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

LL(k) grammars

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing



Section

First and follow sets

Chapter 4 “Parsing”
Course “Compiler Construction”
Martin Steffen
Spring 2023

First and Follow sets

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

First-set of A

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

Follow-set of A

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

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



First sets



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Definition (First set)

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

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

Definition (Nullable)

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

Examples

- in many languages

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

- in many languages:

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

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

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



Remarks

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

$\Rightarrow$ recursive definition



A more algorithmic/recursive definition

- grammar *symbol* X: terminal or non-terminal or ϵ

Definition (First set of a symbol)

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

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

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

   2.1 $First(X)$ contains $First(X_1) \setminus \{\epsilon\}$
   2.2 If, for some $i < n$, *all* $First(X_1), \dots, First(X_i)$ contain ϵ, then $First(X)$ contains $First(X_{i+1}) \setminus \{\epsilon\}$.
   2.3 If all $First(X_1), \dots, First(X_n)$ contain ϵ, then $First(X)$ contains $\{\epsilon\}$.



For words



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Definition (First set of a word)

Given a grammar G and word α. The *first-set* of

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

written $First(\alpha)$ is satisfies the following conditions

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

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

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

```
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
```

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

Initialization

```
1  for all X ∈ Σ_T ∪ {ε} do
2      First[X] := {X}
3  end;
4
5  for all non-terminals A do
6     First[A] := {}
7  end
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Pseudo code

```
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
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example expression grammar (from before)

$$
\begin{array}{rcl}
exp & \rightarrow & exp\ addop\ term \mid term \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & term\ mulop\ factor \mid factor \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
\tag{3}
$$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example expression grammar (expanded)

$$
\begin{aligned}
exp &\rightarrow exp\ addop\ term \\
exp &\rightarrow term \\
addop &\rightarrow + \\
addop &\rightarrow - \\
term &\rightarrow term\ mulop\ factor \\
term &\rightarrow factor \\
mulop &\rightarrow * \\
factor &\rightarrow (\ exp\ ) \\
factor &\rightarrow \mathbf{number}
\end{aligned}
\tag{4}
$$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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

"Run" of the algo

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



Collapsing the rows & final result

- results per pass:

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

- final results (at the end of pass 3, resp. 4):

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



Follow sets



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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

$$Follow_G(A) = \{a \mid S\,\$ \Rightarrow_G^* \alpha_1 A a \alpha_2, \quad a \in \Sigma_T + \{\,\$\,\}\}\ . \tag{5}$$

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

Follow sets, recursively



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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 $Follow(B)$.

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

More imperative representation in pseudo code

```
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
```

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

Expression grammar once more

$$
\begin{aligned}
exp &\rightarrow exp\ addop\ term \\
exp &\rightarrow term \\
addop &\rightarrow + \\
addop &\rightarrow - \\
term &\rightarrow term\ mulop\ factor \\
term &\rightarrow factor \\
mulop &\rightarrow * \\
factor &\rightarrow (\ exp\ ) \\
factor &\rightarrow \mathbf{number}
\end{aligned}
\tag{6}
$$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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



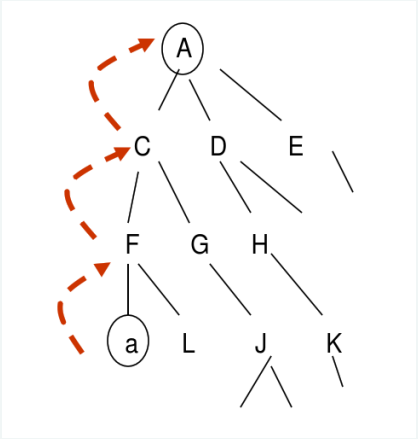
"Run" of the algo

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*) = {$, +, -,)}	

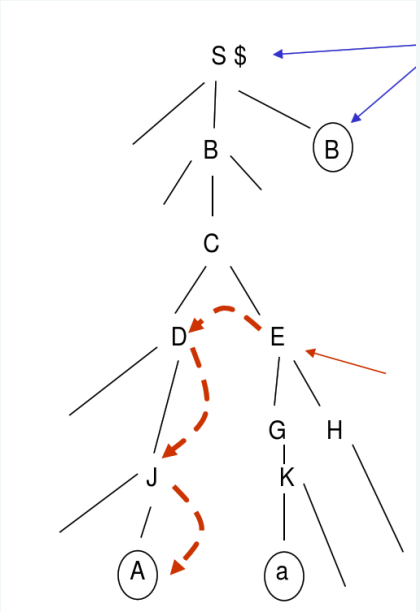


Illustration of first/follow sets

$a \in First(A)$



$a \in Follow(A)$

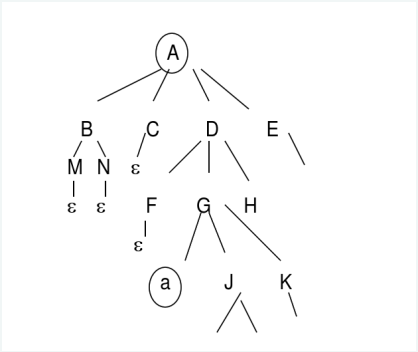


- red arrows: illustration of *information flow* in the algos
- run of $Follow$:
  - relies on $First$
  - in particular $a \in First(E)$ (right tree)
- $\$ \in Follow(B)$

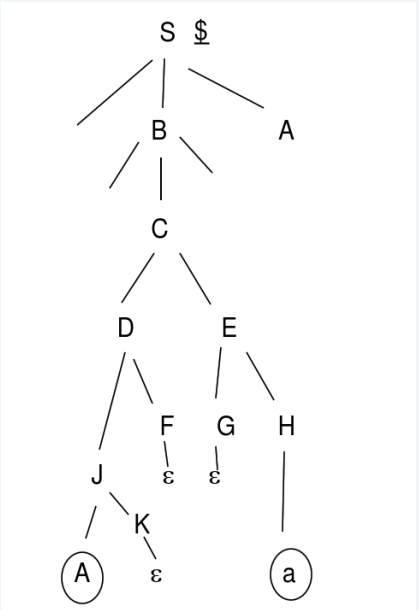


More complex situation (nullability)

$a \in First(A)$



$a \in Follow(A)$





Section

Massaging grammars

Chapter 4 “Parsing”
Course “Compiler Construction”
Martin Steffen
Spring 2023

Some forms of grammars are less desirable than others



- left-recursive production:

$$A \rightarrow A\alpha$$

more precisely: example of *immediate* left-recursion

- 2 productions with common "left factor":

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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Some simple examples for both

- left-recursion

$$exp \rightarrow exp + term$$

- classical example for common left factor: rules for conditionals

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Transforming the expression grammar

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

After removing left recursion

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

- still *unambiguous*
- unfortunate: *associativity* now different!
- note also: ϵ-productions & nullability



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Left-recursion removal



Left-recursion removal

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

- price: ϵ-productions (+ another one, see later)
- *2 cases* to consider
  1. immediate (or direct) recursion
     - simple
     - general
  2. *indirect* (or mutual) recursion

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Left-recursion removal: simplest case

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

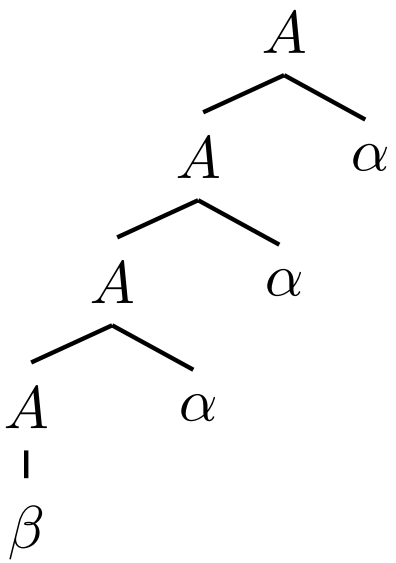
$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

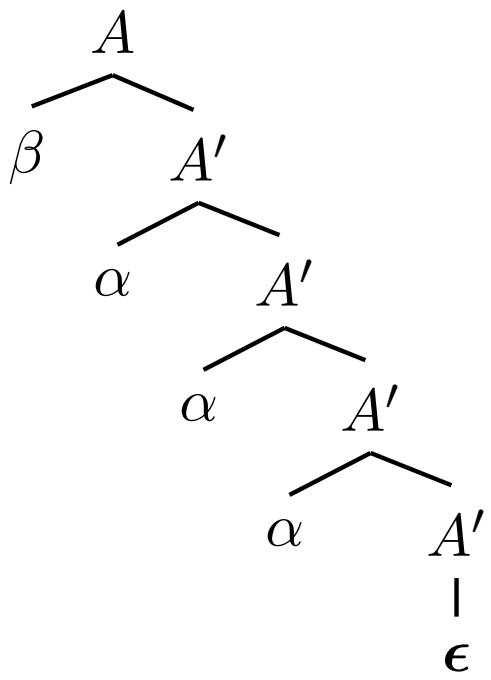


Schematic representation

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

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$





Remarks

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

$$A \to \beta\{\alpha\}$$

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



Left-recursion removal: immediate recursion (multiple)



Before

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

After

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

Note: can be written in *EBNF* as:

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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Removal of: general left recursion

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

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

"current" = rule in the current stage of algo

Example (for the general case)

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example (for the general case)

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example (for the general case)

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

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

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



Example (for the general case)

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

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

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Left factor removal

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

$\Rightarrow$ common left factor undesirable

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

Simple situation

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example: sequence of statements



Before

$$\begin{array}{rcl} stmts & \rightarrow & stmt\ ;\ stmts \\ & | & stmt \end{array}$$

After

$$\begin{array}{rcl} stmts & \rightarrow & stmt\ stmts' \\ stmts' & \rightarrow & ;\ stmts \mid \epsilon \end{array}$$

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example: conditionals

Before

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

After

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example: conditionals (without else)

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Not all factorization doable in "one step"

Starting point

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

After 1 step

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

After 2 steps

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

- note: we choose the *longest* common prefix (= longest



Left factorization

```
1
2  while there are changes to the grammar do
3    for each nonterminal A do
4      let α be a prefix of max. length that is shared
5                    by two or more productions for A
6      if    α ≠ ϵ
7      then
8          let A → α1 | ... | αn be all
9                    prod. for A and suppose that α1, ..., αk share α
10                   so that A → αβ1 | ... | αβk | αk+1 | ... | αn ,
11                   that the βj 's share no common prefix, and
12                   that the αk+1, ..., αn do not share α.
13         replace rule A → α1 | ... | αn by the rules
14         A → αA′ | αk+1 | ... | αn
15         A′ → β1 | ... | βk
16     end
17   end
18 end
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing



Section

LL-parsing (mostly LL(1))

Chapter 4 “Parsing”
Course “Compiler Construction”
Martin Steffen
Spring 2023

Parsing LL(1) grammars

- *this lecture*: we don't do LL(k) with $k > 1$
- LL(1): particularly easy to understand and to implement (efficiently)
- not as expressive than LR(1) (see later), but still kind of decent

LL(1) parsing principle

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

- two flavors for LL(1) parsing here (both are top-down parsers)
  - *recursive descent*
  - *table-based* LL(1) parser
- *predictive* parsers (no backtracking)



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Sample expression grammar again



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

factors and terms

$$
\begin{array}{rcl}
exp & \rightarrow & term\ exp' \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & factor\ term' \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
\tag{7}
$$

Look-ahead of 1: straightforward, but not trivial

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

$\Rightarrow$ read the next token, and, based on that, decide

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

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

Example: 2 productions for non-terminal $factor$

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

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Recursive descent: general set-up

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

Idea

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

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Recursive descent (in C-like)

method `factor` for nonterminal *factor*

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

```
1   void factor () {
2       switch (tok) {
3       case LPAREN: eat(LPAREN); expr(); eat(RPAREN);
4       case NUMBER: eat(NUMBER);
5       }
6   }
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Recursive descent (in ocaml)

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

```
1
2 let factor () =    (* function for factors *)
3   match !tok with
4     LPAREN ->  eat(LPAREN); expr(); eat(RPAREN)
5   | NUMBER ->  eat(NUMBER)
6   | _ -> ()              (* raise an error *)
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Slightly more complex

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

LL(1) principle (again)

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

$\Rightarrow$ definition of the $First$ set

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 $\alpha_1 \neq \alpha_2$:

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

The characterization meantions that the grammar has to be *reduced*. We did not bother to formally define it. At some point earlier, we have said, grammars can be "silly", like



Common problematic situation

- often: common *left factors* problematic

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

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

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

  1. *left-factoring*:

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



Recursive descent for left-factored *if-stmt*

```
1  procedure ifstmt()
2    begin
3      match ("if");
4      match  ("(");
5      exp();
6      match ")";
7      stmt();
8      if    token = "else"
9      then match ("else");
10           stmt()
11     end
12   end;
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Left recursion is a no-go

factors and terms

$$
\begin{array}{rcl}
exp & \rightarrow & exp\ addop\ term \mid term \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & term\ mulop\ factor \mid factor \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
\tag{8}
$$

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

Left-recursion

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

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



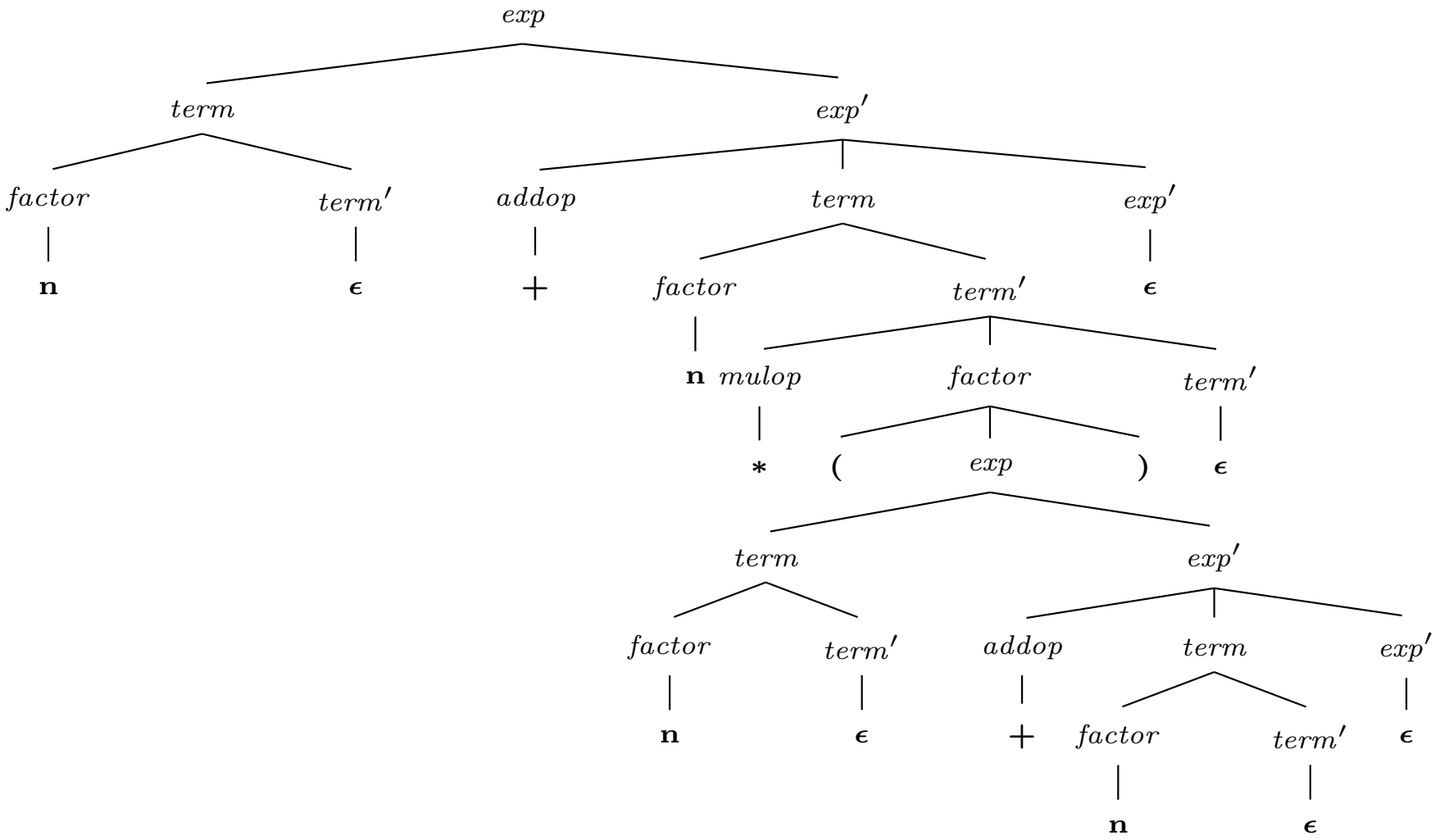
Removing left recursion may help

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

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

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

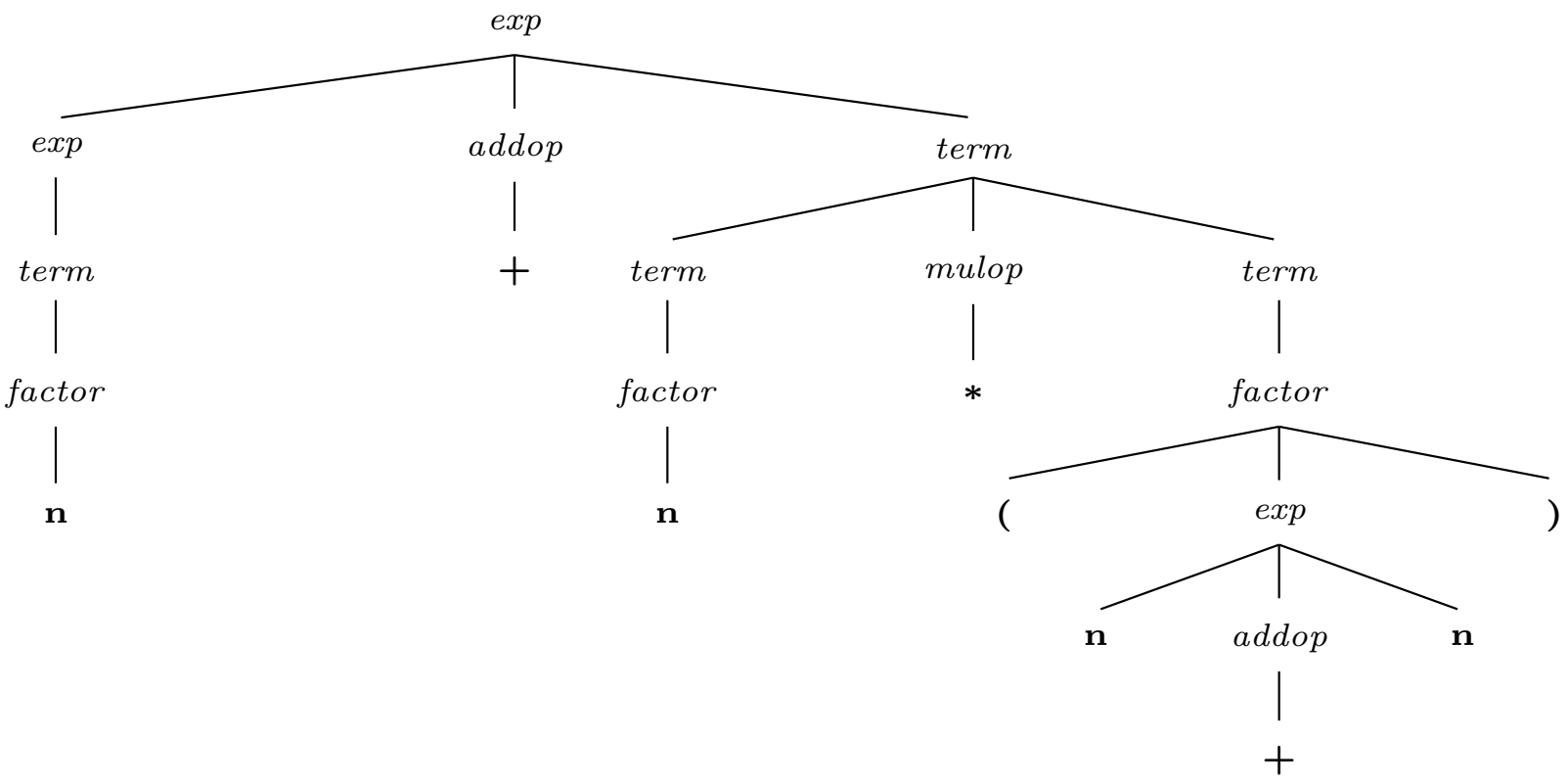
Recursive descent works, alright, but . . .



. . . who wants this form of trees?

Left-recursive grammar with nicer parse trees

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

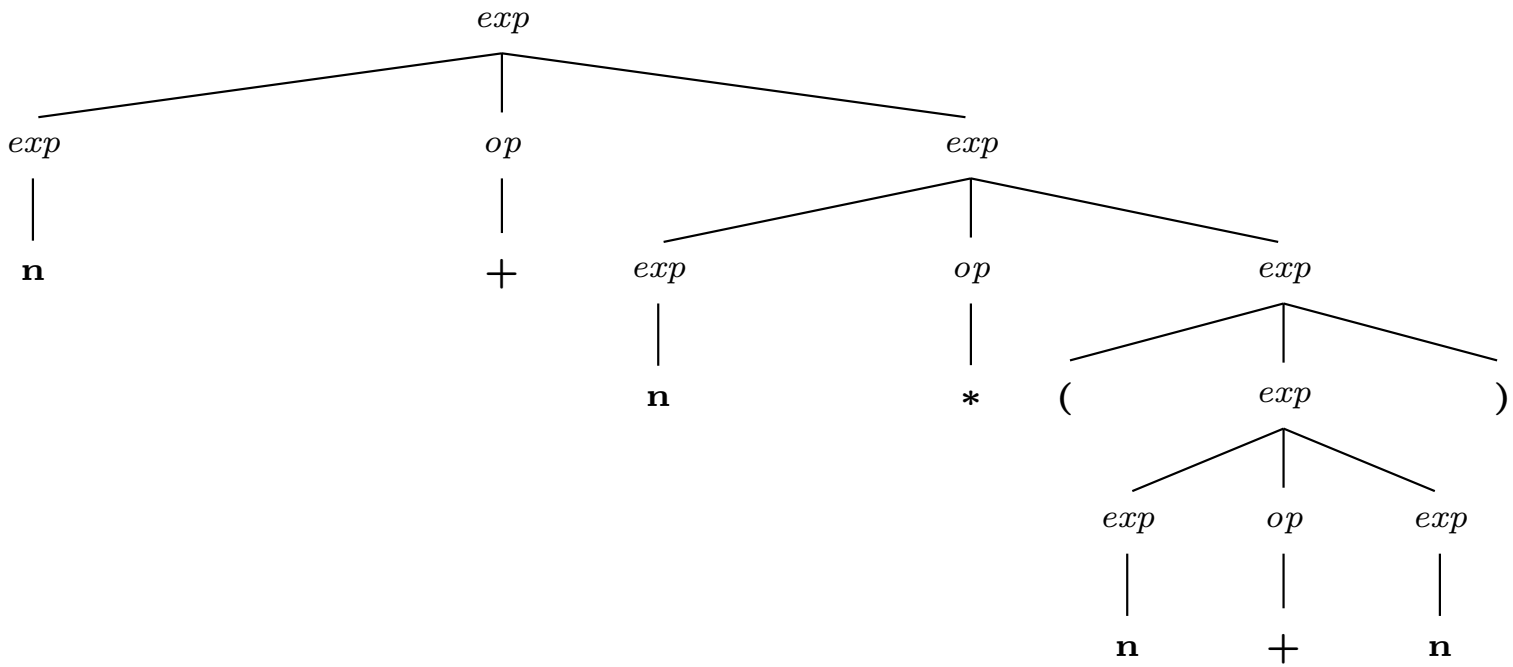


The simple "original" expression grammar (even nicer)

Flat expression grammar

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

$1 + 2 * (3 + 4)$



Associtivity problematic

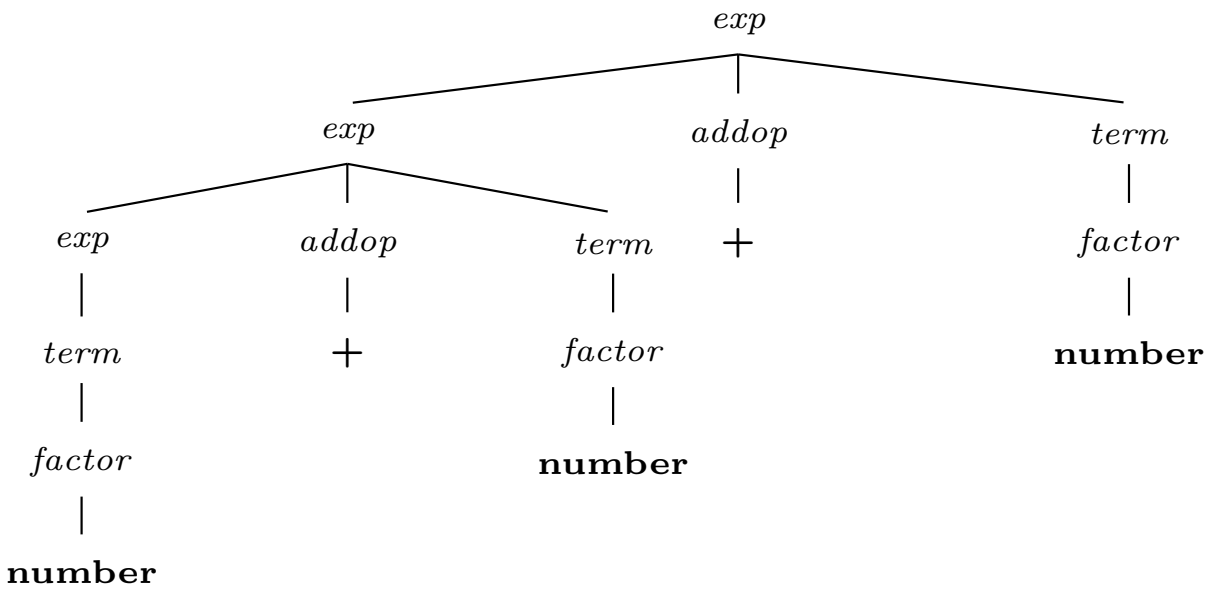
Precedence & assoc.

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

$3 + 4 + 5$

parsed "as"

$(3 + 4) + 5$



Associtivity problematic

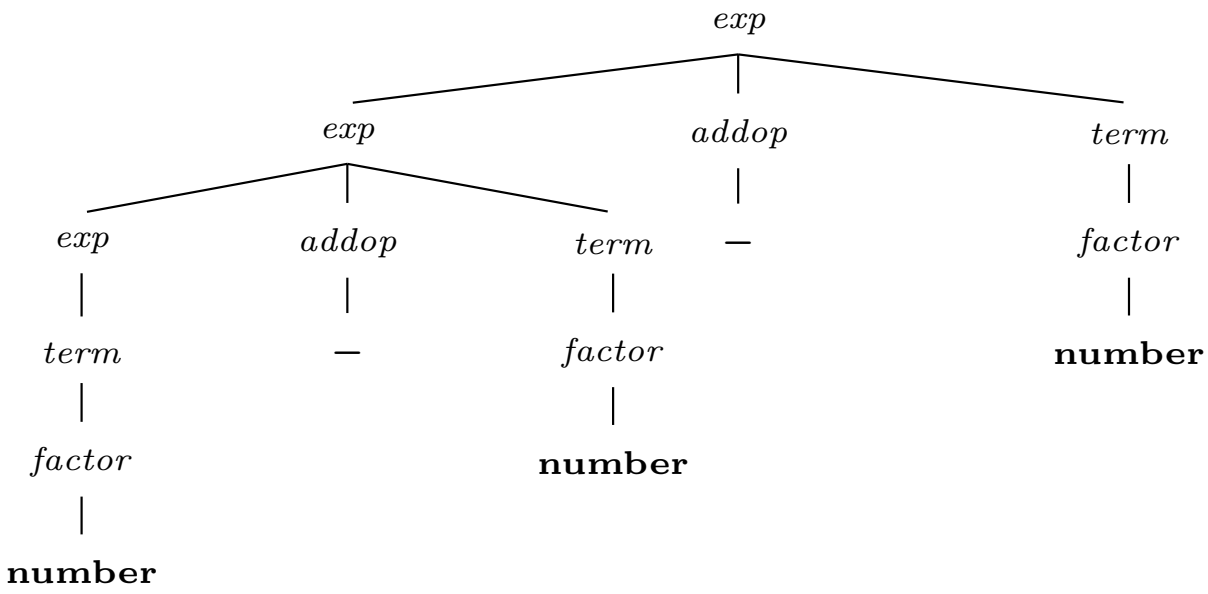
Precedence & assoc.

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

$3 - 4 - 5$

parsed "as"

$(3 - 4) - 5$

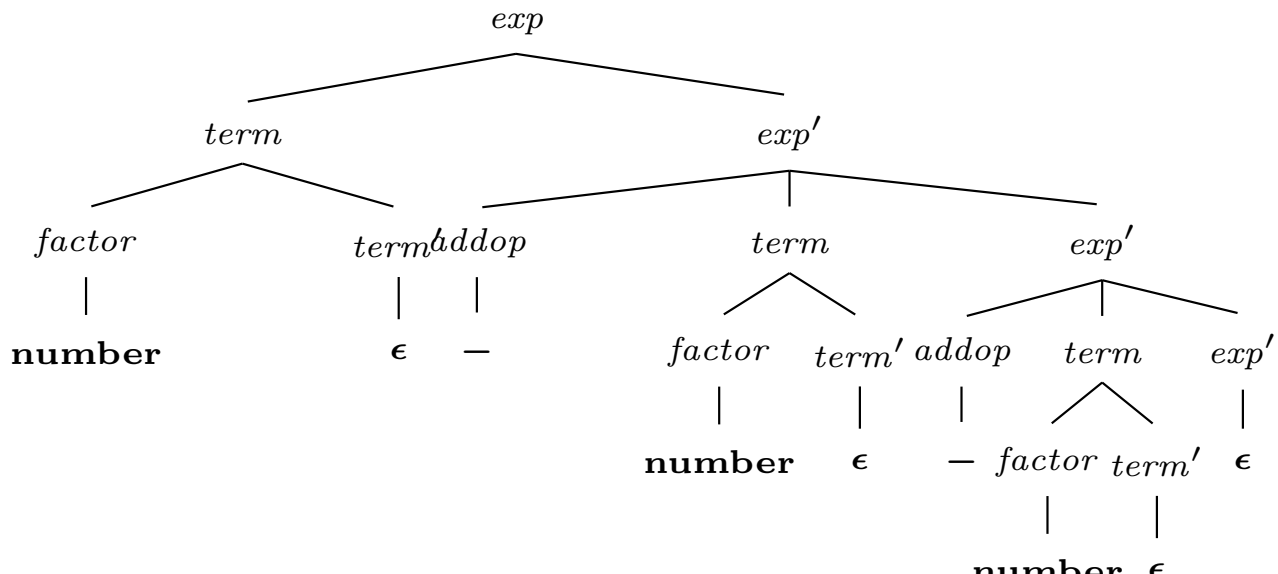


Now use the grammar without left-rec (but right-rec instead)

No left-rec.

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

$3 - 4 - 5$



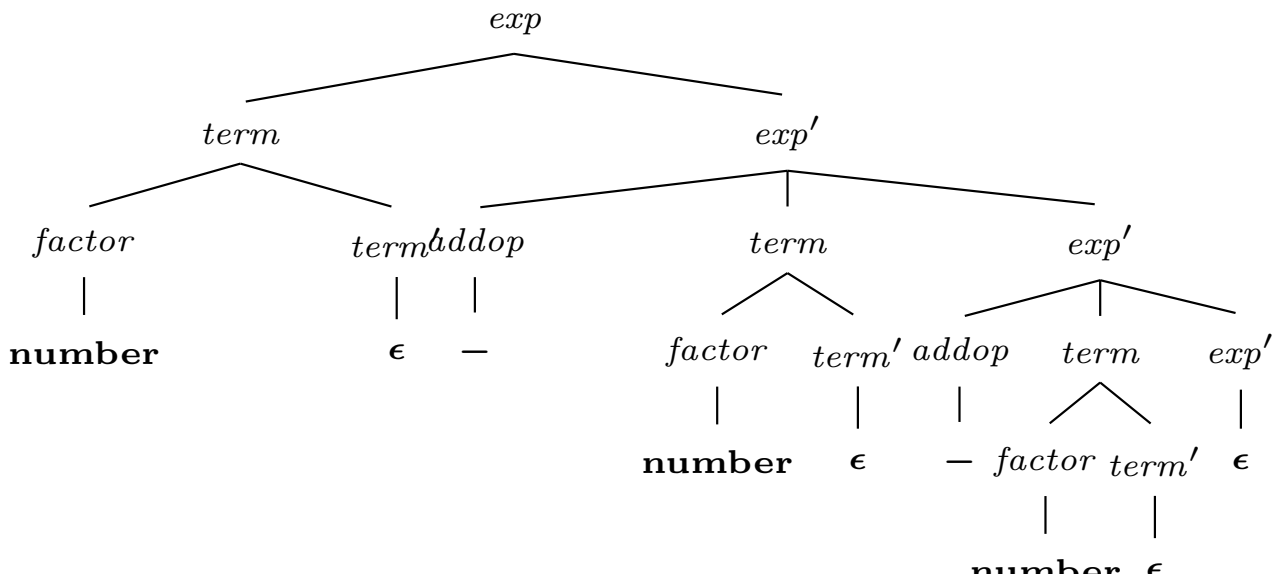
Now use the grammar without left-rec (but right-rec instead)

No left-rec.

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

$3 - 4 - 5$

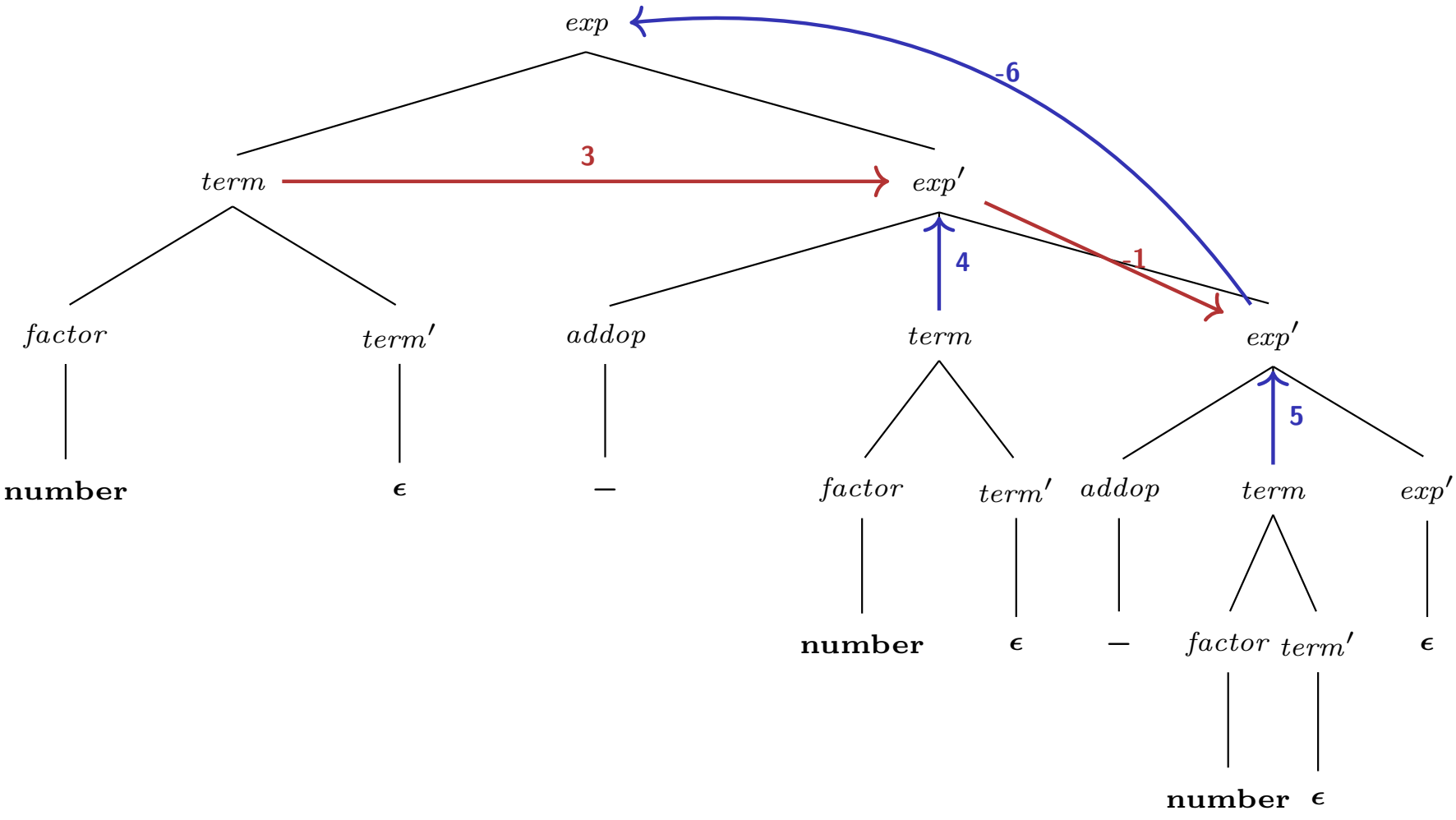
parsed "as"



$3 - (4 - 5)$

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

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



Code to "evaluate" ill-associated such trees correctly

```
1  function exp' (valsofar: int): int;
2  begin
3    if token = '+' or token = '-'
4    then
5      case token of
6        '+': match ('+');
7                valsofar := valsofar + term;
8        '-': match ('-');
9                valsofar := valsofar - term;
10     end case;
11     return exp'(valsofar);
12   else return valsofar
13 end;
```

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

“Designing” the syntax, its parsing, & its AST



trade offs:

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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

AST vs. CST

- once steps 1.–3. are fixed: *parse-trees* fixed!
- parse-trees = *essence* of grammatical derivation process
- often: parse trees only "conceptually" present in a parser
- AST:
  - *abstractions* of the parse trees
  - *essence* of the parse tree
  - actual tree data structure, as output of the parser
  - typically on-the fly: AST built while the parser parses, i.e. while it executes a derivation in the grammar

AST vs. CST/parse tree

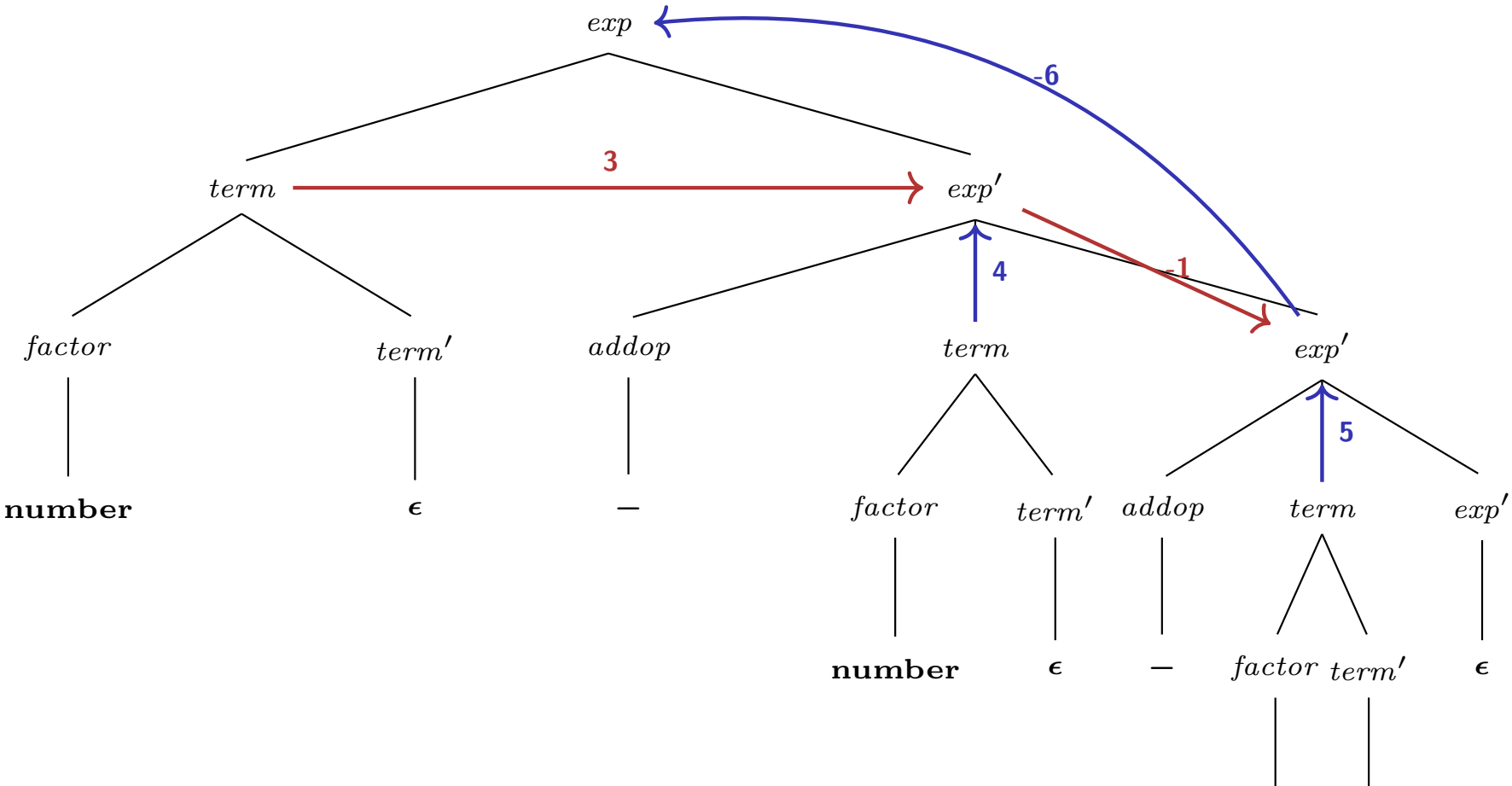
Parser "builds" the AST data structure while "doing" the parse tree



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

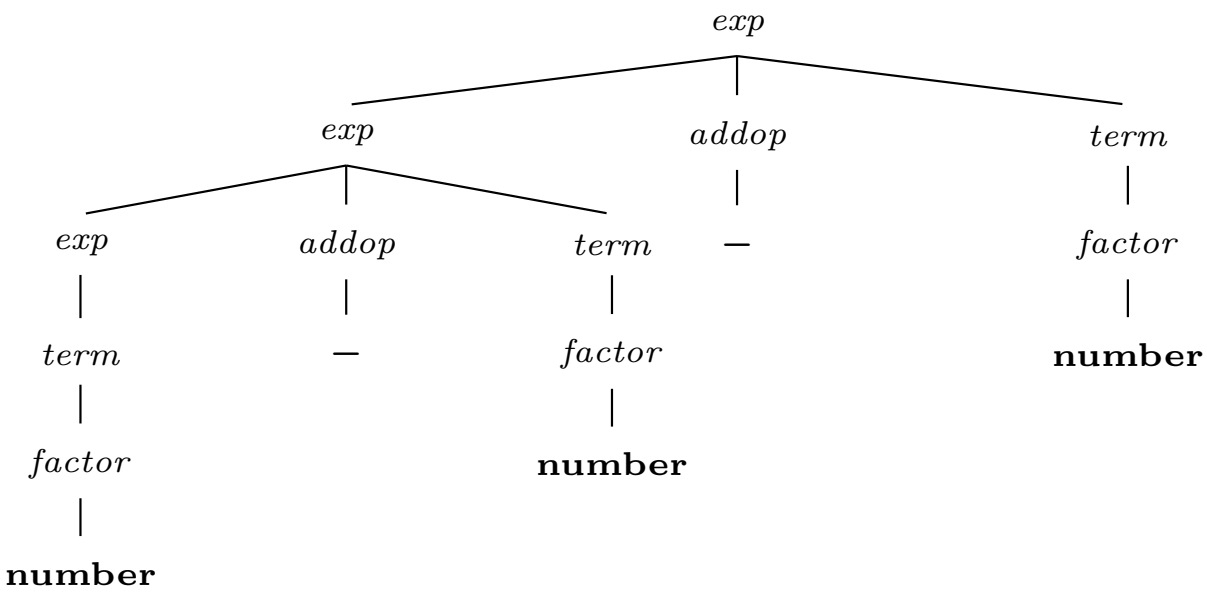
AST: How "far away" from the CST?

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



AST: How “far away” from the CST?

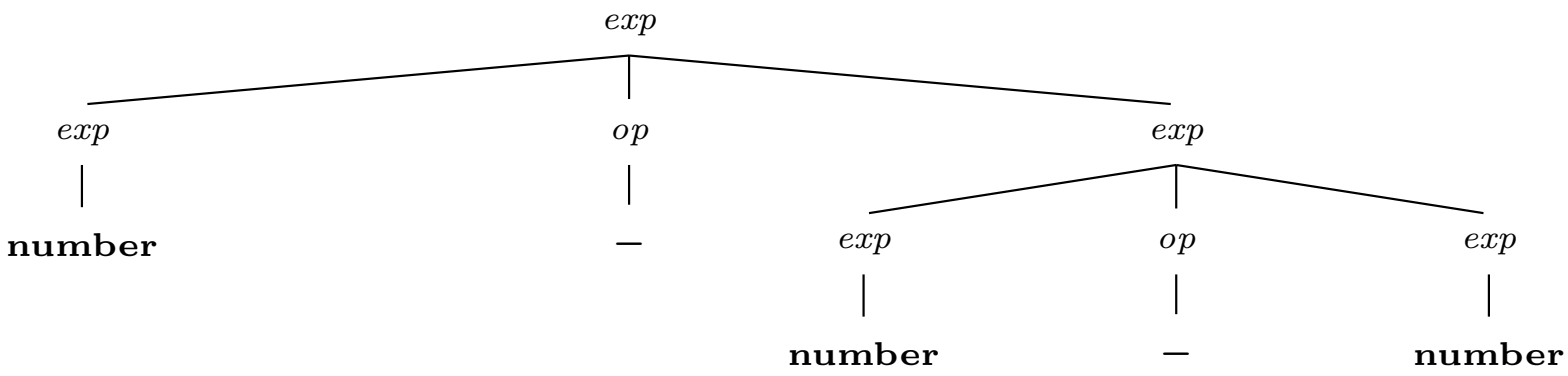
- AST: only thing relevant for later phases $\Rightarrow$ better be *clean* . . .
- AST “=” CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed “weirdly”,



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

AST: How “far away” from the CST?

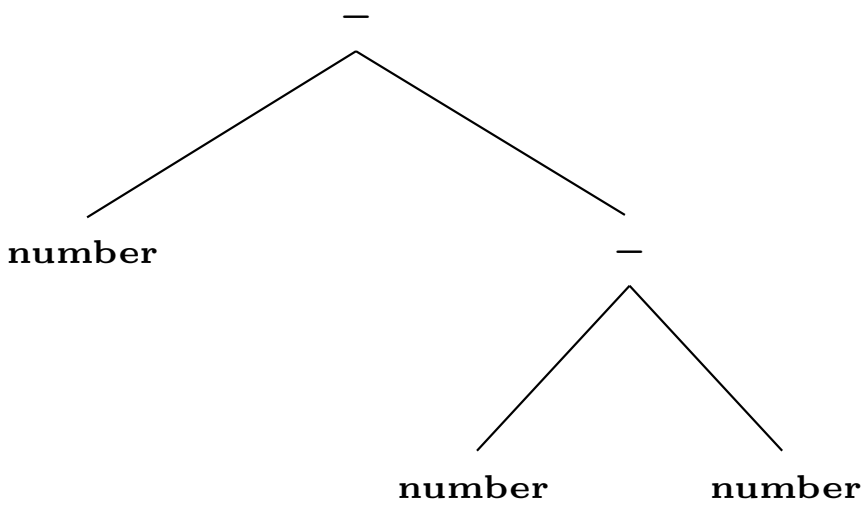
- AST: only thing relevant for later phases $\Rightarrow$ better be *clean* . . .
- AST “=” CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed “weirdly”,



That parse tree looks reasonable clear and intuitive

AST: How “far away” from the CST?

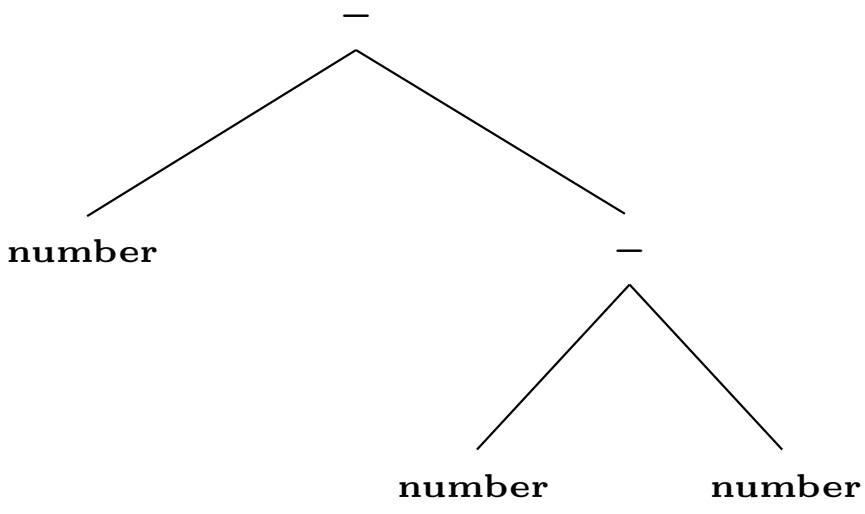
- AST: only thing relevant for later phases $\Rightarrow$ better be *clean* . . .
- AST “=” CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed “weirdly”,



Wouldn’t that be the best AST here?

AST: How "far away" from the CST?

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

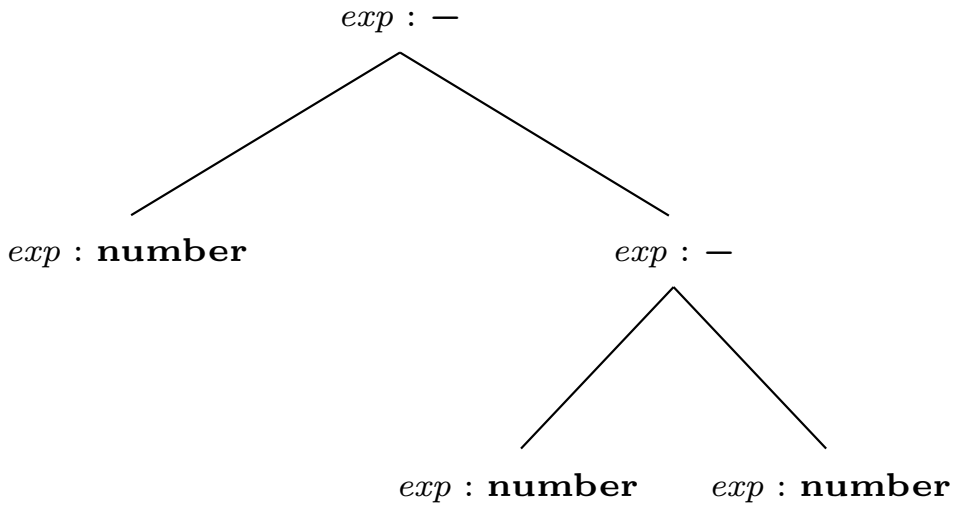


Wouldn't that be the best AST here?

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

AST: How "far away" from the CST?

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



Wouldn't that be the best AST here?

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

This is how it's done (a recipe)

Assume, one has a "non-weird" grammar

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

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

Remember (independent from parsing)

BNF describes trees



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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



Recipe

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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example in Java

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{number}$$
$$op \rightarrow + \mid - \mid *$$

```
1 abstract public class Exp {
2 }
```

```
1 public class BinExp extends Exp {  // exp -> exp op exp
2     public Exp left, right;
3     public Op  op;
4     public BinExp(Exp l, Op o, Exp r) {
5         left=l; op=o; right=r;}
6 }
```

```
1 public class ParentheticExp extends Exp {  // exp -> ( op )
2     public Exp exp;
3     public ParentheticExp(Exp e) {exp = l;}
4 }
```

```
1 public class NumberExp extends Exp {  // exp -> NUMBER
2     public  number;                   // token value
3     public Number(int i) {number = i;}
4 }
```

Example in Java

$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{number}$$
$$op \rightarrow + \mid - \mid *$$

```
1 abstract public class Op {    // non-terminal = abstract
2 }
```

```
1 public class Plus  extends Op {   // op -> "+"
2 }
```

```
1 public class Minus  extends Op {   // op -> "-"
2 }
```

```
1 public class Times extends Op {   // op -> "*"
2 }
```

$3 - (4 - 5)$

```
1 Exp e = new BinExp(
2             new NumberExp(3),
3             new Minus(),
4             new BinExp(new ParentheticExpr(
5                 new NumberExp(4),
6                 new Minus(),
7                 new NumberExp(5))))
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Pragmatic deviations from the recipe

- it's nice to have a guiding principle, but no need to carry it too far ...
- To the very least: the `ParentheticExpr` is completely without purpose: grouping is captured by the tree structure

$\Rightarrow$ that class is *not* needed

- some might prefer an implementation of

$$op \rightarrow + \mid - \mid *$$

as simply integers, for instance arranged like

```
1 public class BinExp extends Exp {  // exp -> exp op exp
2     public Exp left, right;
3     public int  op;
4     public BinExp(Exp l, int o, Exp r) {
5       pos=p; left=l; oper=o; right=r;}
6     public final static int PLUS=0, MINUS=1, TIMES=2;
7 }
```

and used as `BinExpr.PLUS` etc.

Recipe for ASTs, final words:

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

Do it systematically

A clean grammar is the specification of the syntax of the language and thus the parser. It is also a means of communicating with humans what the syntax of the language is, at least communicating with pros, like participants of a compiler course, who of course can read BNF ... A clean grammar is a very systematic and structured thing which consequently *can* and *should* be systematically and cleanly represented in an AST, including judicious and systematic choice of names and conventions (nonterminal exp represented by class `Exp`, non-terminal $stmt$ by class `Stmt` etc)



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

How to produce "something" during RD parsing?

Recursive descent

So far (mostly): RD = top-down (parse-)tree traversal via recursive procedure.[4] Possible outcome: termination or failure.

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

[4] Modulo the fact that the tree being traversed is "conceptual" and not the input of the traversal procedure; instead, the traversal is "steered" by stream of tokens.



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Evaluating an *exp* during RD parsing

```
1  function exp() : int;
2  var temp: int
3  begin
4    temp := term ();          { recursive call }
5    while token = "+" or token = "-"
6      case token  of
7        "+": match ("+");
8             temp := temp + term();
9        "-": match ("-")
10            temp := temp - term();
11     end
12   end
13   return temp;
14 end
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Building an AST: expression

```
1  function exp() : syntaxTree;
2  var temp, newtemp: syntaxTree
3  begin
4    temp := term ();          { recursive call }
5    while token = "+" or token = "-"
6      case token  of
7        "+": match ("+");
8             newtemp := makeOpNode("+");
9             leftChild(newtemp)  := temp;
10            rightChild(newtemp) := term();
11            temp := newtemp;
12       "-": match ("-")
13            newtemp := makeOpNode("-");
14            leftChild(newtemp)  := temp;
15            rightChild(newtemp) := term();
16            temp := newtemp;
17     end
18   end
19   return temp;
20 end
```

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Building an AST: factor

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

```
1  function factor() : syntaxTree;
2  var fact: syntaxTree
3  begin
4    case token of
5      "(": match ("(");
6           fact := exp();
7           match (")");
8      number:
9          match (number)
10         fact := makeNumberNode(number);
11     else : error ...    // fall through
12   end
13   return fact;
14 end
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Building an AST: conditionals

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

```
1  function ifStmt() : syntaxTree;
2  var temp: syntaxTree
3  begin
4    match ("if");
5    match ("(");
6    temp := makeStmtNode("if")
7    testChild(temp) := exp();
8    match (")");
9    thenChild(temp) := stmt();
10   if   token = "else"
11   then match "else";
12        elseChild(temp) := stmt();
13   else elseChild(temp) := nil;
14   end
15   return temp;
16 end
```



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Building an AST: remarks and "invariant"



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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LL(1) parsing

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

LL(1) parsing principle

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

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

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Construction of the parsing table

Table recipe

1. If $A \to \alpha \in P$ and $\alpha \Rightarrow^* \mathbf{a}\beta$, then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$
2. If $A \to \alpha \in P$ and $\alpha \Rightarrow^* \boldsymbol{\epsilon}$ and $S\,\$ \Rightarrow^* \beta A \mathbf{a} \gamma$ (where $\mathbf{a}$ is a token (=non-terminal) *or* $\$$), then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$

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

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

1. If $\mathbf{a} \in First(\alpha)$, 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 $M[A, \mathbf{a}]$.



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example: if-statements

- grammars is left-factored and not left recursive

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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

M[N, T]	**if**	**other**	**else**	0	1	**$**
statement	*statement* → *if-stmt*	*statement* → **other**				
if-stmt	*if-stmt* → **if** (*exp*) *statement* *else-part*					
else-part			*else-part* → **else** *statement* *else-part* → ε			*else-part* → ε
exp				*exp* → **0**	*exp* → **1**	

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



LL(1) table-based algo

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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

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



Expressions

Original grammar

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

	First	*Follow*
exp	$(, \mathbf{number}$	$\$,)$
exp'	$+, -, \epsilon$	$\$,)$
$addop$	$+, -$	$(, \mathbf{number}$
$term$	$(, \mathbf{number}$	$\$,), +, -$
$term'$	$*, \epsilon$	$\$,), +, -$
$mulop$	$*$	$(, \mathbf{number}$
$factor$	$(, \mathbf{number}$	$\$,), +, -, *$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Expressions

Original grammar

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

left-recursive $\Rightarrow$ not LL(k)

	First	*Follow*
exp	$(, \mathbf{number}$	$\mathbf{\$},)$
exp'	$+, -, \epsilon$	$\mathbf{\$},)$
$addop$	$+, -$	$(, \mathbf{number}$
$term$	$(, \mathbf{number}$	$\mathbf{\$},), +, -$
$term'$	$*, \epsilon$	$\mathbf{\$},), +, -$
$mulop$	$*$	$(, \mathbf{number}$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Expressions

Left-rec removed

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

	First	*Follow*
exp	$(, \mathbf{number}$	$\$,)$
exp'	$+, -, \epsilon$	$\$,)$
$addop$	$+, -$	$(, \mathbf{number}$
$term$	$(, \mathbf{number}$	$\$,), +, -$
$term'$	$*, \epsilon$	$\$,), +, -$
$mulop$	$*$	$(, \mathbf{number}$
$factor$	$(, \mathbf{number}$	$\$,), +, -, *$



Expressions: LL(1) parse table

M[N, T]	(	**number**	)	+	-	*	$
exp	*exp* → *term exp′*	*exp* → *term exp′*					
exp′			*exp′* → ε	*exp′* → *addop term exp′*	*exp′* → *addop term exp′*		*exp′* → ε
addop				*addop* → +	*addop* → -		
term	*term* → *factor term′*	*term* → *factor term′*					
term′			*term′* → ε	*term′* → ε	*term′* → ε	*term′* → *mulop factor term′*	*term′* → ε
mulop						*mulop* → *	
factor	*factor* → (*exp*)	*factor* → **number**					





Section

Bottom-up parsing

Chapter 4 “Parsing”
Course “Compiler Construction”
Martin Steffen
Spring 2023

Bottom-up parsing: intro

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

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

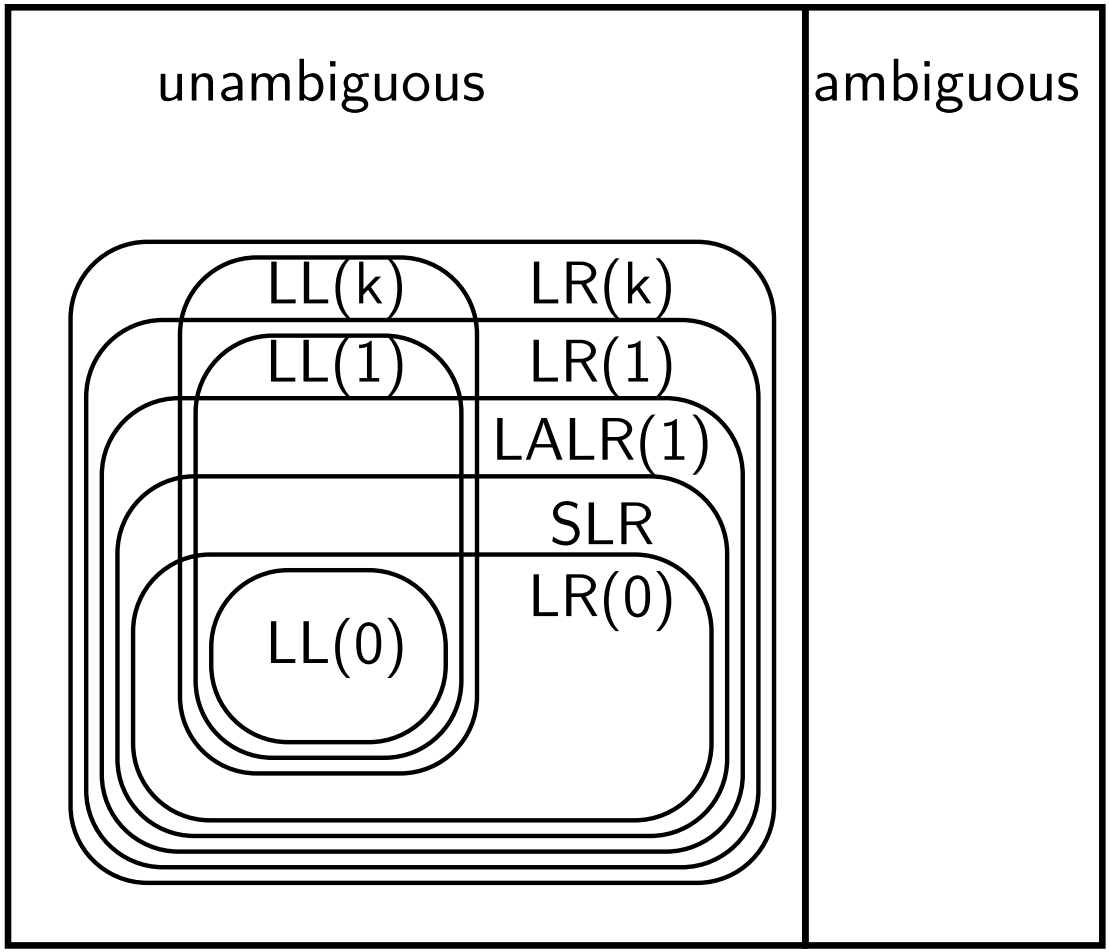
SLR(1)
- expressive enough for most grammars for standard PLs
- same number of states as LR(0)

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)

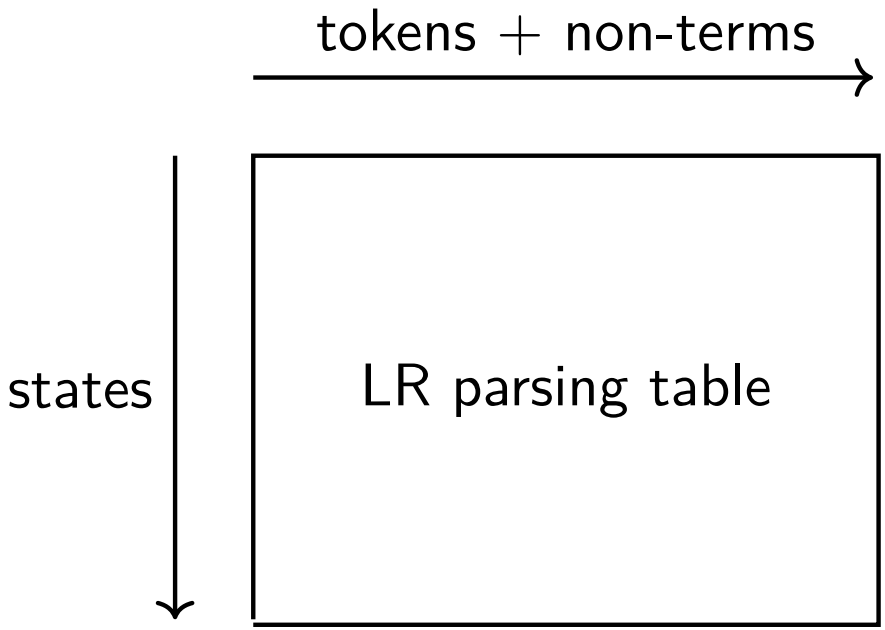




Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LR-parsing and its subclasses

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





Example grammar

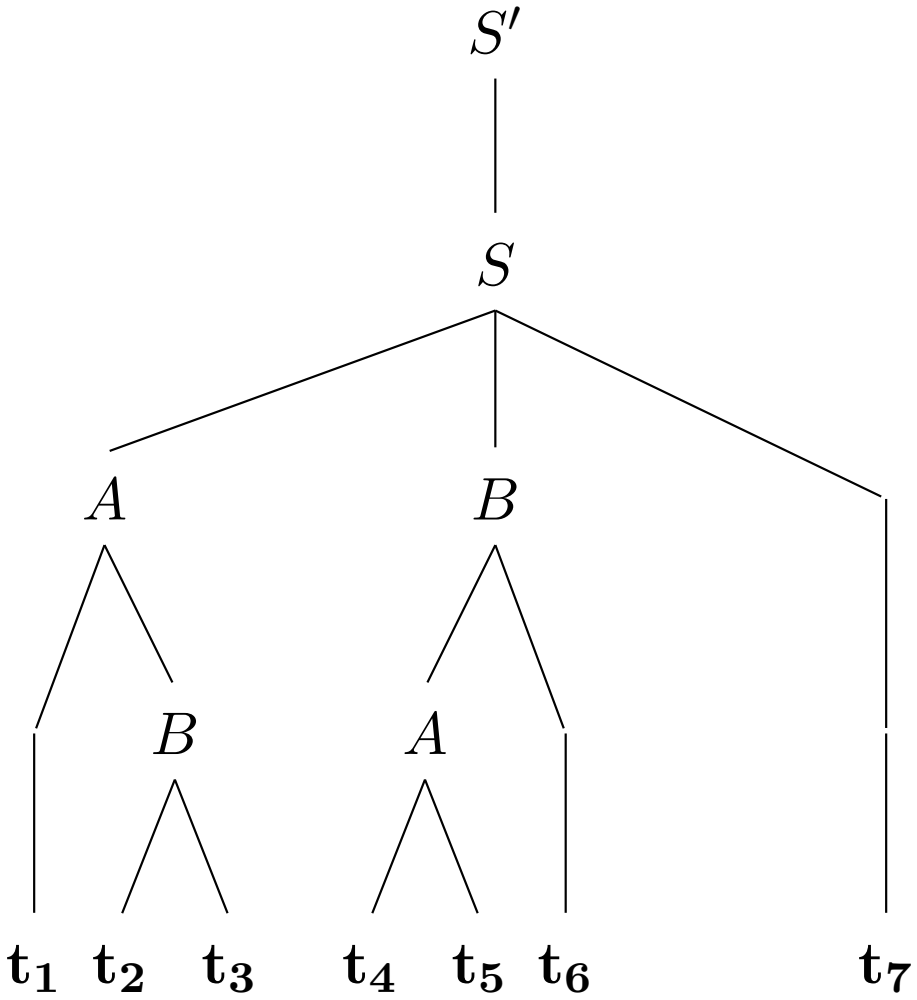
$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & AB\mathbf{t_7} \mid \ldots \\
A & \rightarrow & \mathbf{t_4}\mathbf{t_5} \mid \mathbf{t_1}B \mid \ldots \\
B & \rightarrow & \mathbf{t_2}\mathbf{t_3} \mid A\mathbf{t_6} \mid \ldots
\end{array}
$$

- 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 of a production
  - routinely add another "extra" start-symbol (here S')



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Parse tree for $t_1 \ldots t_7$



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

Potentially puzzling question at first sight:

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

Discussion

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

Right-sentential form: right-most derivation

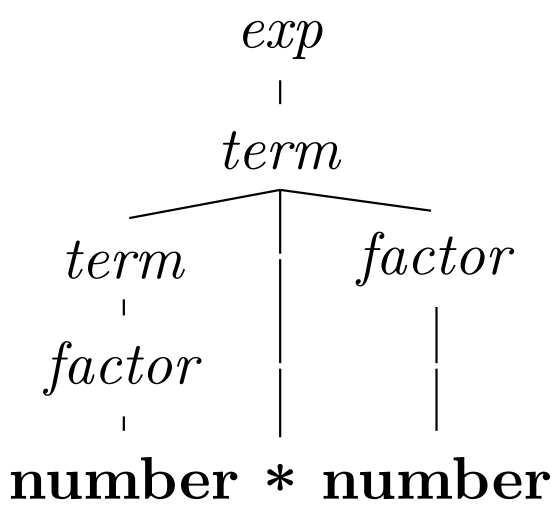
$$S \Rightarrow_r^* \alpha$$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example expression grammar (from before)

$$
\begin{array}{rcl}
exp & \rightarrow & exp\ addop\ term \mid term \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & term\ mulop\ factor \mid factor \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
\tag{9}
$$





Bottom-up parse: Growing the parse tree

$$\mathbf{number} * \mathbf{number}$$

$$\underline{\mathbf{number}} * \mathbf{number}$$

Bottom-up parse: Growing the parse tree

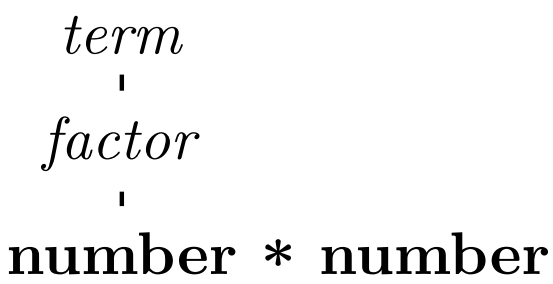


Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

$$\begin{array}{c} factor \\ | \\ \mathbf{number} * \mathbf{number} \end{array}$$

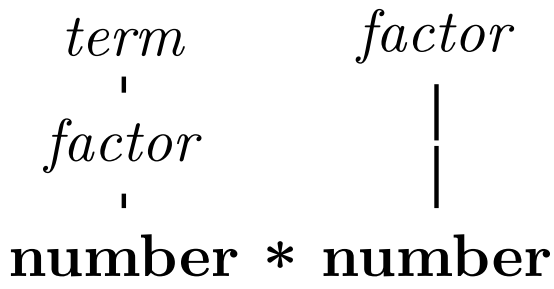
$$\underline{\mathbf{number}} * \mathbf{number} \quad \hookrightarrow \quad \underline{factor} * \mathbf{number}$$

Bottom-up parse: Growing the parse tree



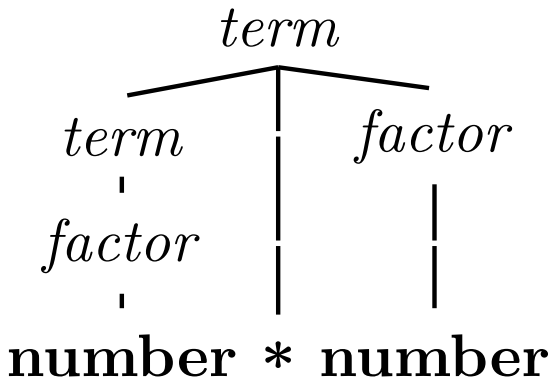
$$\underline{\mathbf{number}} * \mathbf{number} \quad \hookrightarrow \quad \underline{factor} * \mathbf{number}$$
$$\hookrightarrow \quad term * \underline{\mathbf{number}}$$

Bottom-up parse: Growing the parse tree



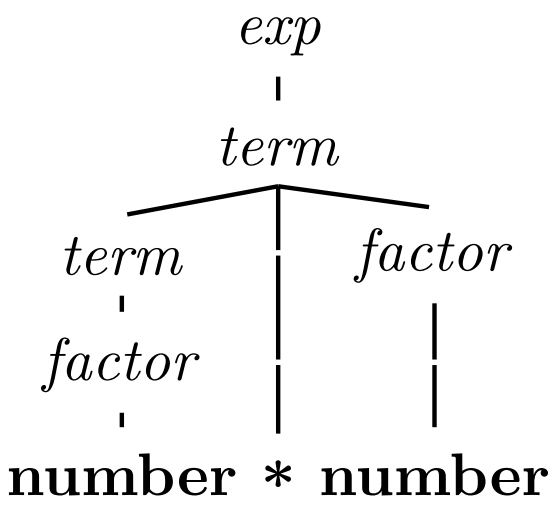
$$\underline{\mathbf{number}} * \mathbf{number} \hookrightarrow \underline{factor} * \mathbf{number} \hookrightarrow term * \underline{\mathbf{number}} \hookrightarrow \underline{term * factor}$$

Bottom-up parse: Growing the parse tree



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

Bottom-up parse: Growing the parse tree



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



Reduction in reverse = right derivation

Reduction

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

Right derivation

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

Underlined entity

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



Handle

Definition (Handle)

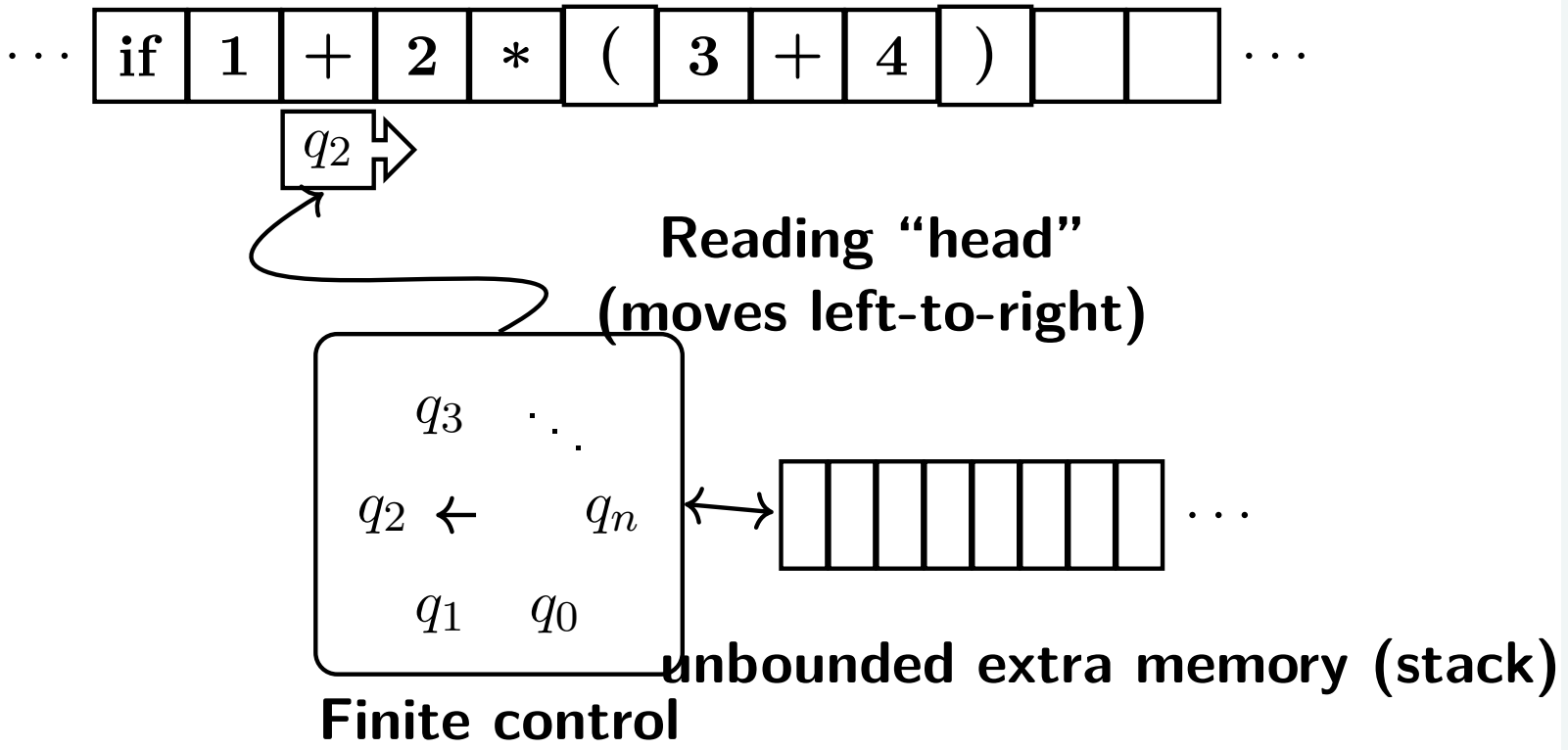
Assume $S \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha\beta w$. A production $A \to \beta$ at position k following α is a *handle of* $\alpha\beta w$. We write $\langle A \to \beta, k \rangle$ for such a handle.

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 (β the part touched by reduction step).
- the $\Rightarrow_r$ -derivation-step *in reverse*:
  - one reduce-step in the LR-parser-machine
  - adding (implicitly in the LR-tree) a new parent to children β (= bottom-up!)
- "handle"-part β can be *empty* ($= \epsilon$)



Schematic picture of parser machine (again)



General LR "parser machine" configuration

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



Schematic run (reduction: from top to bottom)

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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").

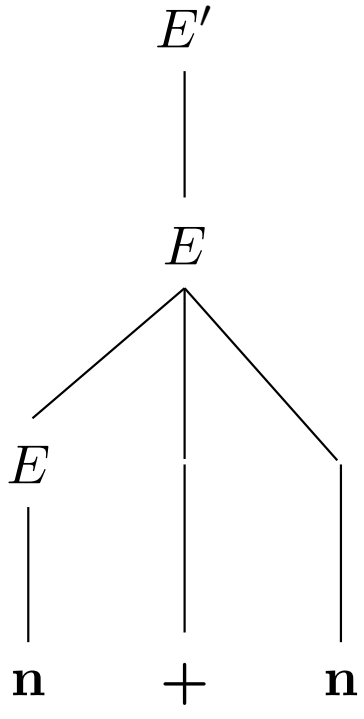
Explanations

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



Example: LR parse for "+" (given the tree)

$$E' \rightarrow E$$
$$E \rightarrow E + \mathbf{n} \mid \mathbf{n}$$



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

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

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

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

Example with ϵ-transitions: parentheses

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

side remark: unlike previous grammar, here:

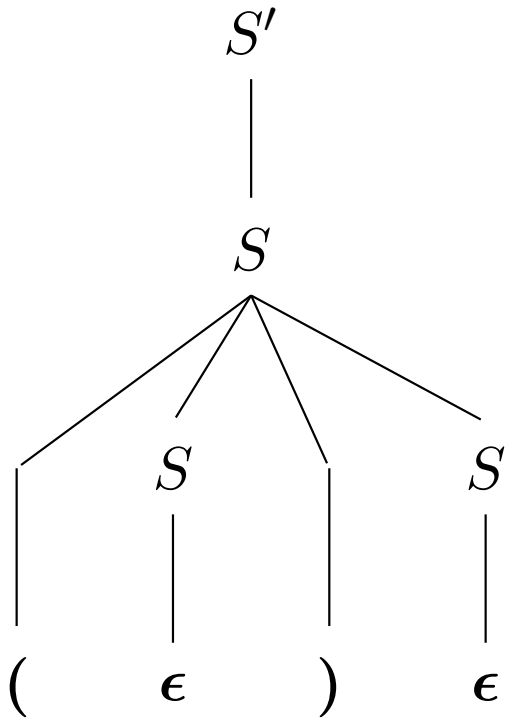
- production with *two* non-terminals on the right

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Parentheses: 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

Right-sentential form: right-most derivation

$$S \Rightarrow_r^* \alpha$$

Explanation

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

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

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

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

Rest

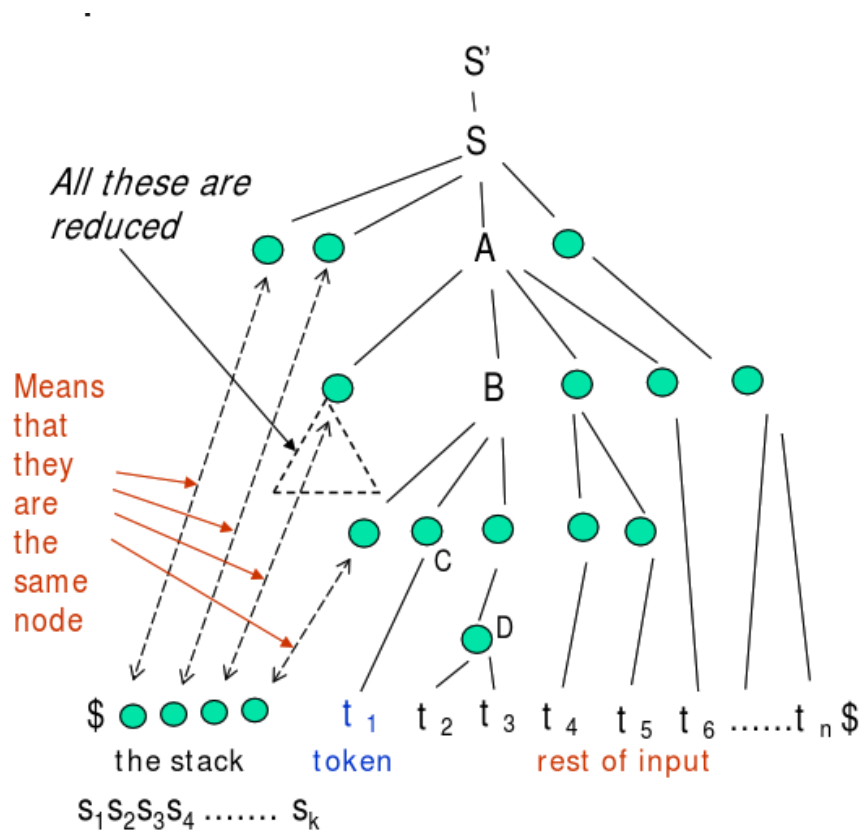
Viable prefixes of right-sentential forms and handles

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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 until now:
  - bottom-up tree building as reverse right-most derivation,
  - stack vs. input,
  - shift and reduce steps
- however: 1 ingredient missing: next step of the engine may depend on
  - top of the stack ("handle")
  - look ahead on the input (but not for LL(0))
  - and: current state of the machine



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

But what are the states of an LR-parser?



General idea (change the slide):

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!

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LR(0) parsing as easy pre-stage

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LR(0) items

LR(0) item

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

Rest

- . : "meta-symbol" (not part of the production)

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

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

complete and initial items

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Grammar for parentheses: 3 productions

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

8 items

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

Remarks

:BEAMER$_{\text{env}}$: ignoreheading :END:

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Grammar for addition: 3 productions

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + \mathbf{number} \mid \mathbf{number}
\end{array}
$$

(coincidentally also:) 8 items

$$
\begin{array}{lcl}
E' & \rightarrow & .E \\
E' & \rightarrow & E. \\
E & \rightarrow & .E + \mathbf{number} \\
E & \rightarrow & E. + \mathbf{number} \\
E & \rightarrow & E + .\mathbf{number} \\
E & \rightarrow & E + \mathbf{number}. \\
E & \rightarrow & .\mathbf{number} \\
E & \rightarrow & \mathbf{number}.
\end{array}
$$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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(!))

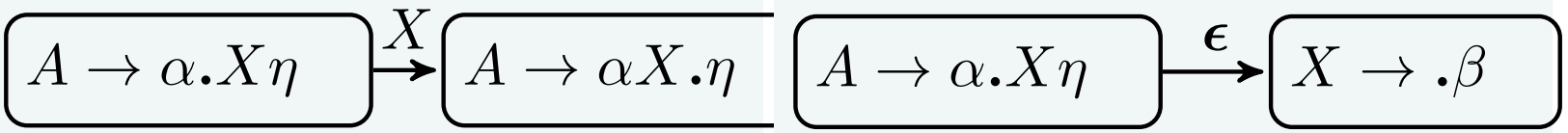


Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

State transitions of the NFA

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

Terminal or non-terminal	ϵ $(X \to \beta)$



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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Transitions for non-terminals and ϵ

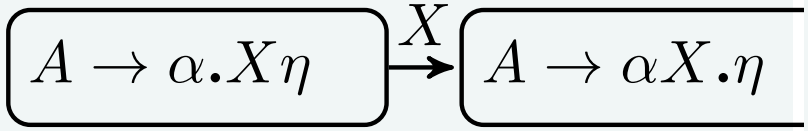
- so far: we never pushed a non-terminal from the input to the stack, we replace in a reduce-step the right-hand side β side by a left-hand side X (non-term)
- but: replacement in a reduce steps can be seen as
  1. pop right-hand side off the stack,
  2. instead, "assume" corresponding non-terminal on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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

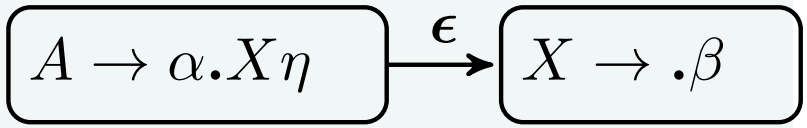
Transitions (repeated)

Terminal or non-terminal



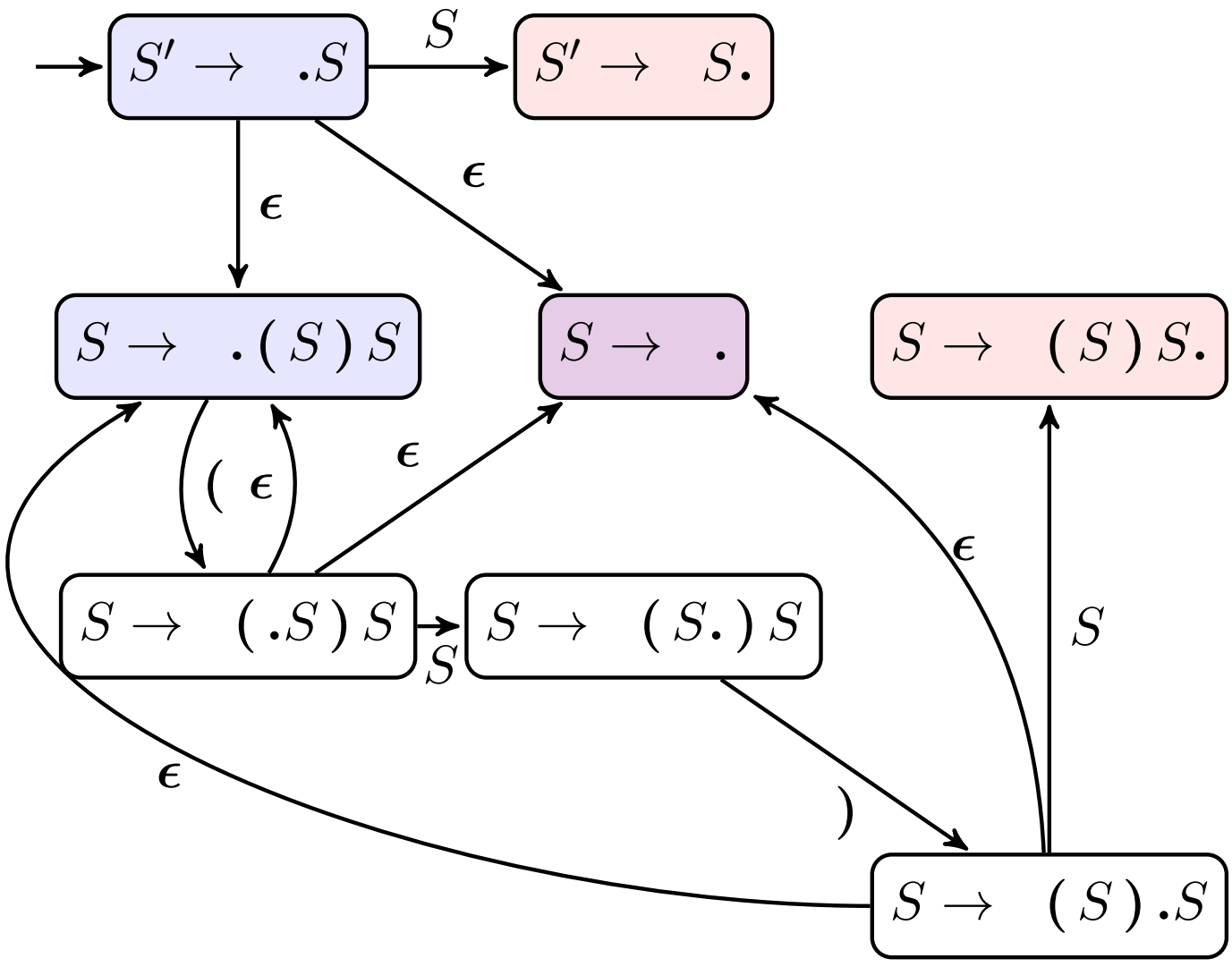
Epsilon (X: non-terminal here)

Given production $X \to \beta$:



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

NFA: parentheses



Initial and final states

initial states:

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

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

final states:

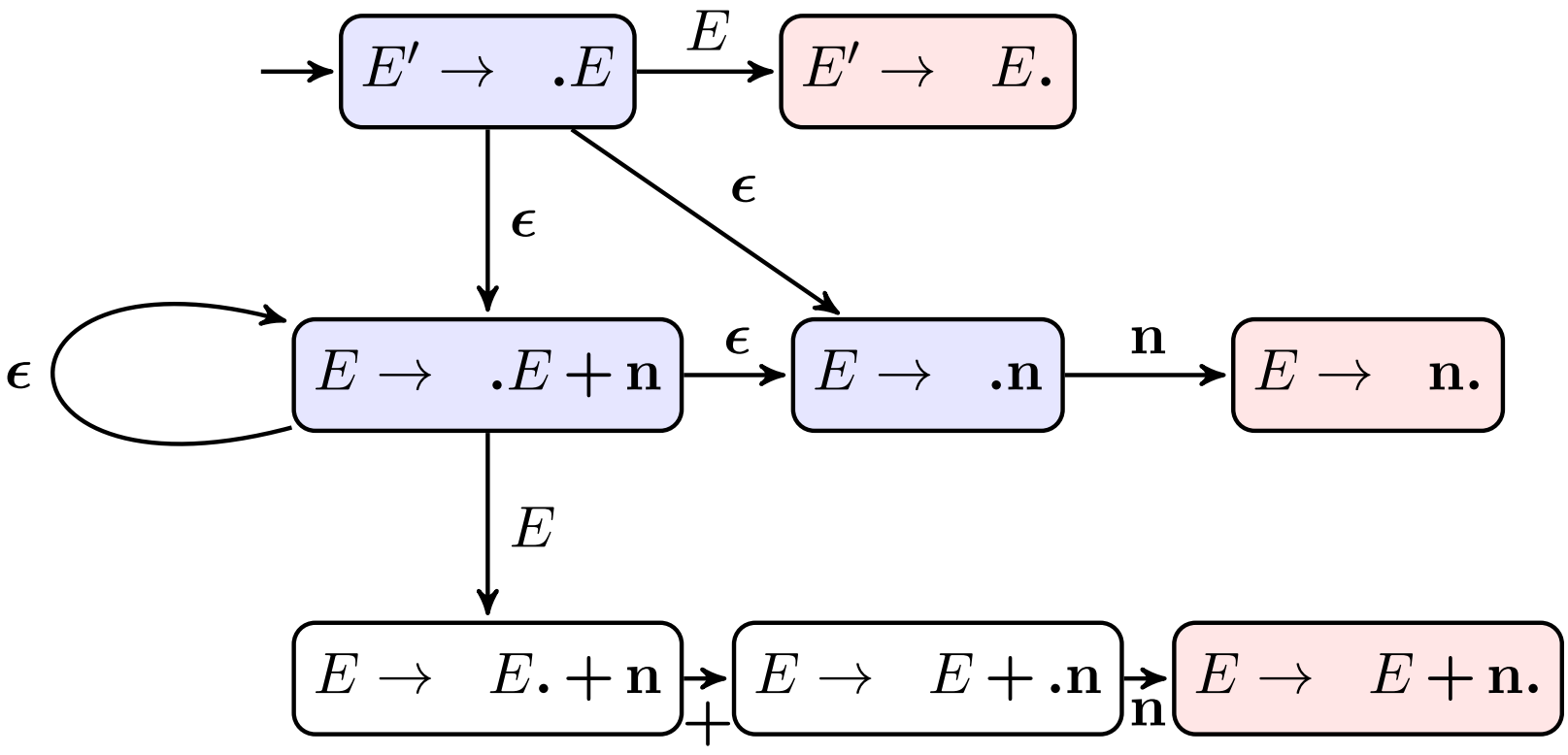
acceptance condition of the *overall* machine: a bit more complex

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

NFA: addition



Determinizing: from NFA to DFA

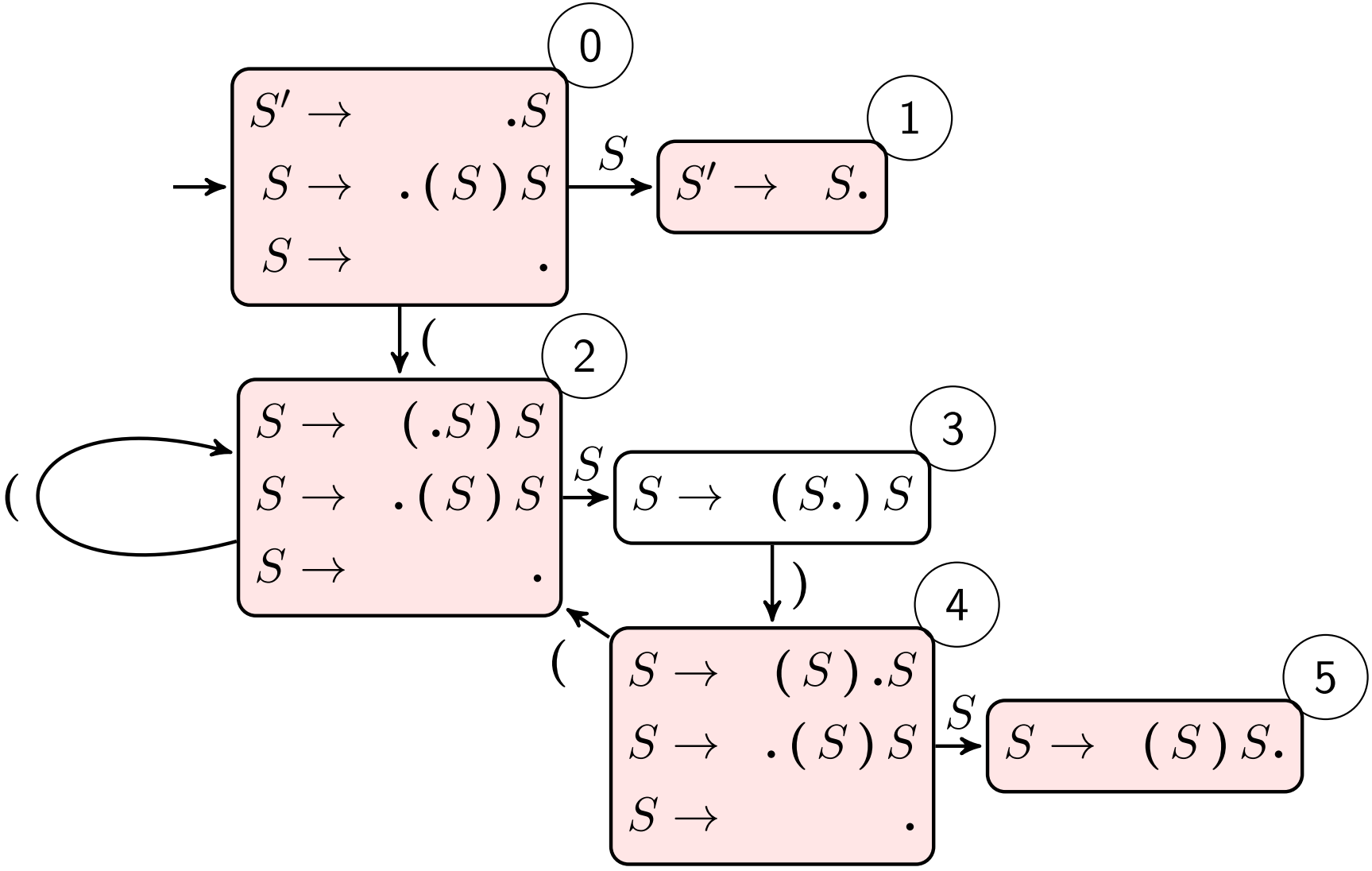
- standard subset-construction[5]
- states then contain *sets* of items
- important: ϵ-closure
- also: *direct* construction of the DFA possible

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

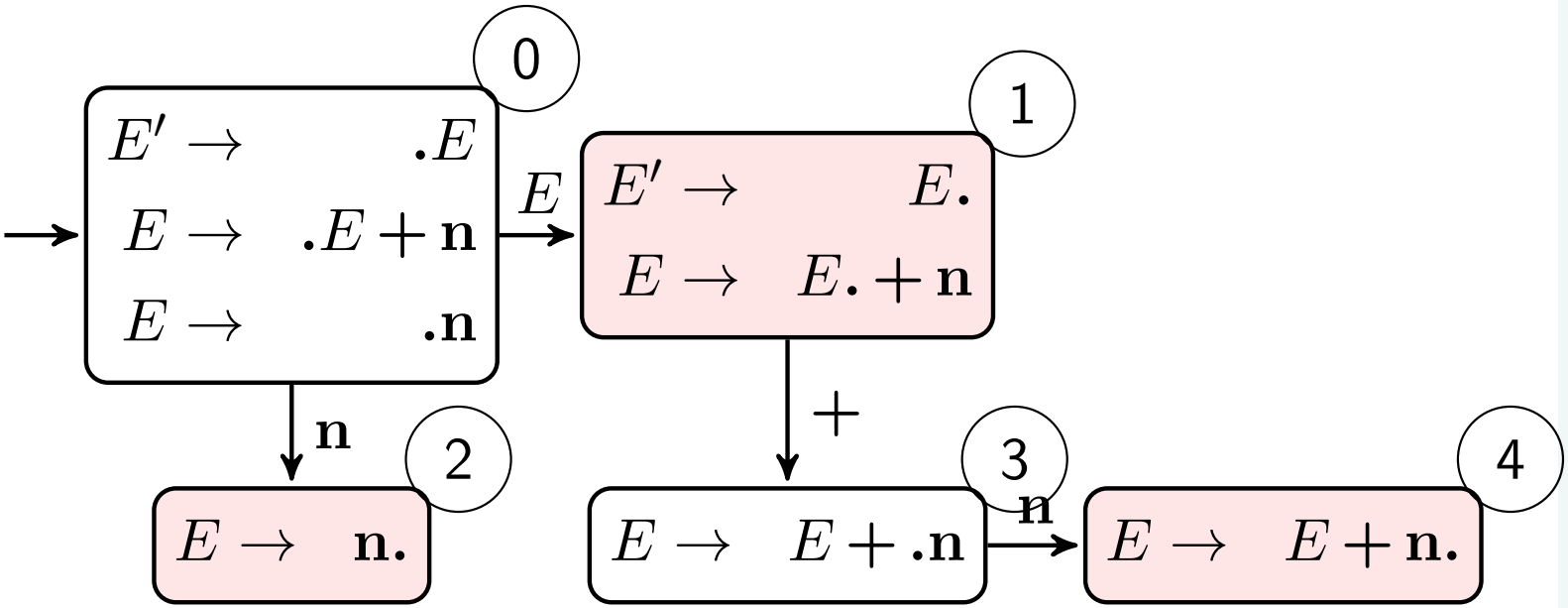


Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

DFA: parentheses



DFA: addition



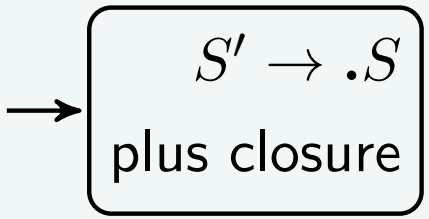
Direct construction of an LR(0)-DFA

- quite easy: just build in the closure directly...

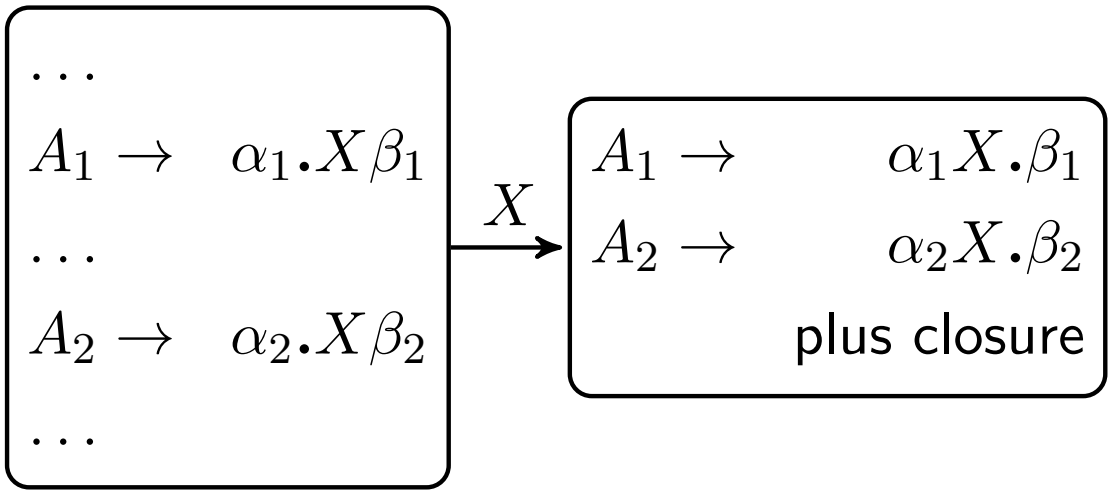
ϵ-closure

- if $A \to \alpha.B\gamma$ is an item in a state where
- there are productions $B \to \beta_1 \mid \beta_2 \ldots$ then
- add items $B \to .\beta_1$, $B \to .\beta_2 \ldots$ 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 $A \rightarrow \alpha . X \beta$ must be included in the post-state
- and all others (indicated by "...") in the pre-state: not included

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 ....



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Stack contents and state of the automaton

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

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

  $\Rightarrow$ crucial when determining *reaction*



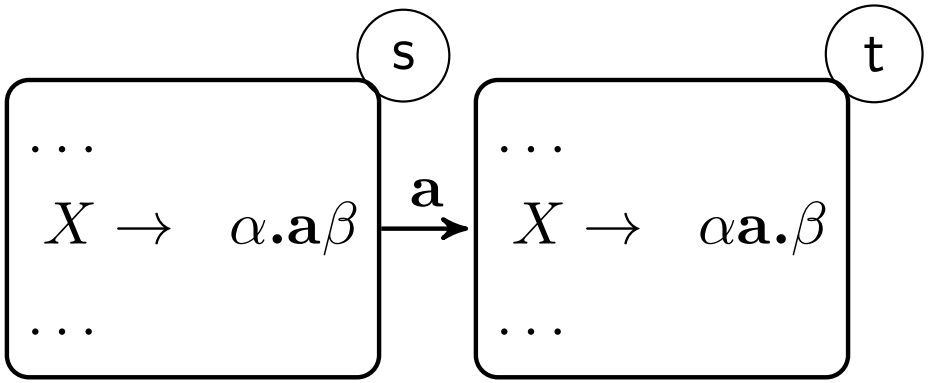
Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

State transition allowing a shift

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

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

- construction thus has transition as follows

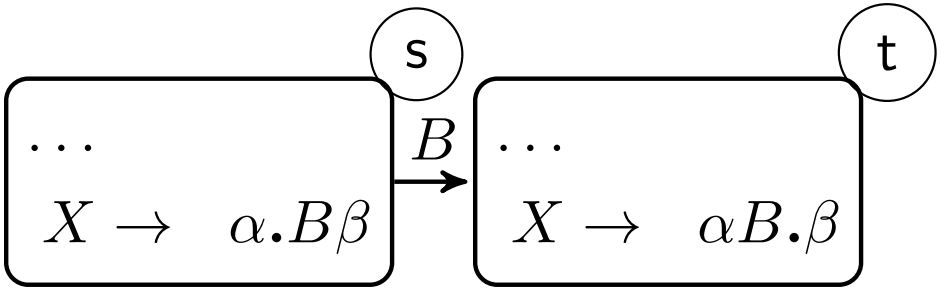


- shift is possible
- if shift is *the* correct operation and $\mathbf{a}$ is terminal symbol corresponding to the current token: state afterwards $= t$



State transition: analogous for non-term's

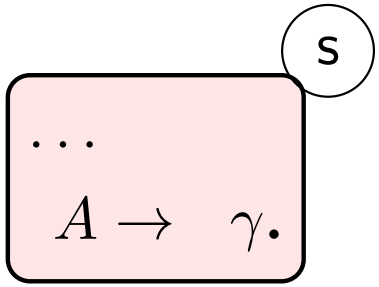
$X \to \alpha.B\beta$



- "goto = shift for non-terms"
- intuition: "second half of a reduce step"

State (not transition) where a reduce is possible

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



- 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:
  - new top state!
  - remember the "goto-transition" (shift of a non-terminal)



Remarks: states, transitions, and reduce steps

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

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

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

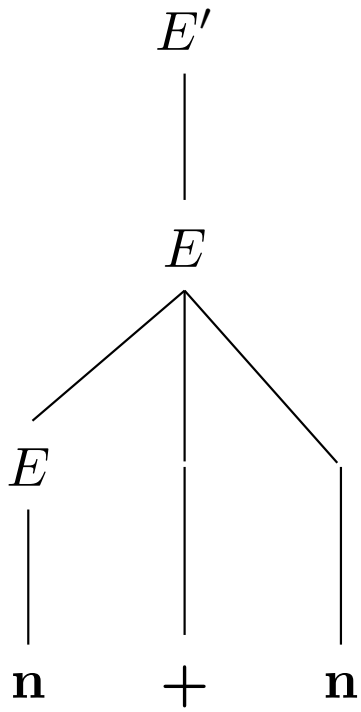
- if the (rhs of the) handle is *removed* from top stack $\Rightarrow$
  - "go back to the (top) state before that handle had been added": *no edge for that*
- later: stack notation simply remembers the state as part of its configuration



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example: LR parsing for addition (given the tree)

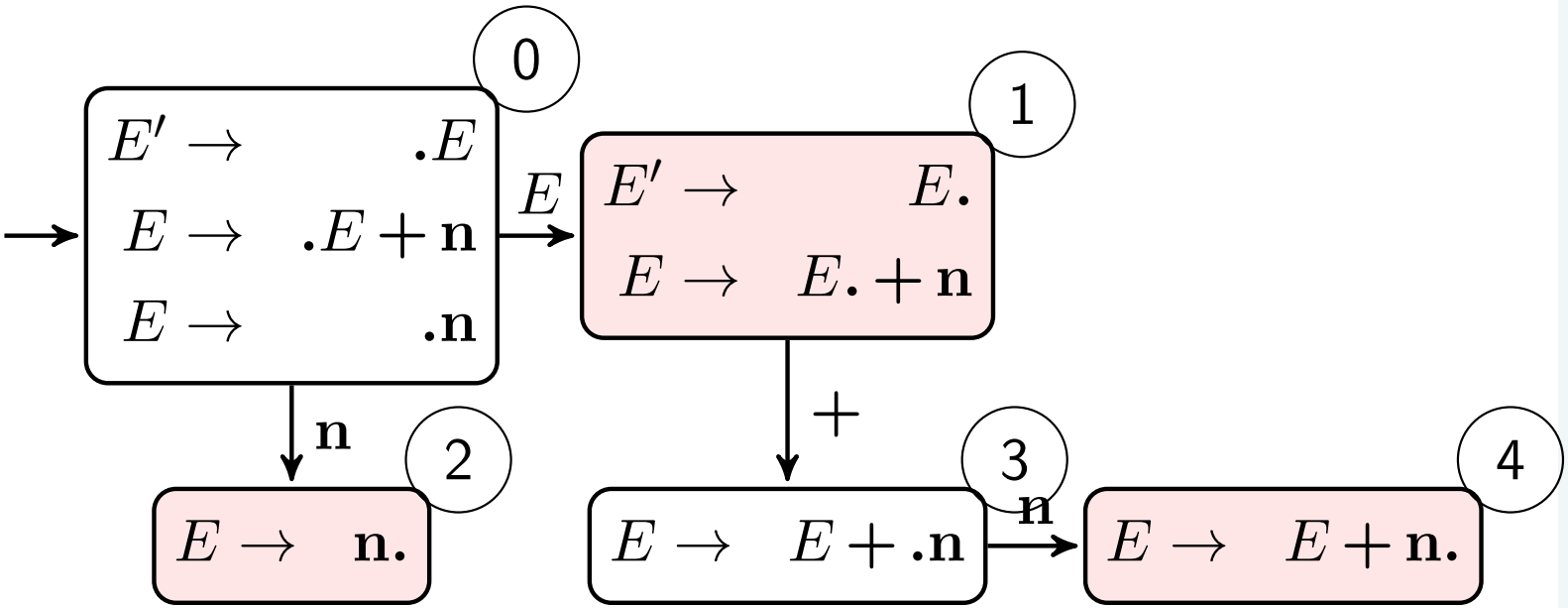
$$
\begin{array}{rcl}
E' & \rightarrow & E \\
E & \rightarrow & E + \mathbf{n} \mid \mathbf{n}
\end{array}
$$



	parse stack	input	action
1	$	n + n $	shift
2	$ n	+ n $	red:. $E \rightarrow \mathbf{n}$
3	$ E	+ n $	shift
4	$ E +	n $	shift
5	$ E + n	$	red. $E \rightarrow E + \mathbf{n}$
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) grammars



LR(0) grammar

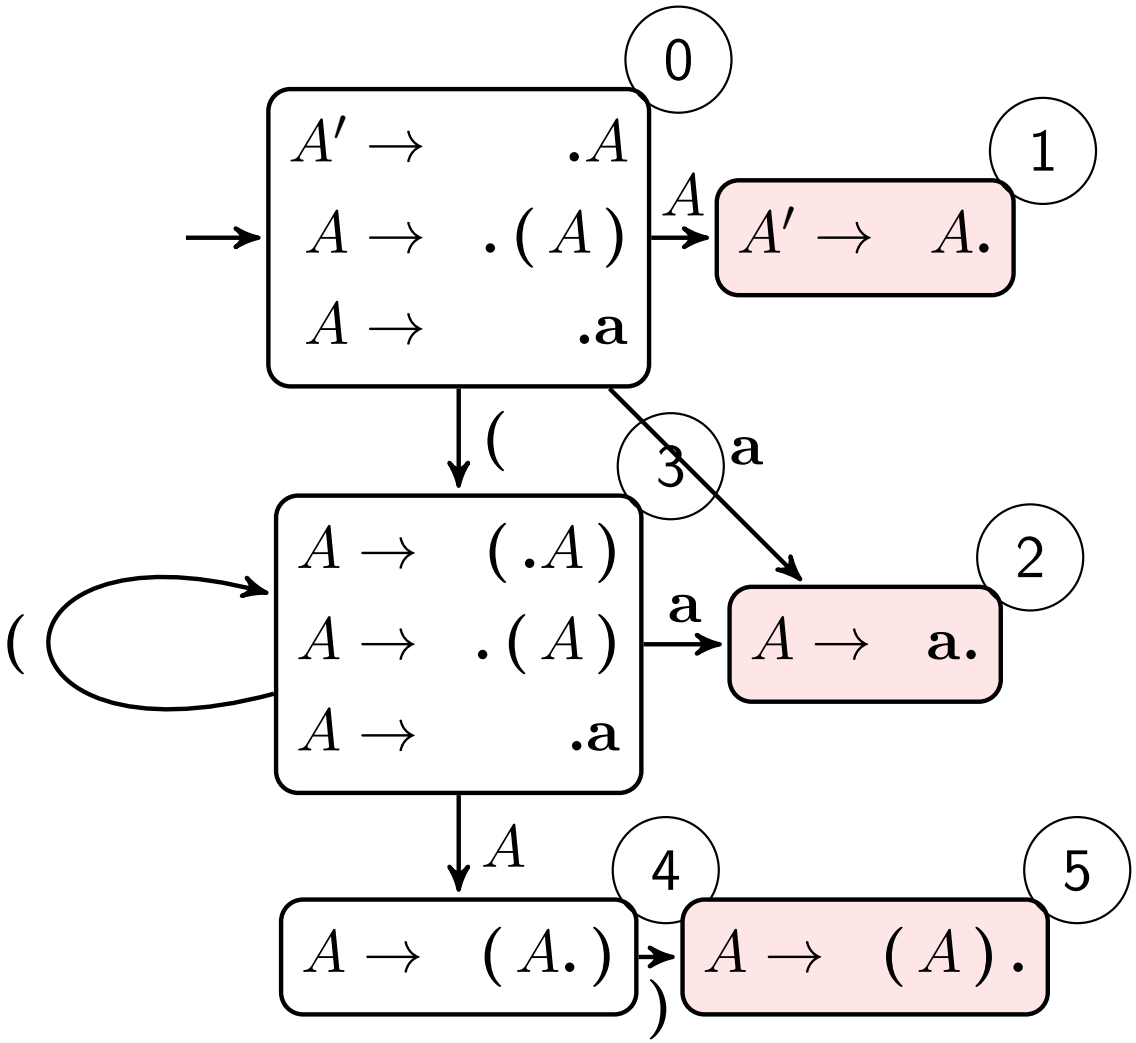
The top-state alone determines the next step.

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

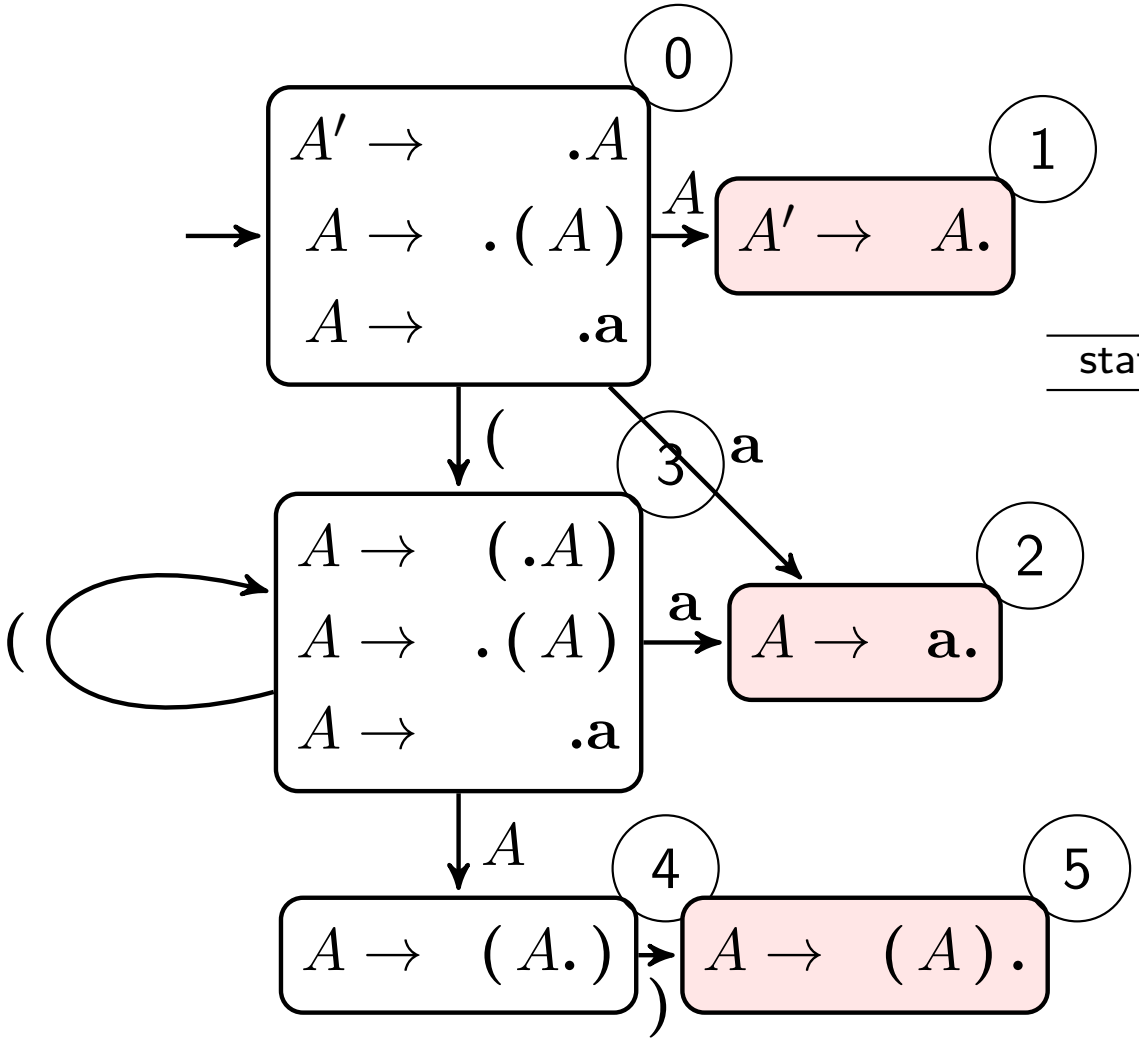
Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Simple parentheses

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

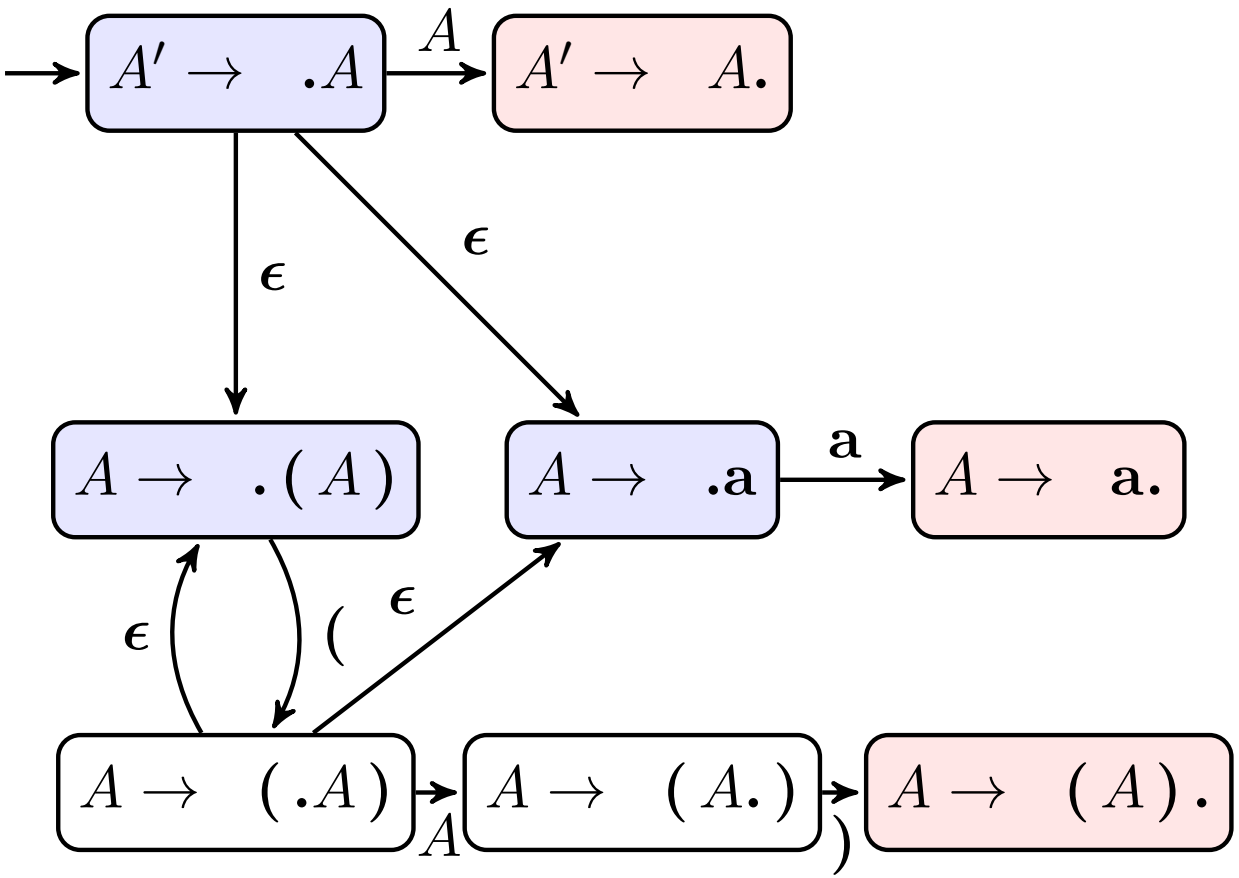


Simple parentheses is LR(0)



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

NFA for simple parentheses (bonus slide)



Parsing table for an LR(0) grammar

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

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

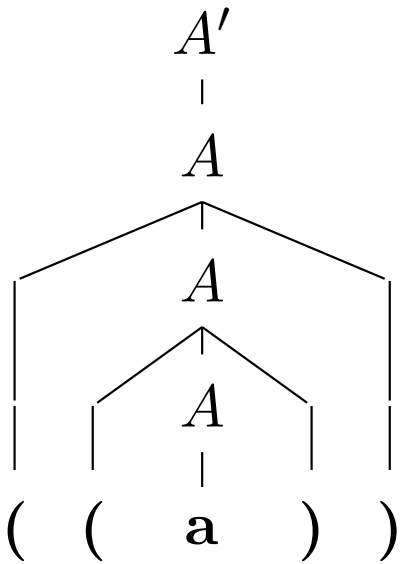


Parsing of ((a))

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

Parsing of erroneous input

- empty slots it the table: "errors"

stage	parsing stack	input	action
1	$\$_0$	$(\ (\ \mathbf{a}\ )\ \$$	shift
2	$\$_0(_3$	$(\ \mathbf{a}\ )\ \$$	shift
3	$\$_0(_3(_3$	$\mathbf{a}\ )\ \$$	shift
4	$\$_0(_3(_3\mathbf{a}_2$	$)\ \$$	reduce $A \rightarrow \mathbf{a}$
5	$\$_0(_3(_3A_4$	$)\ \$$	shift
6	$\$_0(_3(_3A_4)_5$	$\$$	reduce $A \rightarrow (\ A\ )$
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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LR(0) parsing algo, given DFA

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

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

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

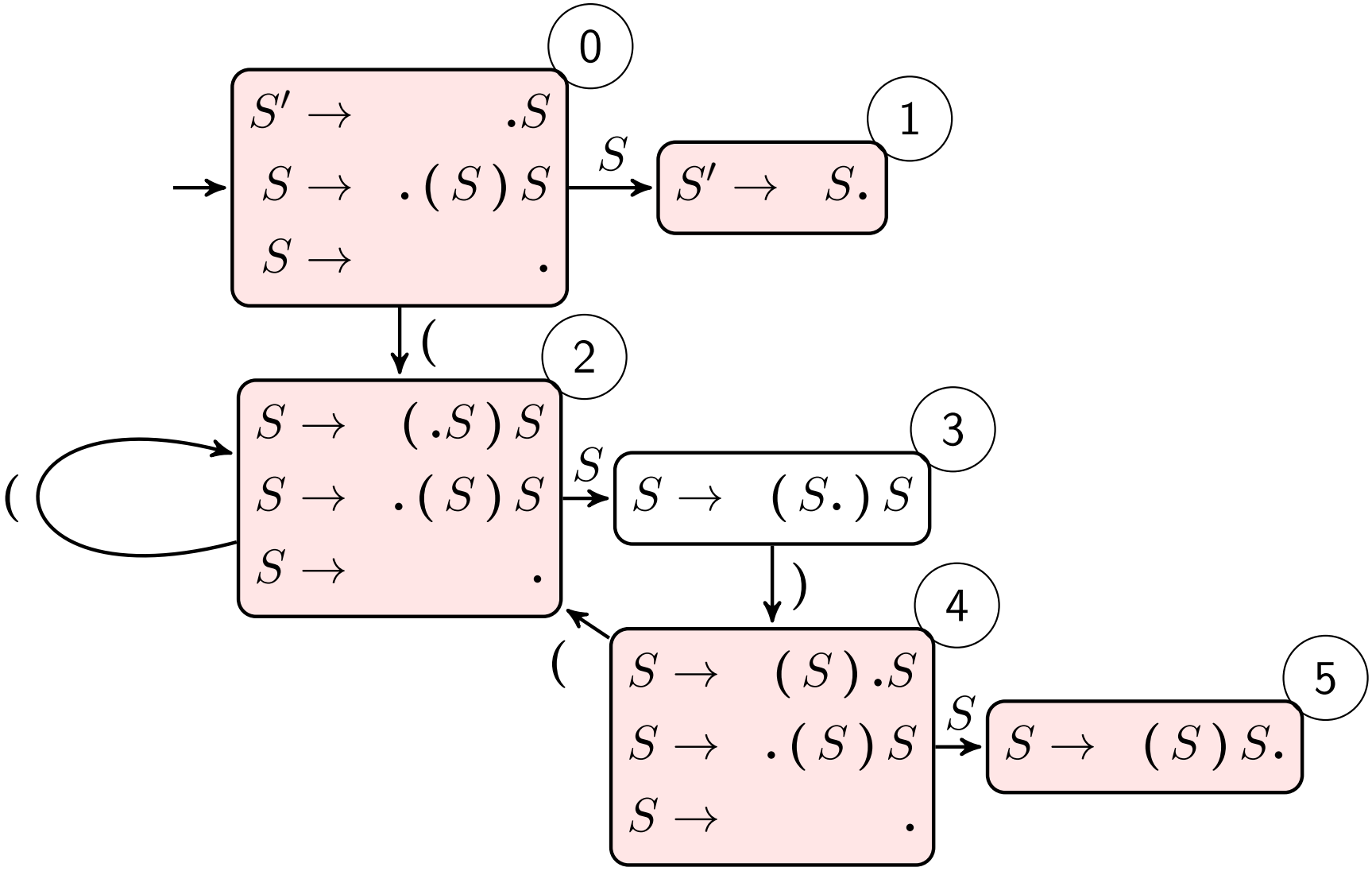
     back up: assume to be in state u which is *now* head state

     push: push A to the stack, new head state t where $u \xrightarrow{A} t$ (in the DFA)



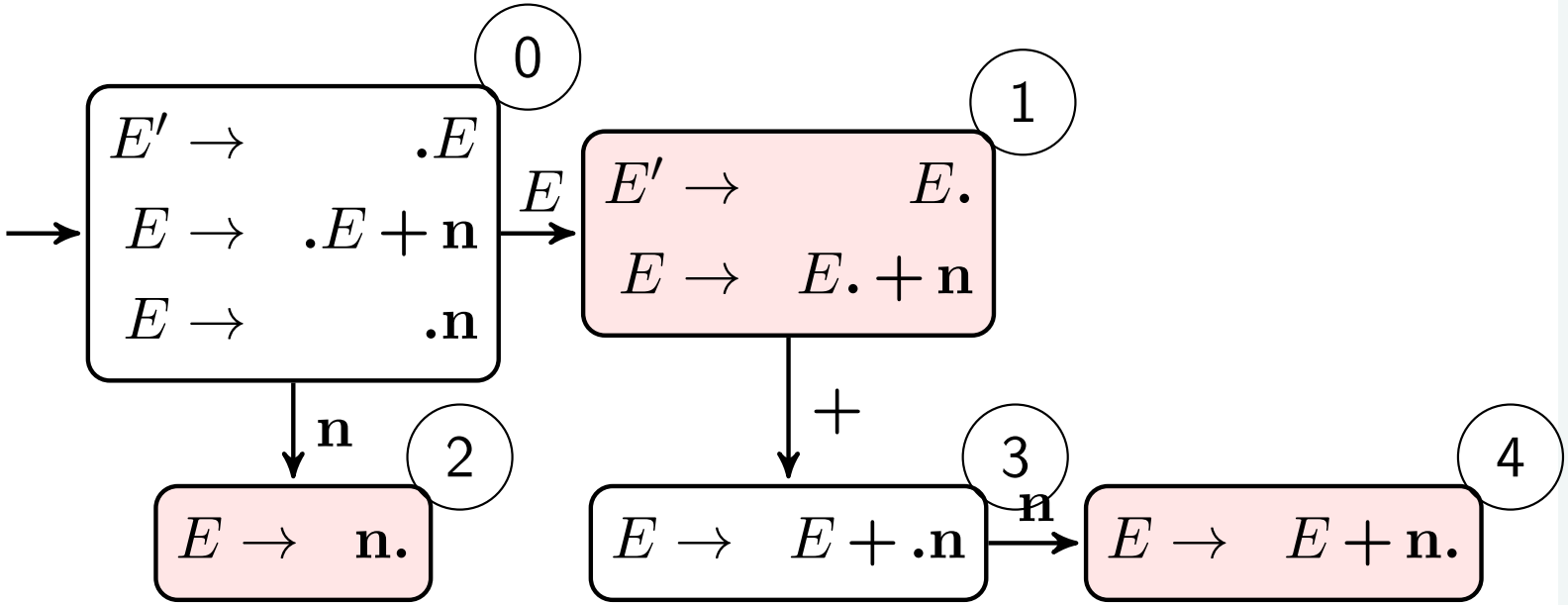
DFA parentheses again: LR(0)?

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

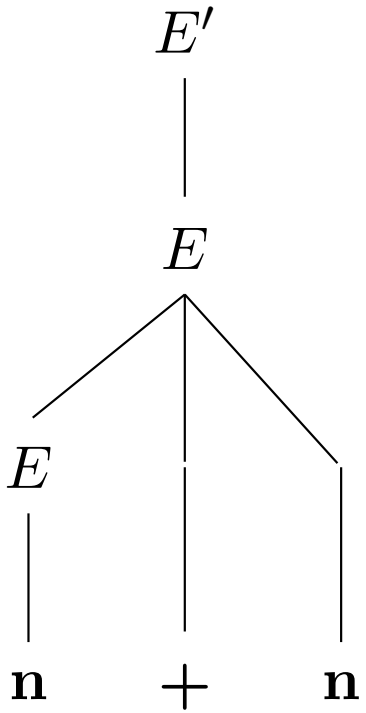


DFA addition again: LR(0)?

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



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



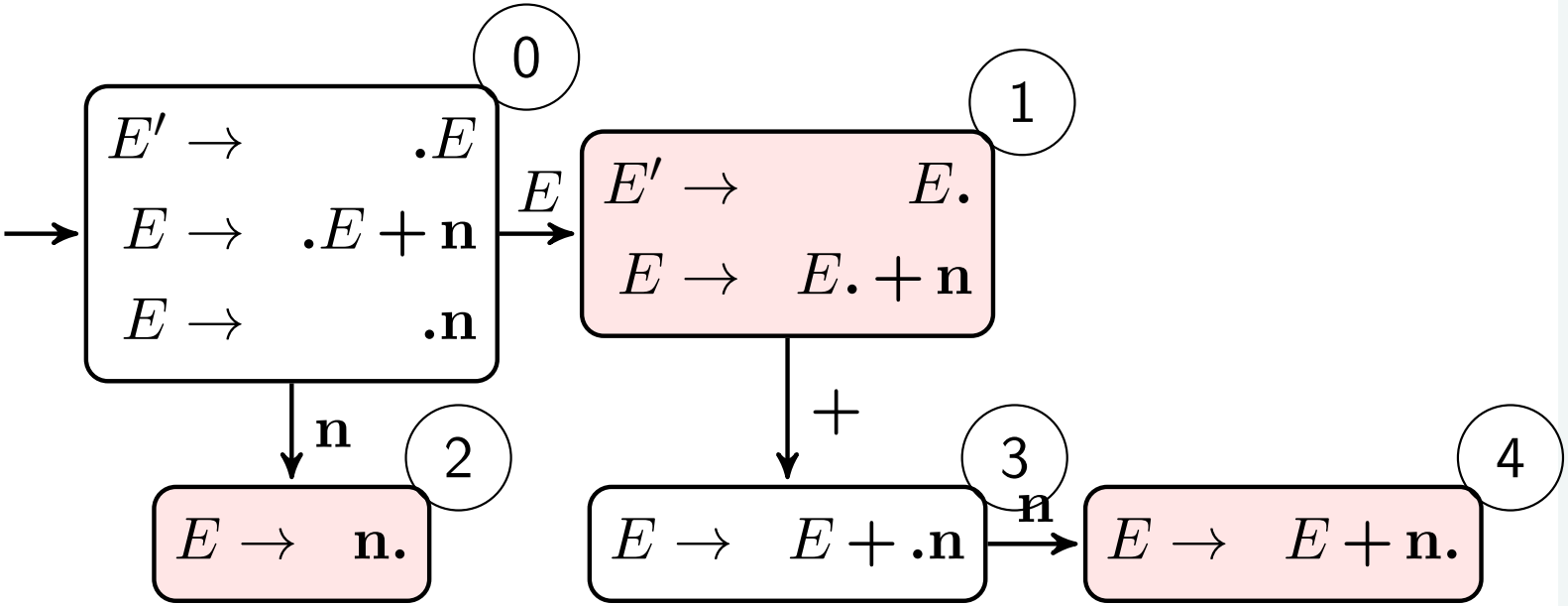
	parse stack	input	action
1	$\$$	$\mathbf{n} + \mathbf{n}\,\$$	shift
2	$\$\,\mathbf{n}$	$+\,\mathbf{n}\,\$$	red:. $E \to \mathbf{n}$
3	$\$\,E$	$+\,\mathbf{n}\,\$$	shift
4	$\$\,E +$	$\mathbf{n}\,\$$	shift
5	$\$\,E + \mathbf{n}$	$\$$	red. $E \to E + \mathbf{n}$
6	$\$\,E$	$\$$	red.: $E' \to E$
7	$\$\,E'$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

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

$\Rightarrow$ look-ahead on the input (without building the tree yet)

- LR(1) and its variants: *look-ahead of 1*

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)

One look-ahead

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Resolving LR(0) reduce/reduce conflicts

LR(0) reduce/reduce conflict:

$$\begin{array}{r} \cdots \\ A \rightarrow \alpha\bullet \\ \cdots \\ B \rightarrow \beta\bullet \end{array}$$



Resolving LR(0) reduce/reduce conflicts

LR(0) reduce/reduce conflict:

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

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

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

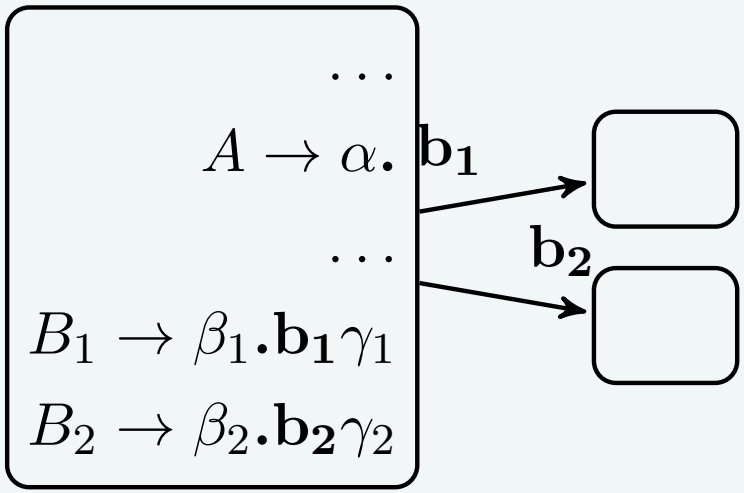
$\Rightarrow$ next symbol (in `token`) decides!

  - if `token` $\in Follow(A)$ then reduce using $A \rightarrow \alpha$
  - if `token` $\in Follow(B)$ then reduce using $B \rightarrow \beta$
  - ...



Resolving LR(0) shift/reduce conflicts

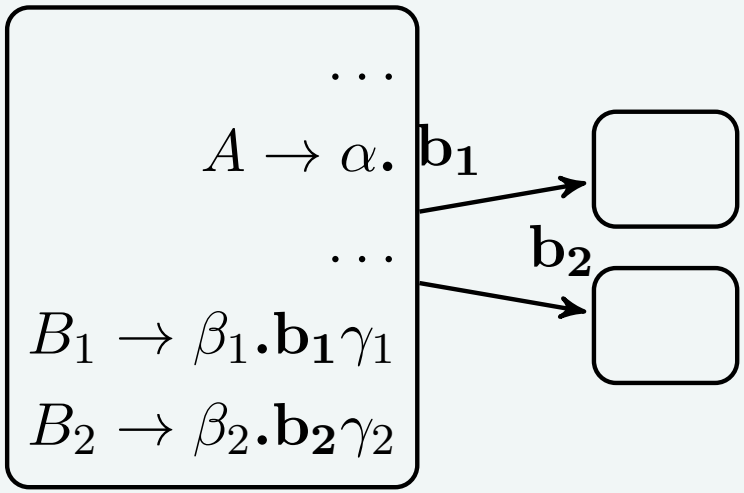
LR(0) shift/reduce conflict:





Resolving LR(0) shift/reduce conflicts

LR(0) shift/reduce conflict:



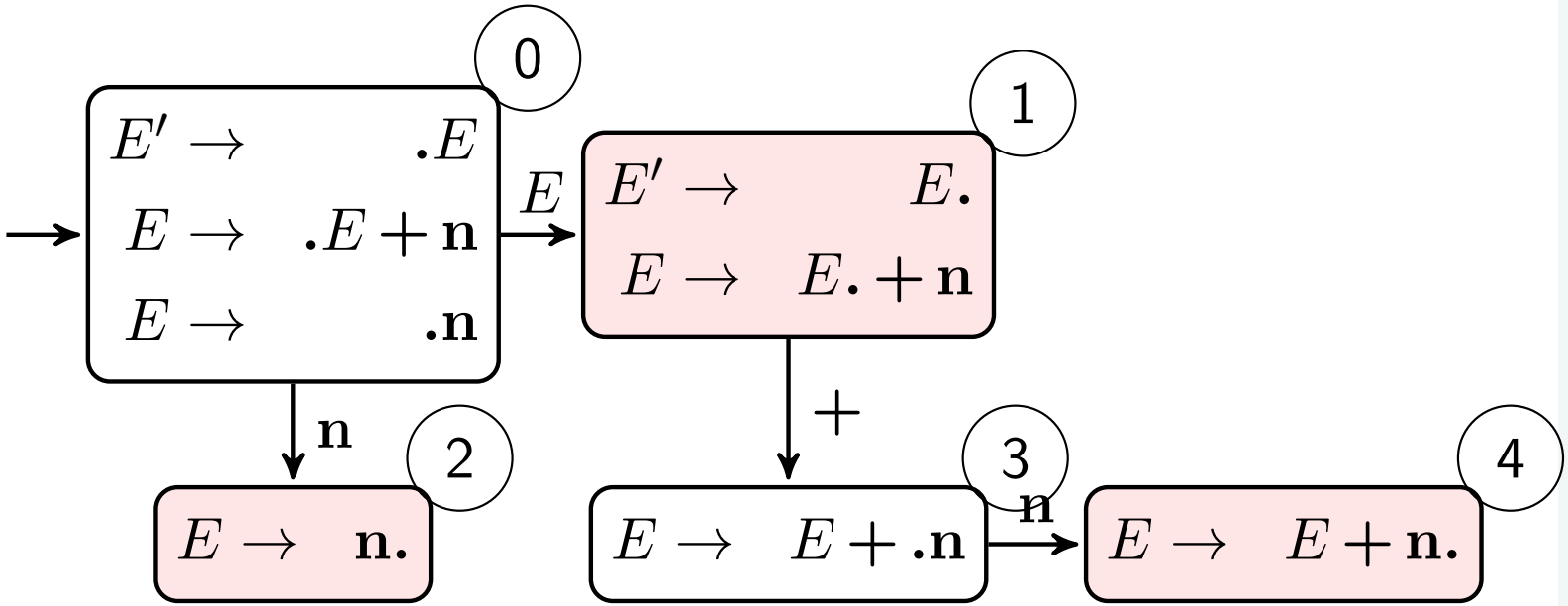
SLR(1) solution: again: use follow sets of non-terms

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

$\Rightarrow$ next symbol (in `token`) decides!

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

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

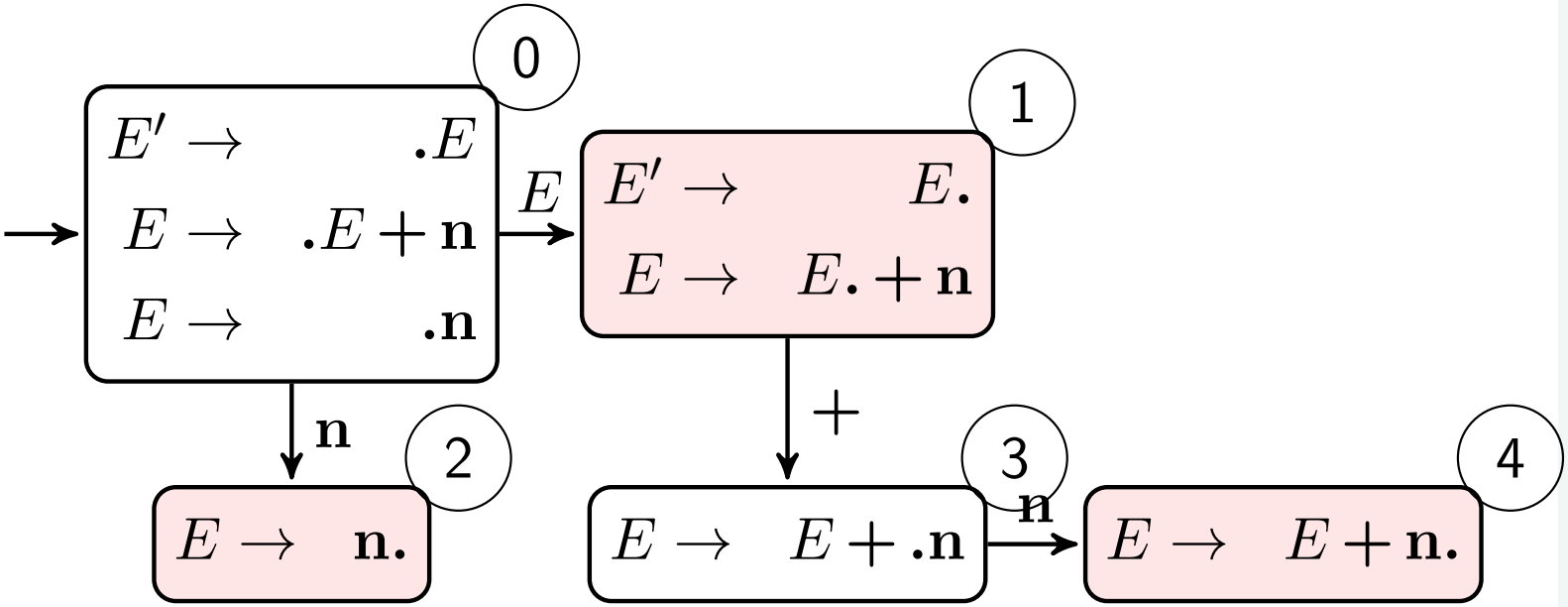
     pop: remove γ (including "its" states from the stack)

     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



Parsing table for SLR(1)



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

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



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 SLR(1) *shift-reduce* conflicts
  - would result in non-unique entries in SLR(1)-table[6]

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

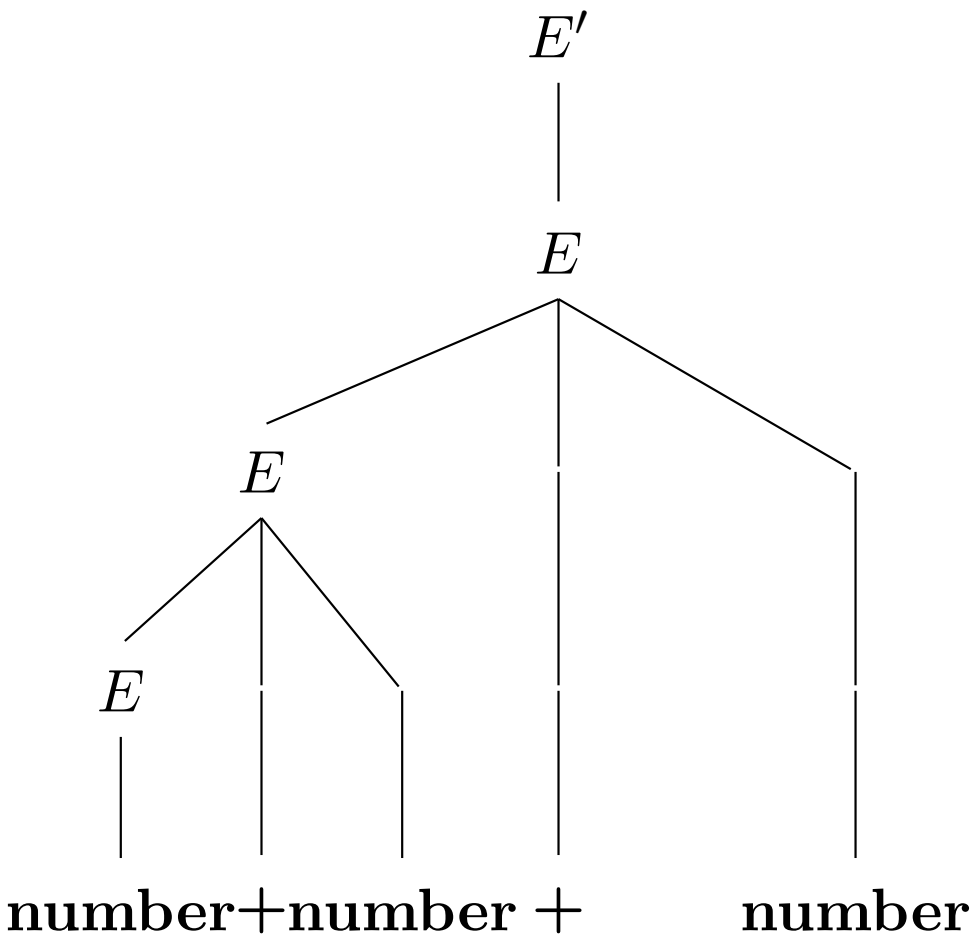
SLR(1) parser run (= "reduction")

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

stage	parsing stack	input	action
1	$\mathbf{\$}_0$	$\mathbf{n}+\mathbf{n}+\mathbf{n}\,\mathbf{\$}$	shift: 2
2	$\mathbf{\$}_0\mathbf{n}_2$	$+\mathbf{n}+\mathbf{n}\,\mathbf{\$}$	reduce: $E \to \mathbf{n}$
3	$\mathbf{\$}_0 E_1$	$+\mathbf{n}+\mathbf{n}\,\mathbf{\$}$	shift: 3
4	$\mathbf{\$}_0 E_1 +_3$	$\mathbf{n}+\mathbf{n}\,\mathbf{\$}$	shift: 4
5	$\mathbf{\$}_0 E_1 +_3 \mathbf{n}_4$	$+\mathbf{n}\,\mathbf{\$}$	reduce: $E \to E+\mathbf{n}$
6	$\mathbf{\$}_0 E_1$	$\mathbf{n}\,\mathbf{\$}$	shift 3
7	$\mathbf{\$}_0 E_1 +_3$	$\mathbf{n}\,\mathbf{\$}$	shift 4
8	$\mathbf{\$}_0 E_1 +_3 \mathbf{n}_4$	$\mathbf{\$}$	reduce: $E \to E+\mathbf{n}$
9	$\mathbf{\$}_0 E_1$	$\mathbf{\$}$	accept



Corresponding parse tree





Revisit the parentheses again: SLR(1)?

Grammar: parentheses

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

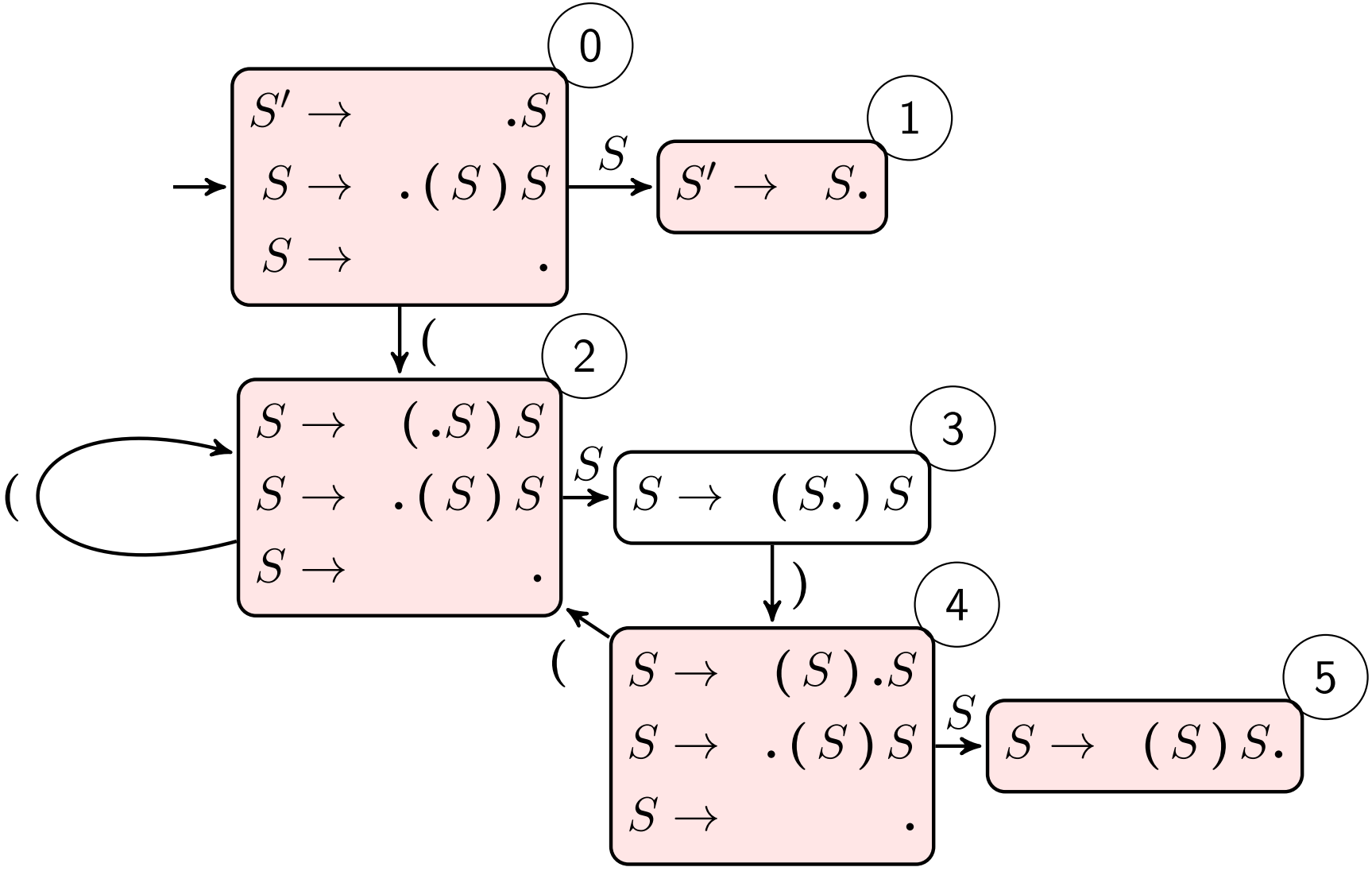
Follow set

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

DFA for parentheses



SLR(1) parse table

state	input			goto
	(	)	$	S
0	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	1
1			accept	
2	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	3
3		$s:4$		
4	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	5
5		$r: S \to (\ S\ )\ S$	$r: S \to (\ S\ )\ S$	



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

state	input			goto
	(	)	$	S
0	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	1
1			accept	
2	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	3
3		$s:4$		
4	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	5
5		$r: S \to (\ S\ )\ S$	$r: S \to (\ S\ )\ S$	

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



Ambiguity & LR-parsing

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

Additional means of disambiguation:

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

- for the second point ("let the parser decide according to its preferences"):
  - use sparingly and cautiously
  - typical example: *dangling-else*
  - even if parsers makes a decision, programmar may or may not "understand intuitively" the resulting parse tree (and thus AST)



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example of an ambiguous grammar

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

In the following, E for exp, etc.



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Simplified conditionals

Simplified "schematic" if-then-else

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

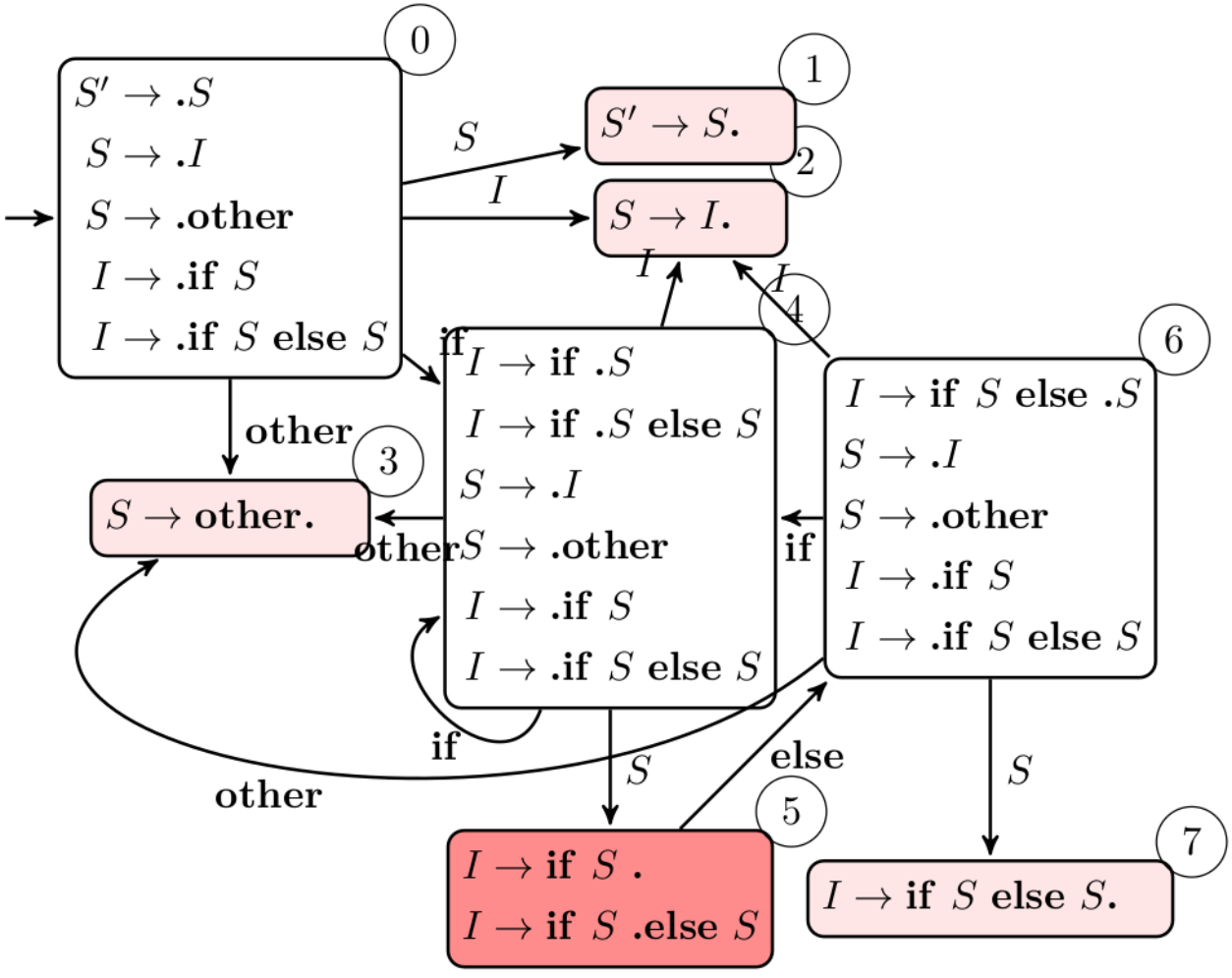
Follow-sets

	$Follow$
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

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

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

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

- *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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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	$\mathbf{\$_0}$	**if if other else other $**	shift: 4
2	$\mathbf{\$_0 if_4}$	**if other else other $**	shift: 4
3	$\mathbf{\$_0 if_4 if_4}$	**other else other $**	shift: 3
4	$\mathbf{\$_0 if_4 if_4 other_3}$	**else other $**	reduce: 2
5	$\mathbf{\$_0 if_4 if_4} S_5$	**else other $**	shift 6
6	$\mathbf{\$_0 if_4 if_4} S_5 \mathbf{else_6}$	**other $**	shift: 3
7	$\mathbf{\$_0 if_4 if_4} S_5 \mathbf{else_6 other_3}$	**$**	reduce: 2
8	$\mathbf{\$_0 if_4 if_4} S_5 \mathbf{else_6} S_7$	**$**	reduce: 4
9	$\mathbf{\$_0 if_4} I_2$	**$**	reduce: 1
10	$\mathbf{\$_0} S_1$	**$**	accept



Parser run, different choice

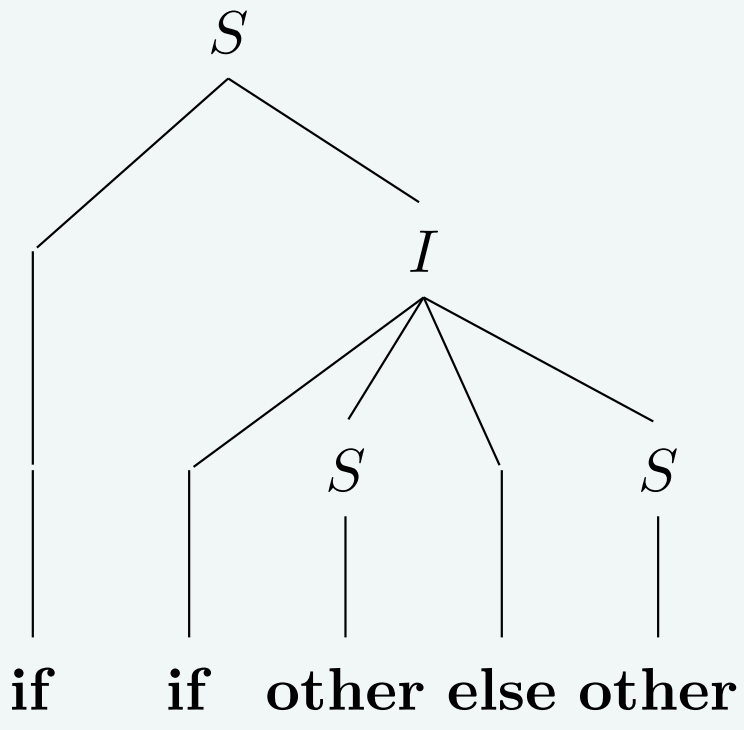
state	input				goto	
	if	**else**	**other**	**$**	S	I
0	$s:4$		$s:3$		1	2
1				accept		
2		$r:1$		$r:1$		
3		$r:2$		$r:2$		
4	$s:4$		$s:3$		5	2
5		$s:6$		$r:3$		
6	$s:4$		$s:3$		7	2
7		$r:4$		$r:4$		

stage	parsing stack	input	action
1	$\mathbf{\$_0}$	**if if other else other $**	shift: 4
2	$\mathbf{\$_0 if_4}$	**if other else other $**	shift: 4
3	$\mathbf{\$_0 if_4 if_4}$	**other else other $**	shift: 3
4	$\mathbf{\$_0 if_4 if_4 other_3}$	**else other $**	reduce: 2
5	$\mathbf{\$_0 if_4 if_4} S_5$	**else other $**	reduce 3
6	$\mathbf{\$_0 if_4} I_2$	**else other $**	reduce 1
7	$\mathbf{\$_0 if_4} S_5$	**else other $**	shift 6
8	$\mathbf{\$_0 if_4} S_5 \mathbf{else_6}$	**other $**	shift 3
9	$\mathbf{\$_0 if_4} S_5 \mathbf{else_6 other_3}$	**$**	reduce 2
10	$\mathbf{\$_0 if_4} S_5 \mathbf{else_6} S_7$	**$**	reduce 4
11	$\mathbf{\$_0} S_1$	**$**	accept

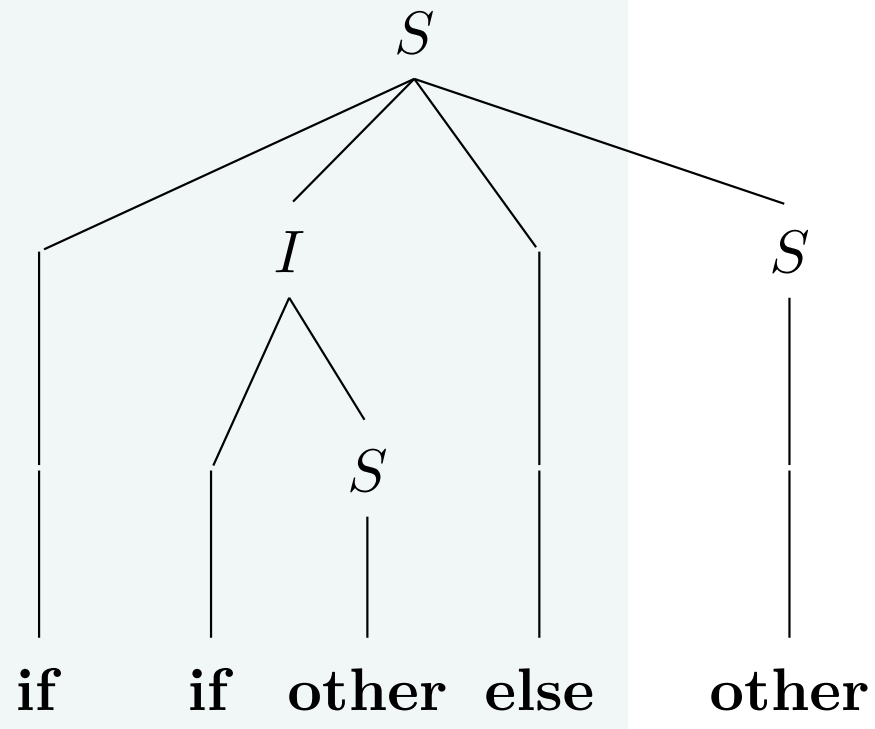


Parse trees for the "simple conditions"

shift-precedence: conventional



"wrong" tree



standard "dangling else" convention

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

Use of ambiguous grammars

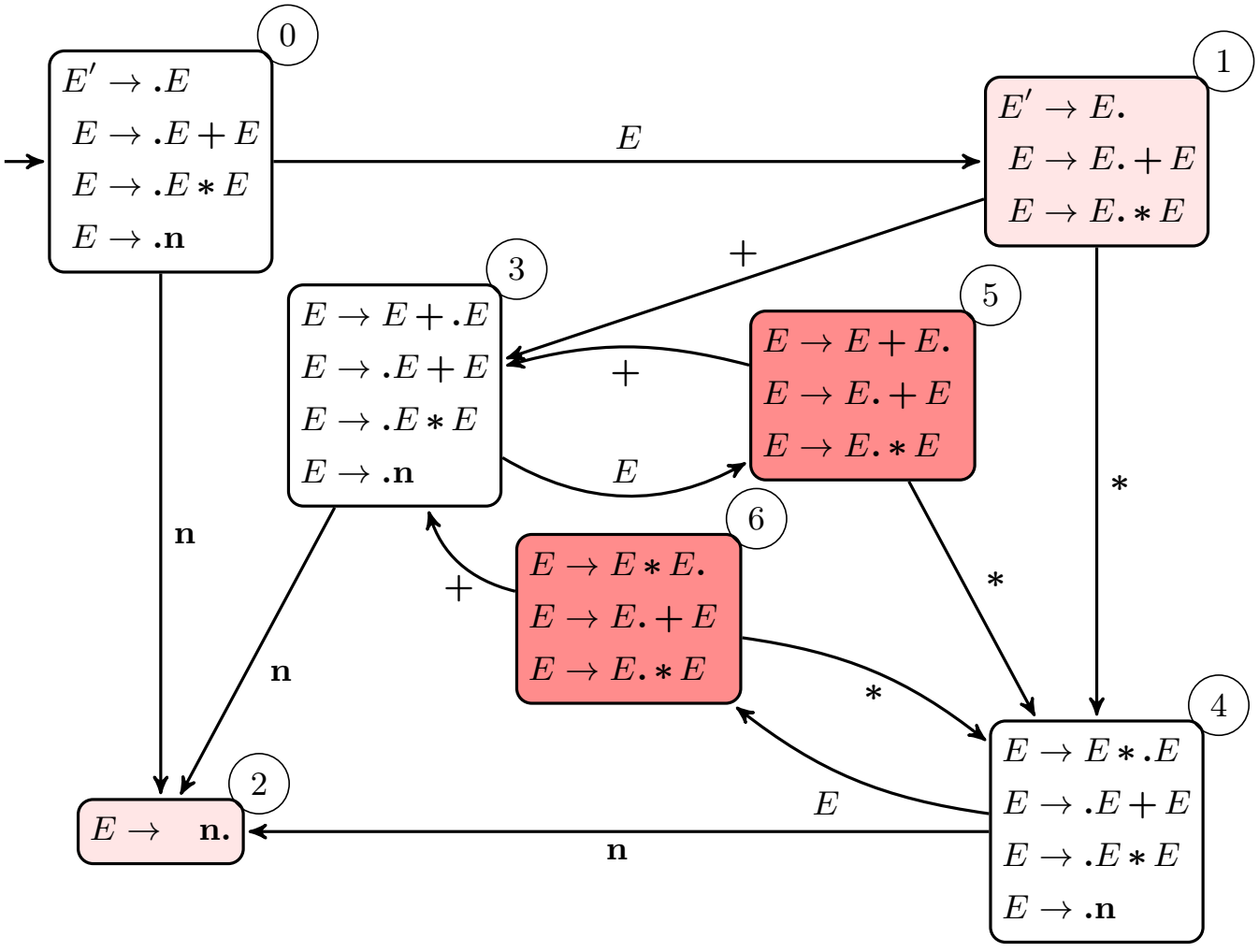
- advantage of ambiguous grammars: often simpler
- if ambiguous: grammar guaranteed to have conflicts
- can be (often) resolved by specifying *precedence* and *associativity*
- supported by tools like `yacc` and `CUP` ...

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + E \mid E * E \mid \mathbf{number}
\end{array}
$$



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

DFA for $+$ and $\times$



States with conflicts

- state 5
  - stack contains ...$E + E$
  - for input \$: reduce, since shift not allowed form \$
  - for input $+$; reduce, as $+$ is *left-associative*
  - for input $*$: shift, as $*$ has *precedence* over $+$
- state 6:
  - stack contains ...$E * E$
  - for input \$: reduce, since shift not allowed form \$
  - for input $+$; reduce, a $*$ has *precedence* over $+$
  - for input $*$: reduce, as $*$ is *left-associative*
- see also the table on the next slide



Parse table + and ×

state	input				goto
	$\mathbf{n}$	$+$	$*$	$\$$	E
0	$s:2$				1
1		$s:3$	$s:4$	accept	
2		$r:E\to\mathbf{n}$	$r:E\to\mathbf{n}$	$r:E\to\mathbf{n}$	
3	$s:2$				5
4	$s:2$				6
5		$r:E\to E+E$	$s:4$	$r:E\to E+E$	
6		$r:E\to E*E$	$r:E\to E*E$	$r:E\to E*E$	

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

Defined as *right-associative*. See exercise



Compare: unambiguous grammar for + and *



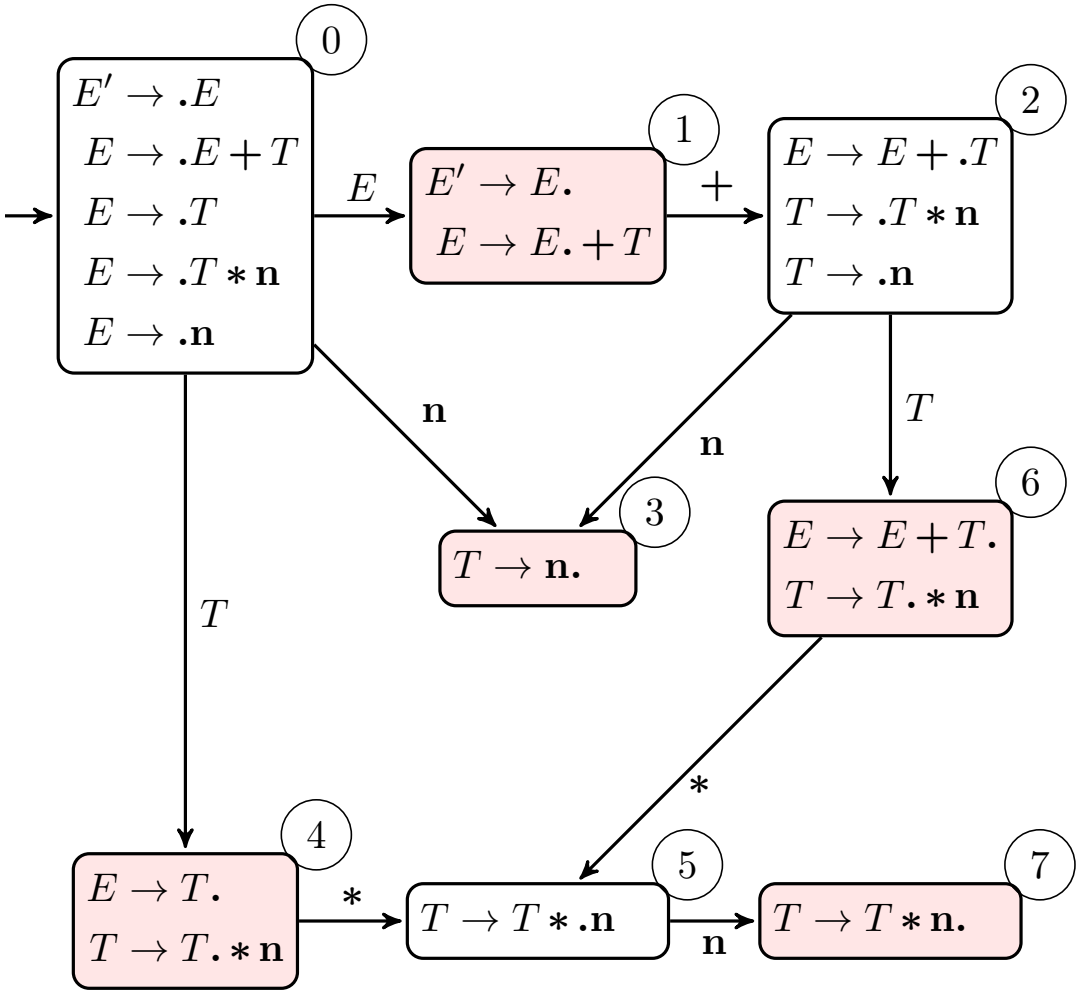
Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Unambiguous grammar: precedence and left-assoc built in

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

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

DFA for unambiguous + and ×





DFA remarks

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

    state 1: $Follow(E') = \{\$\}$
    state 4: $Follow(E) = \{\$, +\}$
    state 6: $Follow(E) = \{\$, +\}$
    state 3/7: $Follow(T) = \{\$, +, *\}$
  - in no case there's a shift/reduce conflict (check the outgoing edges vs. the follow set)
  - there's not reduce/reduce conflict either



LR(1) parsing

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

Basic restriction of SLR(1)

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

A help to remember

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Limitations of SLR(1) grammars

Assignment grammar fragment

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

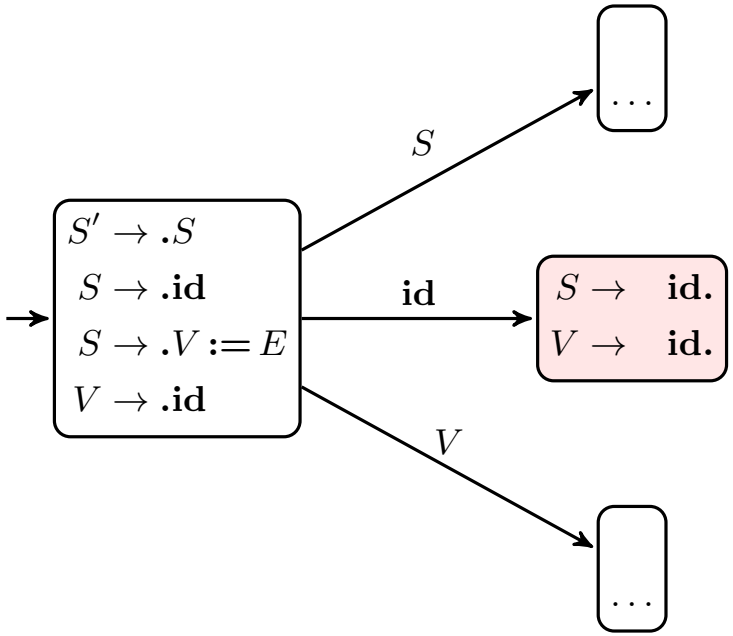
Assignment grammar fragment, simplified

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

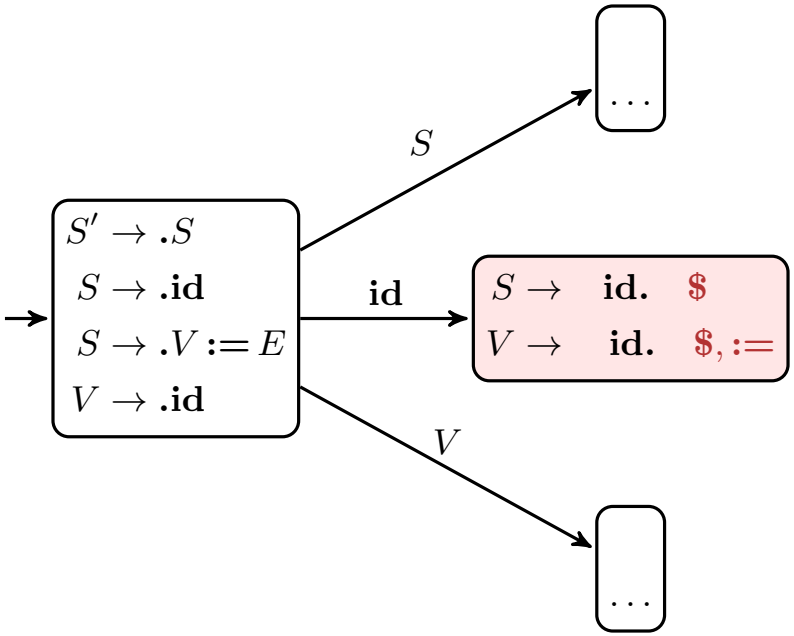
Non-SLR(1): Reduce/reduce conflict



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

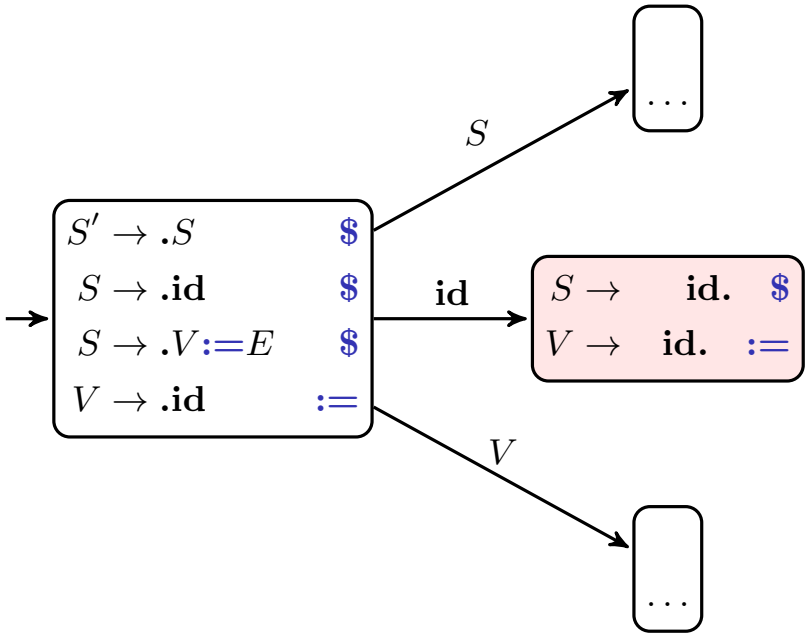


Non-SLR(1): Reduce/reduce conflict



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

Situation can be saved: more look-ahead





Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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

Add precision in the states of the automaton already

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

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LR(1) items

- main idea: simply make the look-ahead part of the item
- obviously: proliferation of states[8]

LR(1) items

$$[A \rightarrow \alpha.\beta, \mathbf{a}] \tag{10}$$

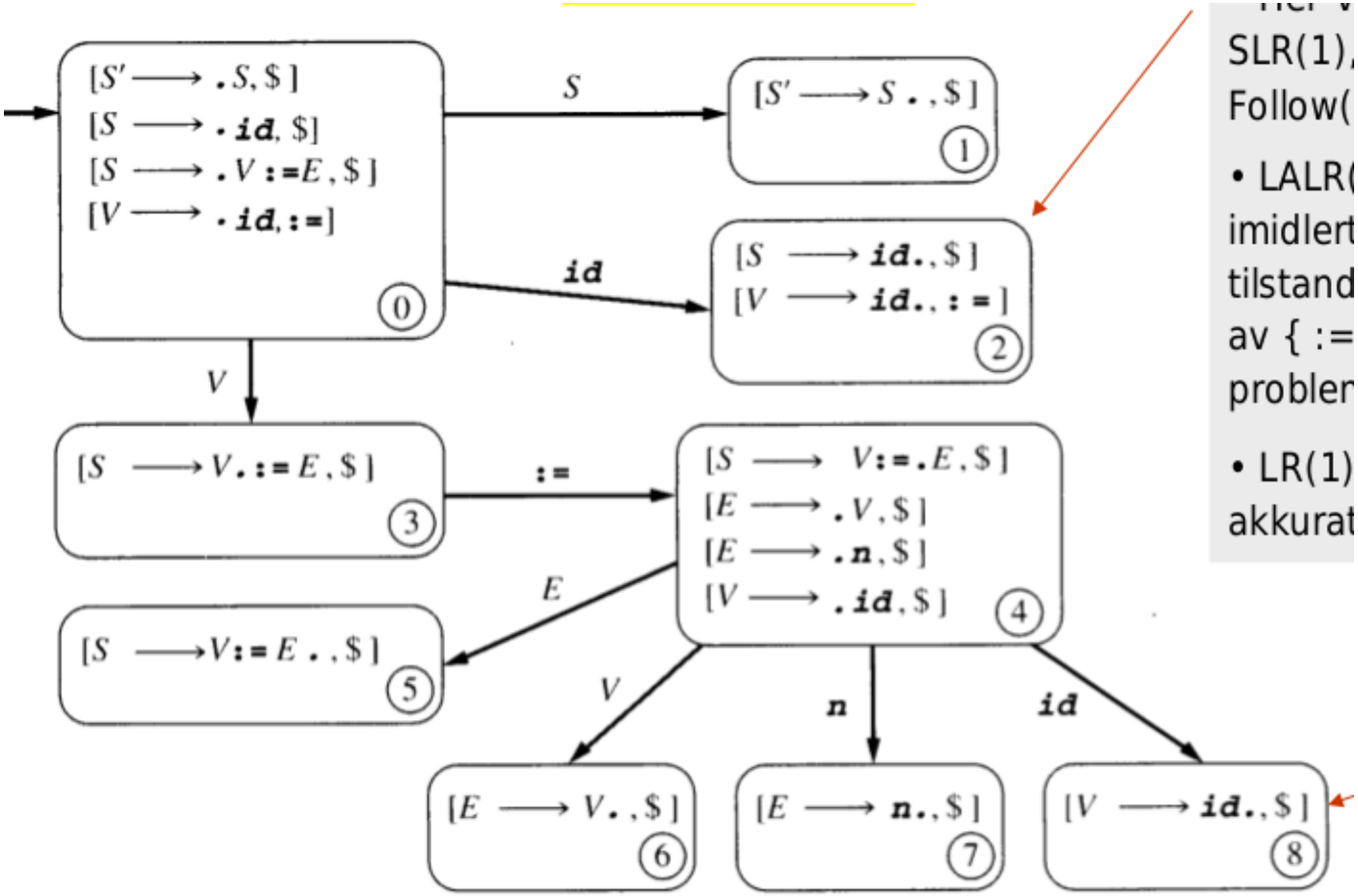
- $\mathbf{a}$: terminal/token, including $\$$

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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



Remarks on the DFA

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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)	LALR(1)
LR(0)-item-based parsing, with *afterwards* adding some extra "pre-compiled" info (about follow-sets) to increase expressivity	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

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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LR(1) transitions: ϵ

ϵ-transition

for all

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

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

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

for all $B \to \beta \mid \ldots$

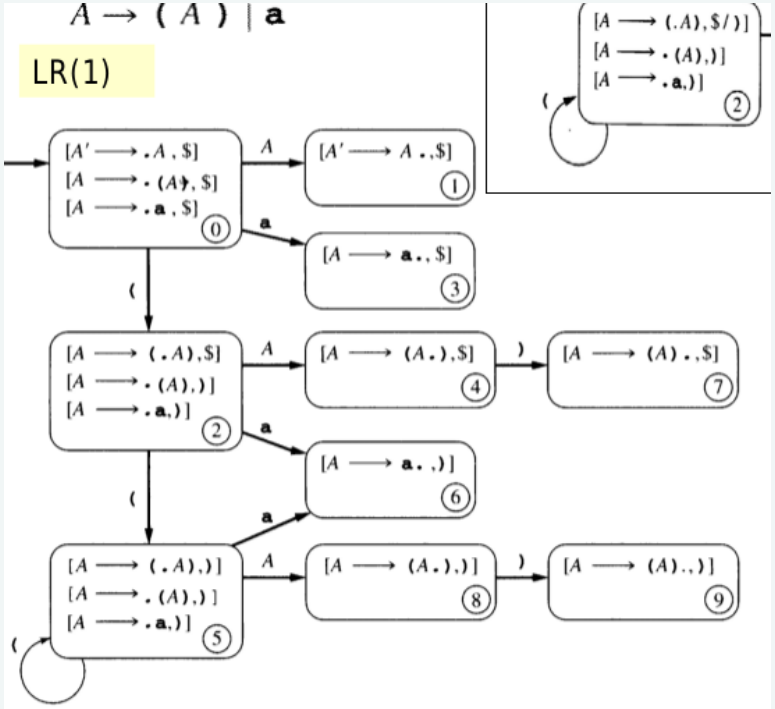
$$[A \to \alpha . B \quad , \mathbf{a}] \xrightarrow{\epsilon} [B \to . \beta \quad , \mathbf{a}]$$

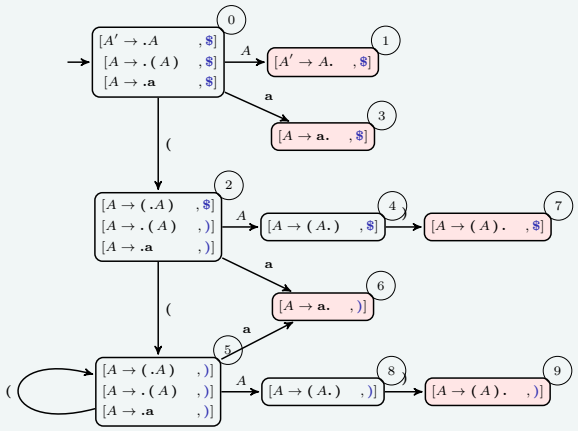


Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

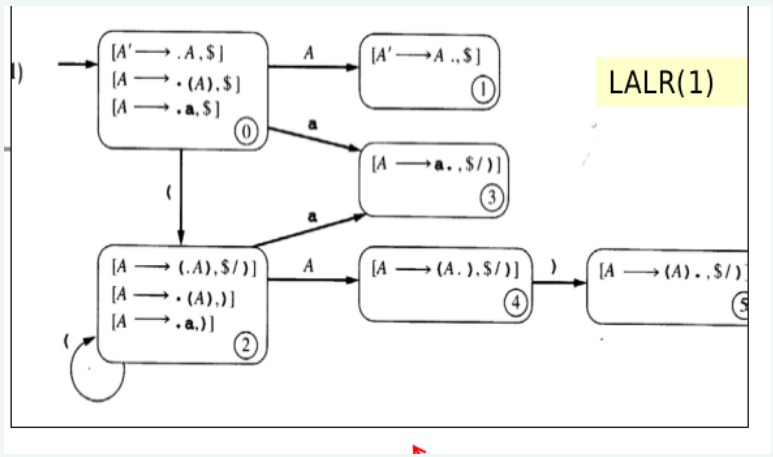
LALR(1) vs.\ LR(1)

LR(1)





LALR(1) (check if new)



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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

LALR(1)-DFA by collapse

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



Concluding remarks of LR / bottom up parsing

- all constructions (here) based on BNF (not EBNF)
- *conflicts* (for instance due to ambiguity) can be solved by
  - reformulate the grammar, but generarate the same language[9]
  - 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)

[9] 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

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

Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Error handling

Minimal requirement

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

- 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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Error messages

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



Error recovery in bottom-up parsing

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

$\Rightarrow$ 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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Possible error situation

	parse stack	input	action
1	$\$_0 \mathbf{a}_1 \mathbf{b}_2 \mathbf{c}_3 (_4 \mathbf{d}_5 \mathbf{e}_6$	$\mathbf{f}\ )\ \mathbf{g}\ \mathbf{h} \ldots \$$	no entry for $\mathbf{f}$

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

Possible error situation

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Panic mode recovery

Algo

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example again

	parse stack	input	action
1	$\$_0 \mathbf{a}_1 \mathbf{b}_2 \mathbf{c}_3 (_4 \mathbf{d}_5 \mathbf{e}_6$	$\mathbf{f}\ )\ \mathbf{g}\mathbf{h} \ldots \$$	no entry for $\mathbf{f}$

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

Example again

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

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



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

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!



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 $\mathbf{n}$ 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).



Targets & Outline
Introduction to parsing
Top-down parsing
First and follow sets
Massaging grammars
LL-parsing (mostly LL(1))
Bottom-up parsing

References I



Bibliography

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

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