# Chapter 4

## Parsing (will be polished/updated )

Course “Compiler Construction”
Martin Steffen
Spring 2024

‘

# Chapter 4

## Learning Targets of Chapter “Parsing (will be polished/updated )”.

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 (will be polished/up-dated )”.

**Introduction to parsing**

**Top-down parsing**

**First and follow sets**

**First and follow sets**

**Massaging grammars**

**LL-parsing (mostly LL(1))**

**Error handling**

**Bottom-up parsing**

# Section

## Introduction to parsing

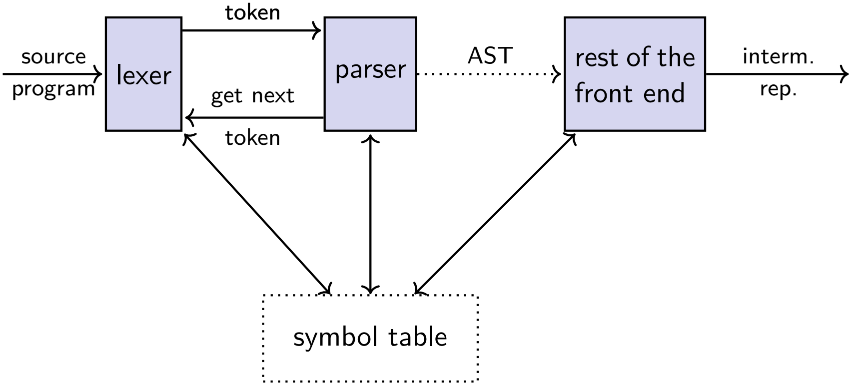
Chapter 4 "Parsing (will be polished/updated )"
Course "Compiler Construction"
Martin Steffen
Spring 2024

# 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 . . . )


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. |

# Parsing restricted classes of CFGs

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

# Classes of CFG grammars/languages

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

**top-down parsing, in particular**

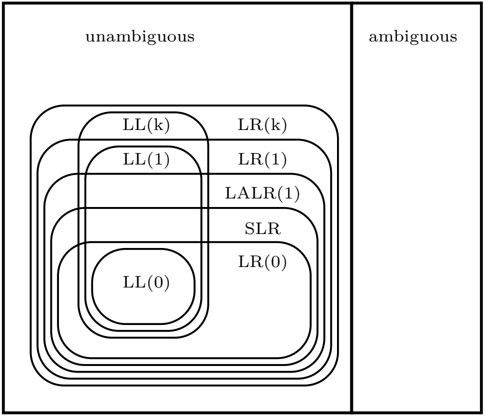
- LL(1)
- recursive descent

**bottom-up parsing**

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

- grammars typically written in *pure* BNF

# Relationship of some grammar (not language) classes




taken from [1]

# Section

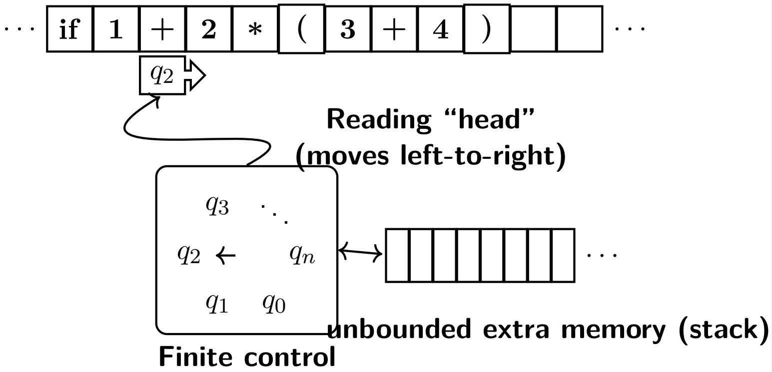
## Top-down parsing

Chapter 4 “Parsing (will be polished/updated )”
Course “Compiler Construction”
Martin Steffen
Spring 2024

# General task (once more)

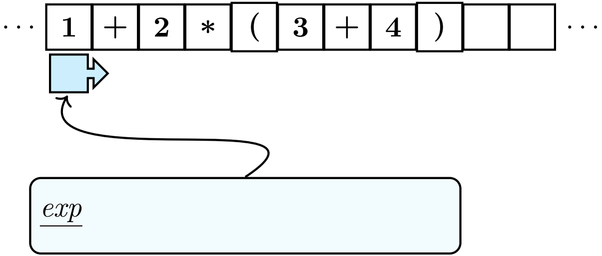
- Given: a CFG (but appropriately restricted)
- Goal: "systematic method" s.t.
  1. for every given word $w$: check syntactic correctness
  2. [build AST/representation of the parse tree as side effect]
  3. [do reasonable error handling]

# Schematic view on “parser machine”




Note: sequence of *tokens* (not characters)

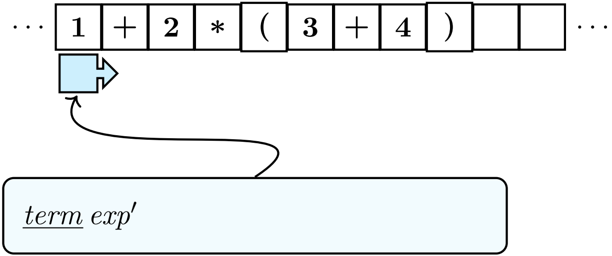
# Derivation of an expression




## factors and terms

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

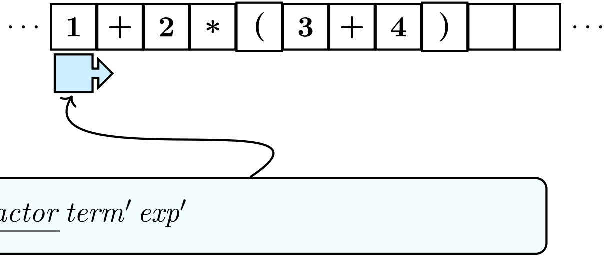
# Derivation of an expression




## factors and terms

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

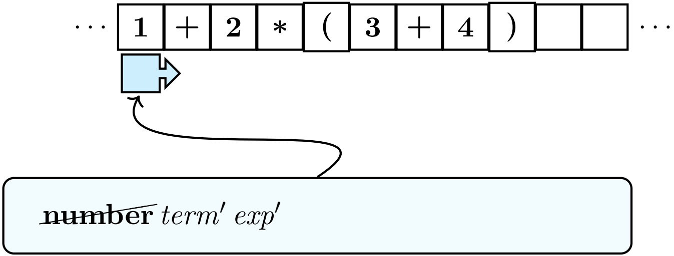
# Derivation of an expression




### factors and terms

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

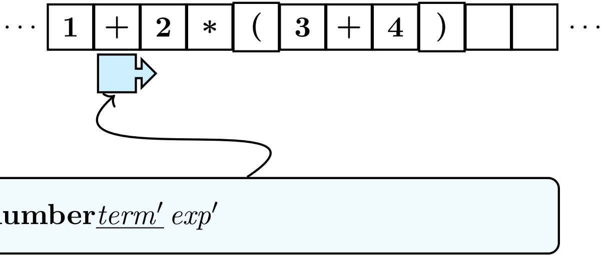
# Derivation of an expression




## factors and terms

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

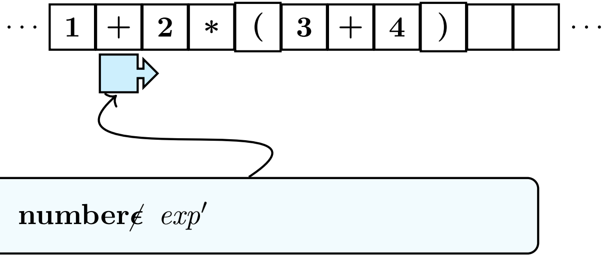
# Derivation of an expression




## factors and terms

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

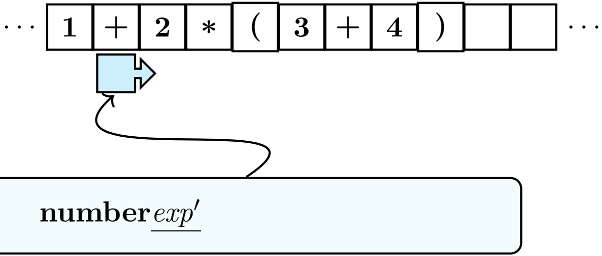
# Derivation of an expression




## factors and terms

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

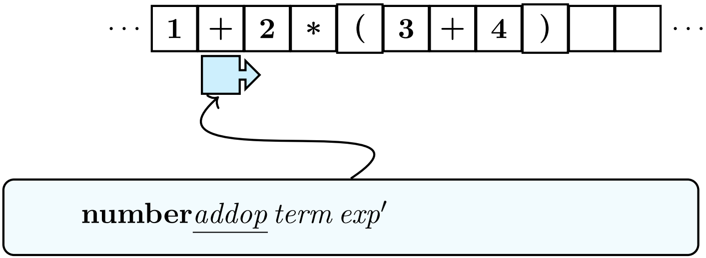
# Derivation of an expression




## factors and terms

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

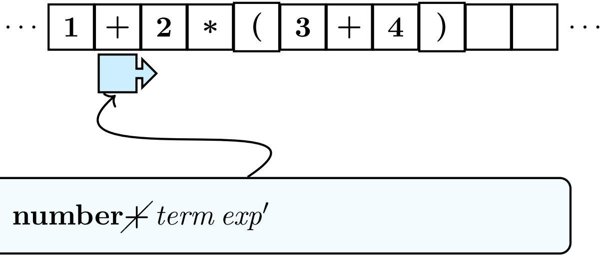
# Derivation of an expression




## factors and terms

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

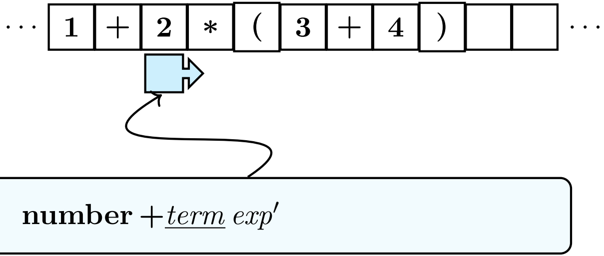
# Derivation of an expression




**factors and terms**

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

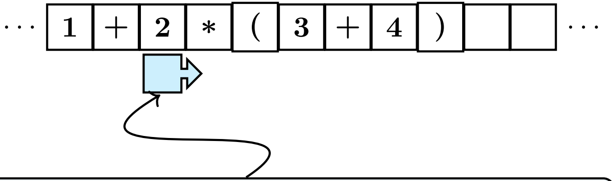
# Derivation of an expression




## factors and terms

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

# Derivation of an expression

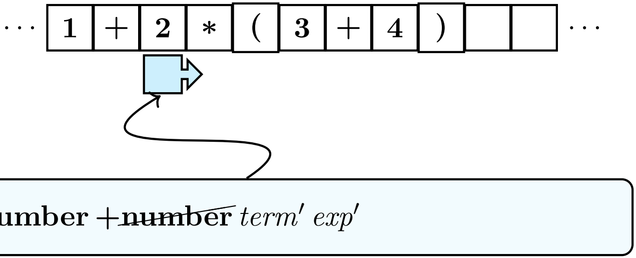



$\textbf{number} + \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'$

## factors and terms

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

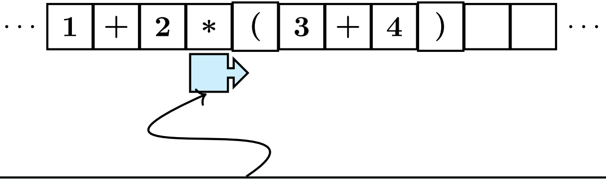
# Derivation of an expression




## factors and terms

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

# Derivation of an expression

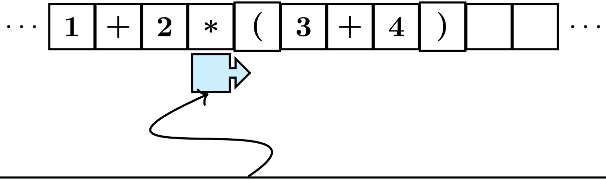



$$\textbf{number} + \textbf{number}\, \underline{term'}\; exp'$$

## factors and terms

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

# Derivation of an expression

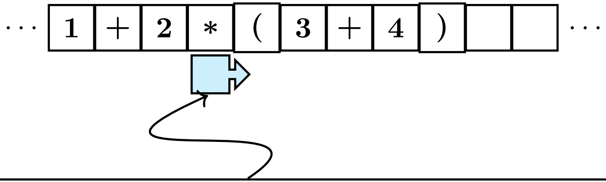



$$\textbf{number} + \textbf{number}\,\underline{\mathit{mulop}}\,\mathit{factor}\ \mathit{term}'\ \mathit{exp}'$$

## factors and terms

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

# Derivation of an expression

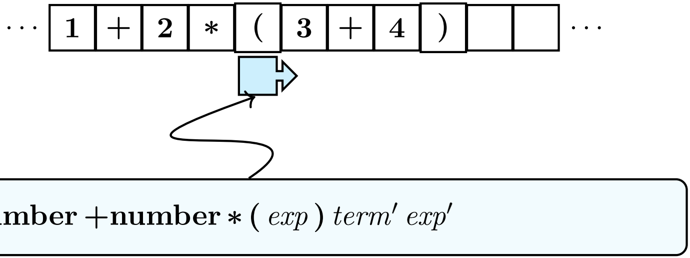



$$\mathbf{number}\ \mathbf{+}\mathbf{number}\ \not{*}\ \underline{\textit{factor}}\ \textit{term}'\ \textit{exp}'$$

## factors and terms

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

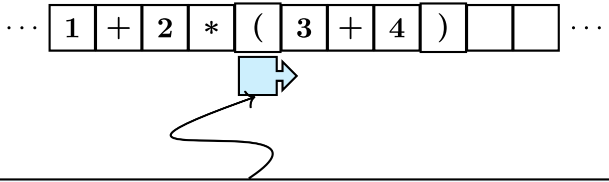
# Derivation of an expression




## factors and terms

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

# Derivation of an expression

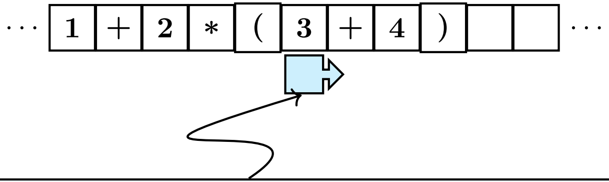



$$\textbf{number} + \textbf{number} * (\; exp \;)\; term'\; exp'$$

## factors and terms

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

# Derivation of an expression

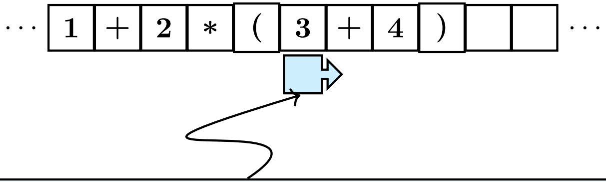



$$\textbf{number} + \textbf{number} * (\, \underline{exp}\, )\ term'\ exp'$$

## factors and terms

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

# Derivation of an expression

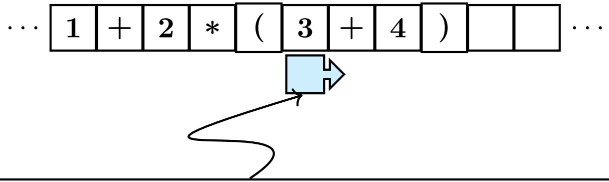



$$\mathbf{number} + \mathbf{number} * ( \underline{\textit{term}}\ \textit{exp}' )\ \textit{term}'\ \textit{exp}'$$

**factors and terms**

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

# Derivation of an expression

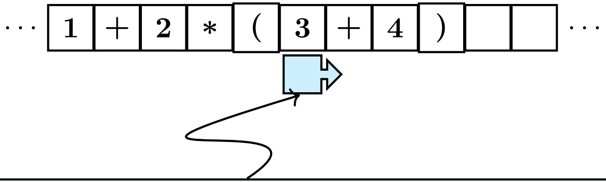



$$\mathbf{number} + \mathbf{number} * (\, \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'\,)\ \mathit{term}'\ \mathit{exp}'$$

## factors and terms

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

# Derivation of an expression

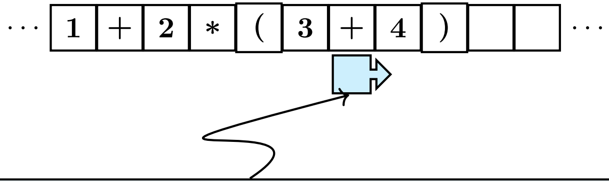



$$\mathbf{number} + \mathbf{number} * ( \sout{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}'$$

## factors and terms

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

# Derivation of an expression

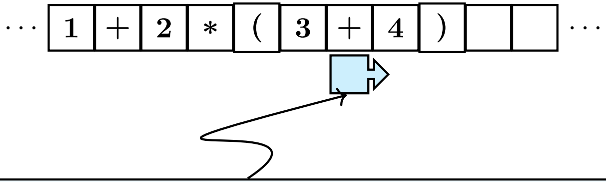



$$\mathbf{number} + \mathbf{number} * ( \mathbf{number}\, \underline{term'}\; exp' \,)\; term'\; exp'$$

## factors and terms

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

# Derivation of an expression

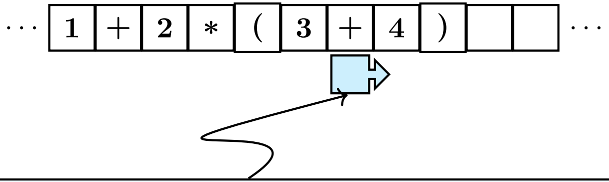



$$\textbf{number} + \textbf{number} * ( \textbf{number} \not\epsilon \; exp' \; ) \; term' \; exp'$$

## factors and terms

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

# Derivation of an expression

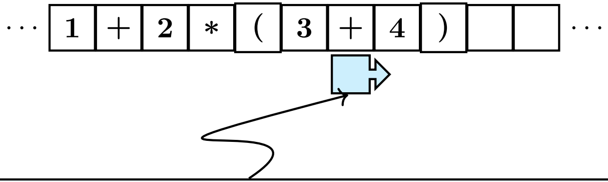



$$\textbf{number} + \textbf{number} * ( \textbf{number}\, \underline{\mathit{exp}'}\, )\ \mathit{term}'\ \mathit{exp}'$$

**factors and terms**

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \qquad (1)\\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \boldsymbol{\epsilon}\\
\mathit{addop} &\rightarrow \mathbf{+} \mid \mathbf{-}\\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}'\\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \boldsymbol{\epsilon}\\
\mathit{mulop} &\rightarrow \mathbf{*}\\
\mathit{factor} &\rightarrow (\ \mathit{exp}\ ) \mid \textbf{number}
\end{aligned}
$$

# Derivation of an expression

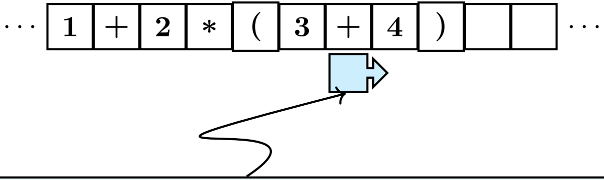



$$\textbf{number} + \textbf{number} * (\ \textbf{number}\ \underline{\mathit{addop}}\ \mathit{term}\ \mathit{exp}'\ )\ \mathit{term}$$

## factors and terms

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

# Derivation of an expression

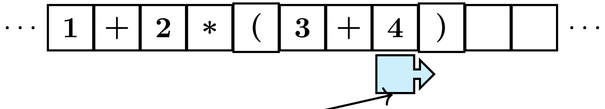



$$\textbf{number} + \textbf{number} * ( \textbf{number} \not+ \textit{term}\ \textit{exp}' ) \textit{term}'\ \textit{ex}$$

## factors and terms

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

# Derivation of an expression

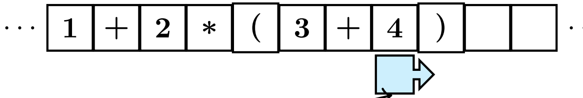



$$\textbf{number} + \textbf{number} * ( \textbf{number} + \underline{\mathit{term}}\ \mathit{exp}' )\ \mathit{term}'\ \mathit{e}$$

## factors and terms

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

# Derivation of an expression

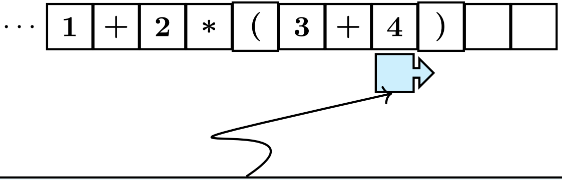



$$\textbf{number} + \textbf{number} * (\, \textbf{number} + \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'$$

## factors and terms

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \boldsymbol{\epsilon} \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \boldsymbol{\epsilon} \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\, \mathit{exp}\,) \mid \textbf{number}
\end{aligned}
\tag{1}
$$

# Derivation of an expression

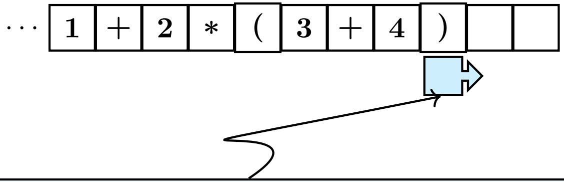



$\mathbf{number} + \mathbf{number} * ( \mathbf{number} + \sout{\mathbf{number}}\ term'\ e$

### factors and terms

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

# Derivation of an expression

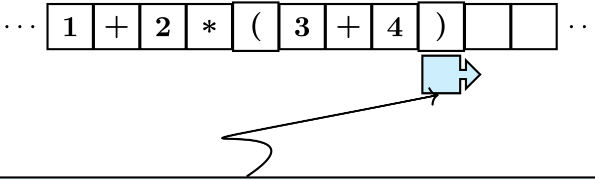



$\mathbf{number} + \mathbf{number} * ( \mathbf{number} + \mathbf{number} \underline{\textit{term}'}$

## factors and terms

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

# Derivation of an expression

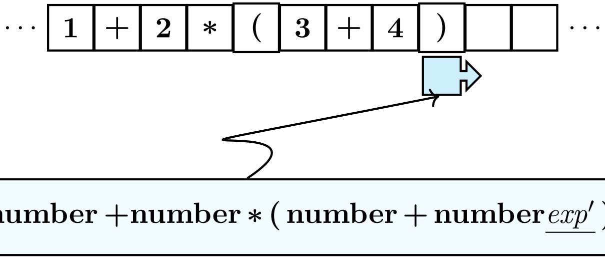



$\textbf{number} + \textbf{number} * ( \textbf{number} + \textbf{number} \not\epsilon \; exp'$

## factors and terms

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

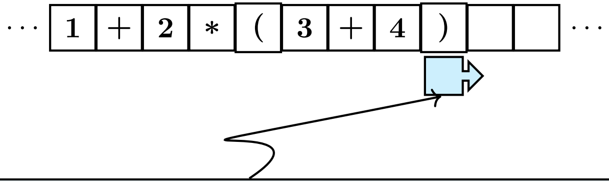
# Derivation of an expression




## factors and terms

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

# Derivation of an expression

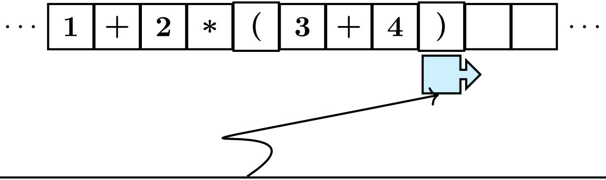



$\mathbf{number} + \mathbf{number} * ( \mathbf{number} + \mathbf{number} \not\epsilon\, ) \; t$

## 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

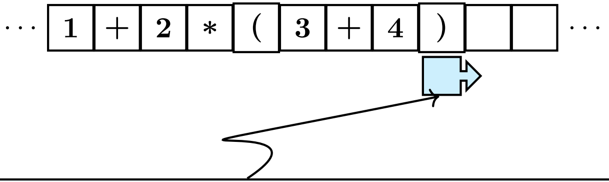



**number + number * ( number + number )** *te*

## factors and terms

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

# Derivation of an expression

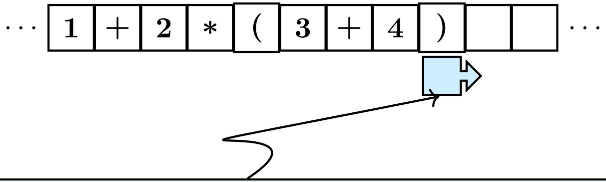



**number** +**number** * ( **number** + **number** ) *t*

## factors and terms

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

# Derivation of an expression

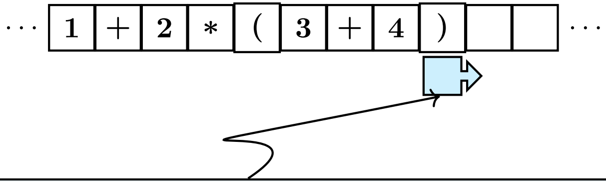



**number + number * ( number + number )**

## factors and terms

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

# Derivation of an expression

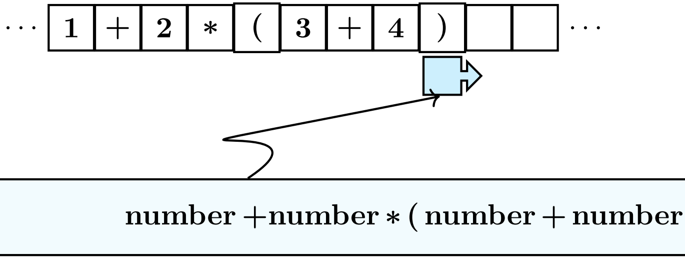



**number + number * ( number + number )**

## factors and terms

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

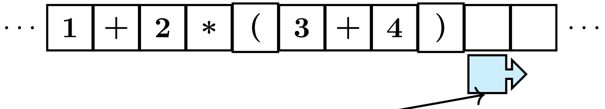
# Derivation of an expression




**factors and terms**

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

# Derivation of an expression




**number** + **number** ∗ ( **number** + **number**

## factors and terms

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

# Remarks concerning the derivation

Note:

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

Notation:

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

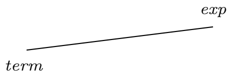
# Not as a "film" but at a glance: reduction sequence

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

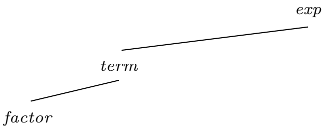
# Best viewed as a tree

$exp$

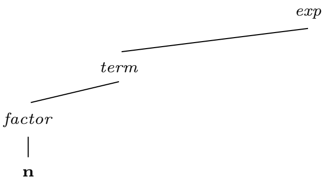
# Best viewed as a tree



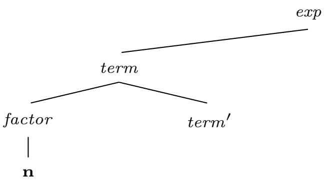
# Best viewed as a tree



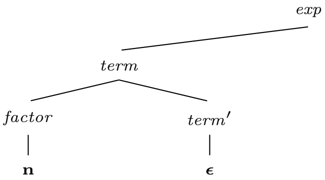
# Best viewed as a tree



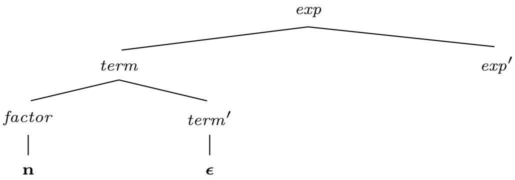
# Best viewed as a tree



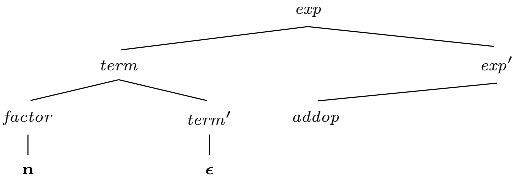
# Best viewed as a tree



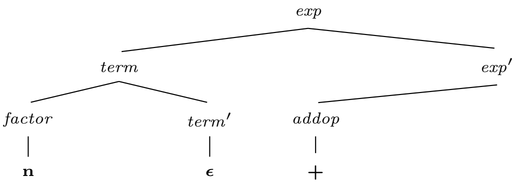
# Best viewed as a tree



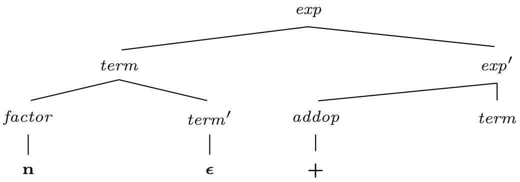
# Best viewed as a tree



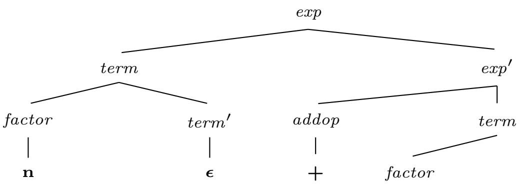
# Best viewed as a tree



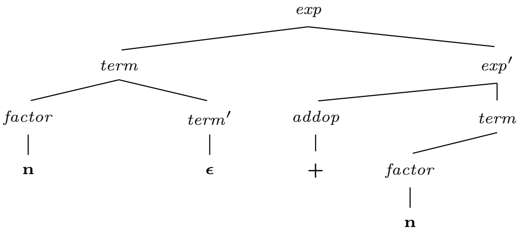
# Best viewed as a tree



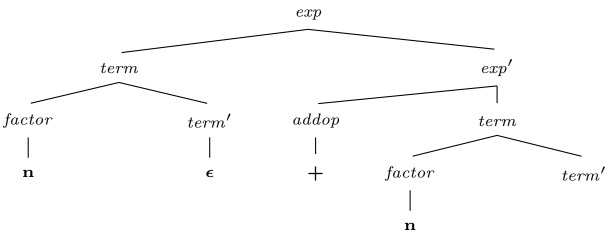
# Best viewed as a tree



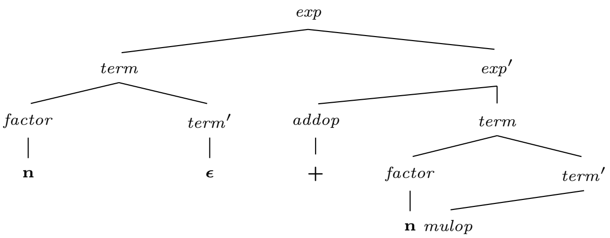
# Best viewed as a tree



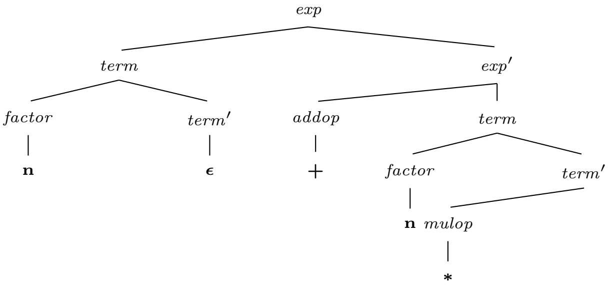
# Best viewed as a tree



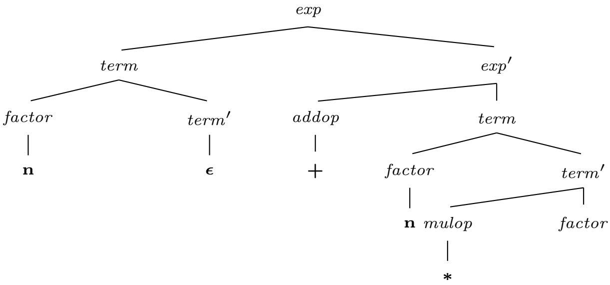
# Best viewed as a tree



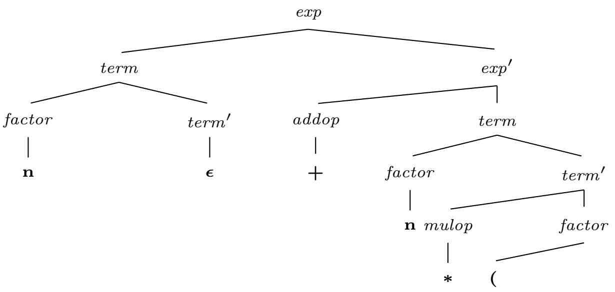
# Best viewed as a tree



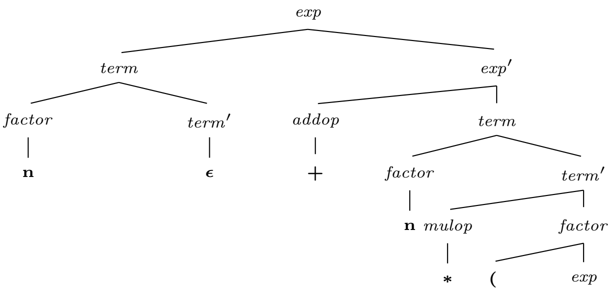
# Best viewed as a tree



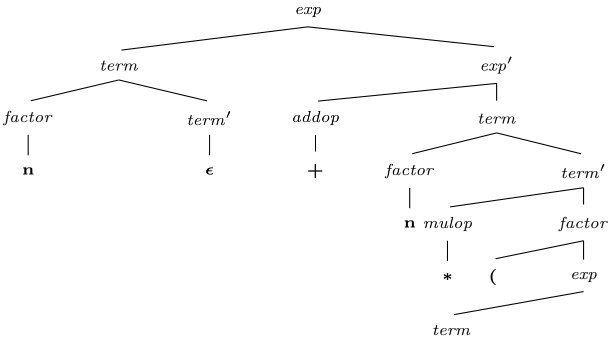
# Best viewed as a tree



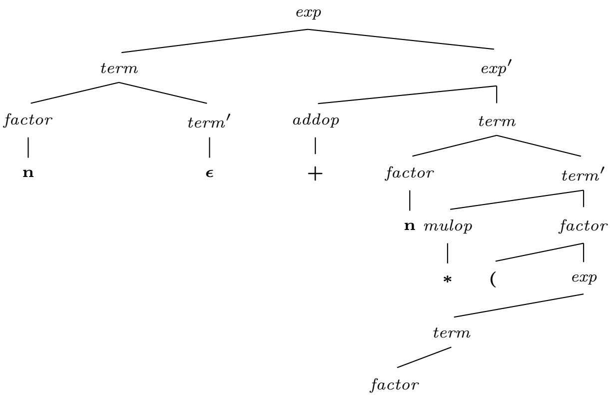
# Best viewed as a tree



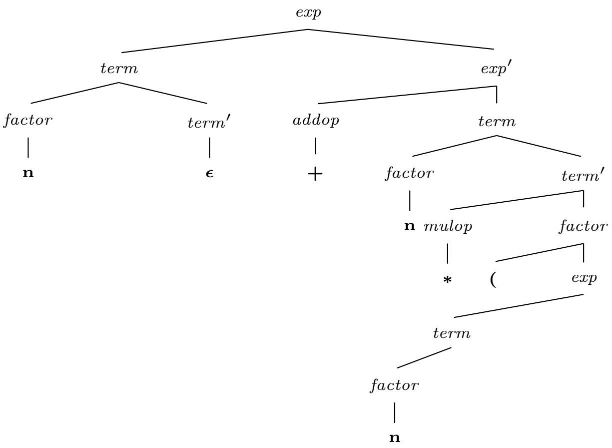
# Best viewed as a tree



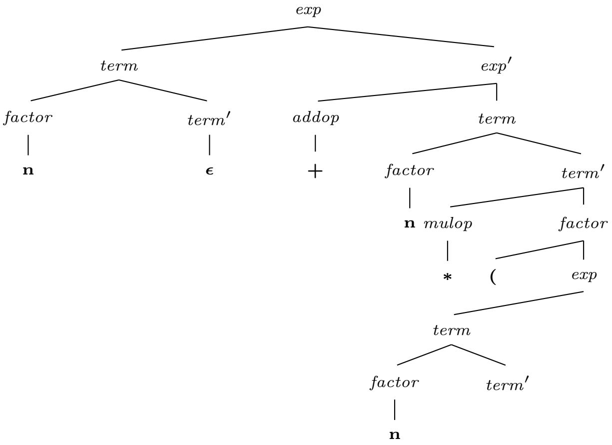
# Best viewed as a tree



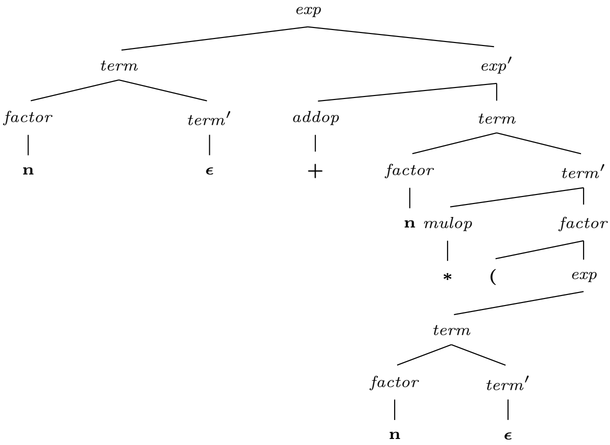
# Best viewed as a tree



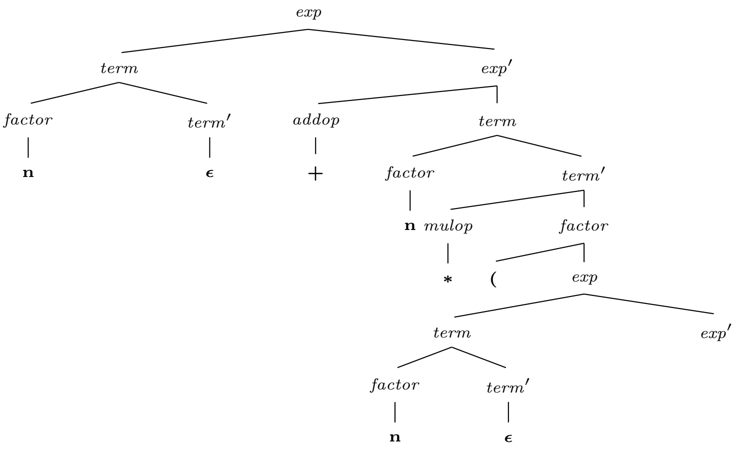
# Best viewed as a tree



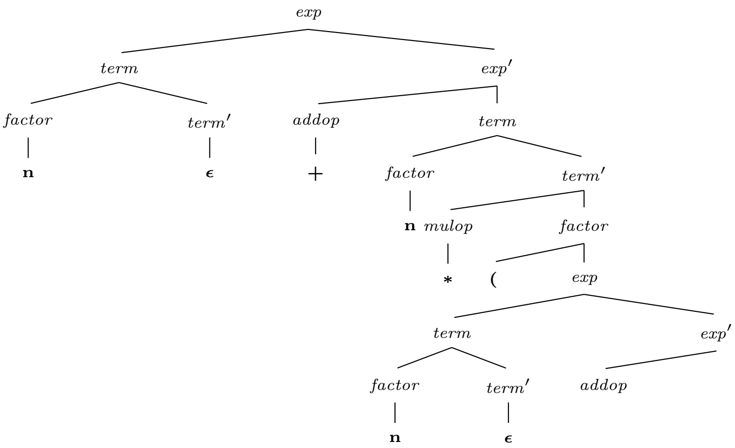
# Best viewed as a tree



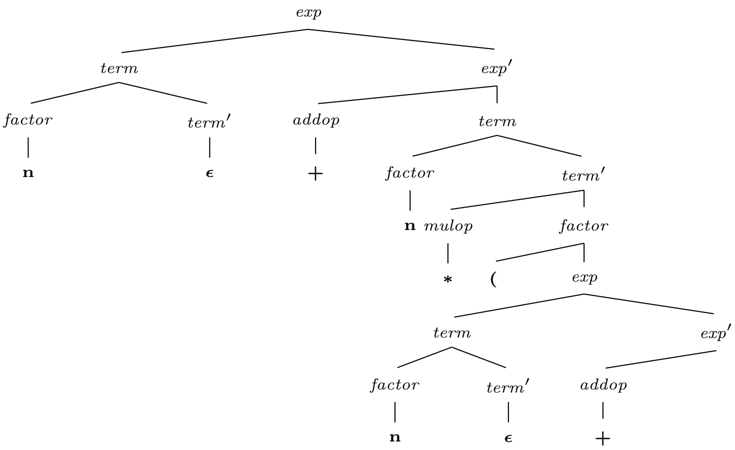
# Best viewed as a tree



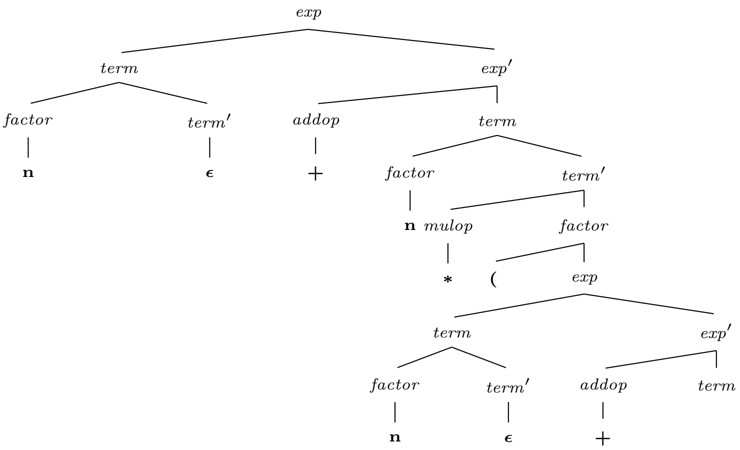
# Best viewed as a tree



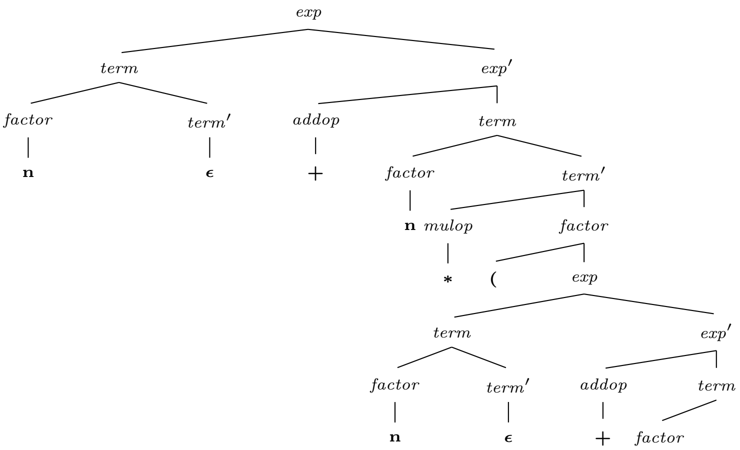
# Best viewed as a tree



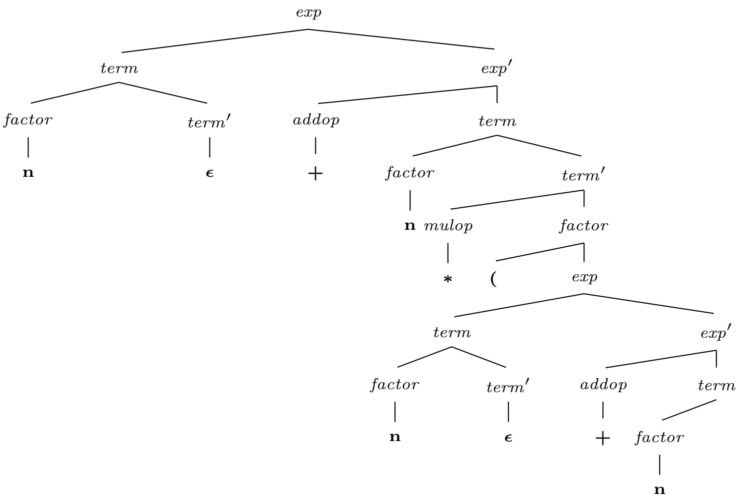
# Best viewed as a tree



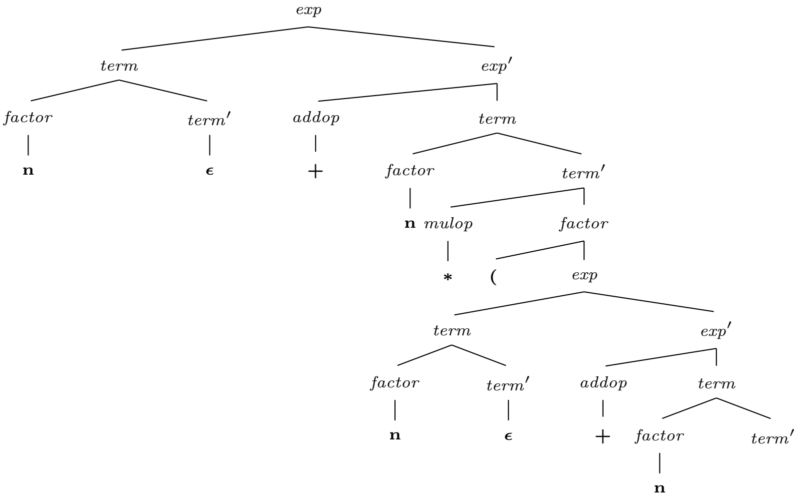
# Best viewed as a tree



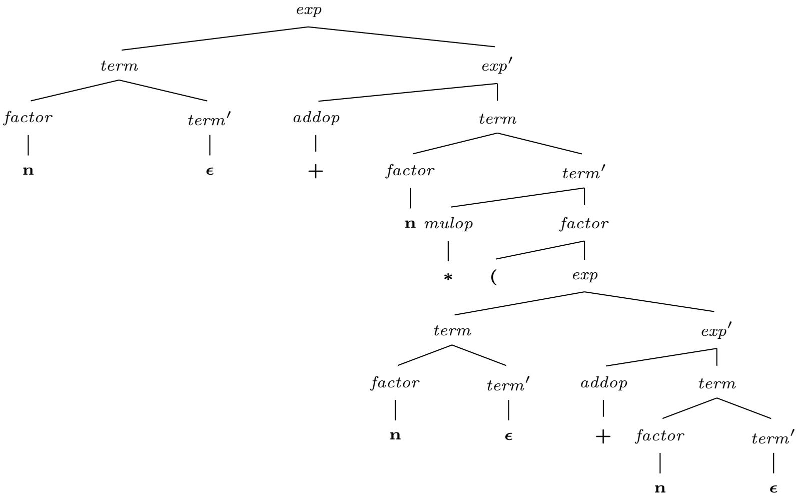
# Best viewed as a tree



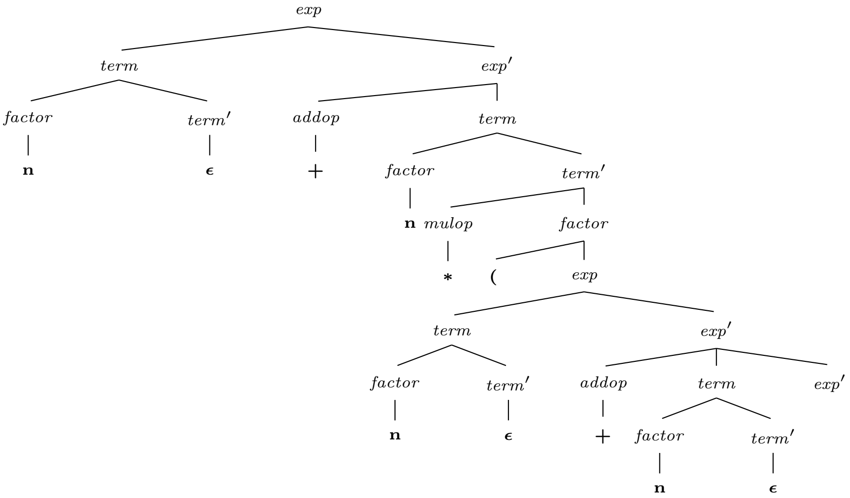
# Best viewed as a tree



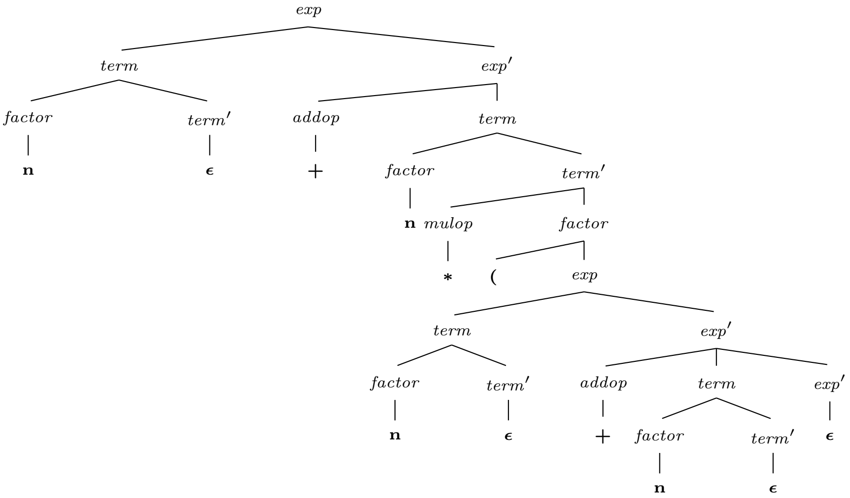
# Best viewed as a tree



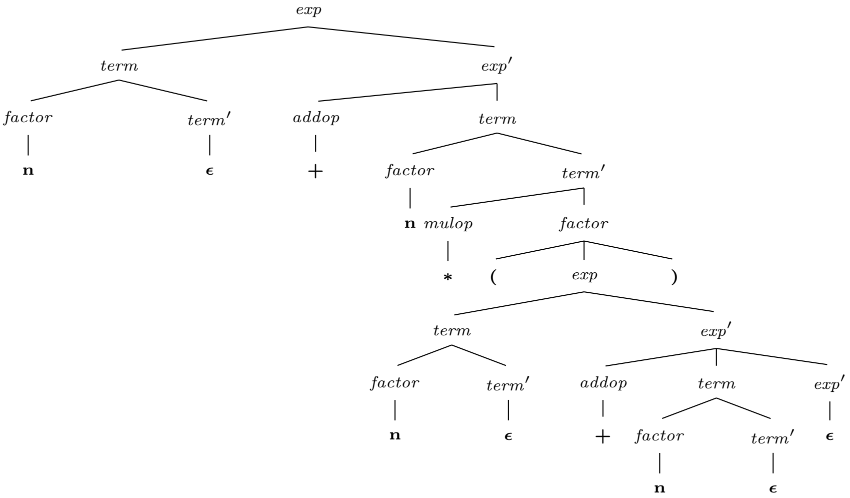
# Best viewed as a tree



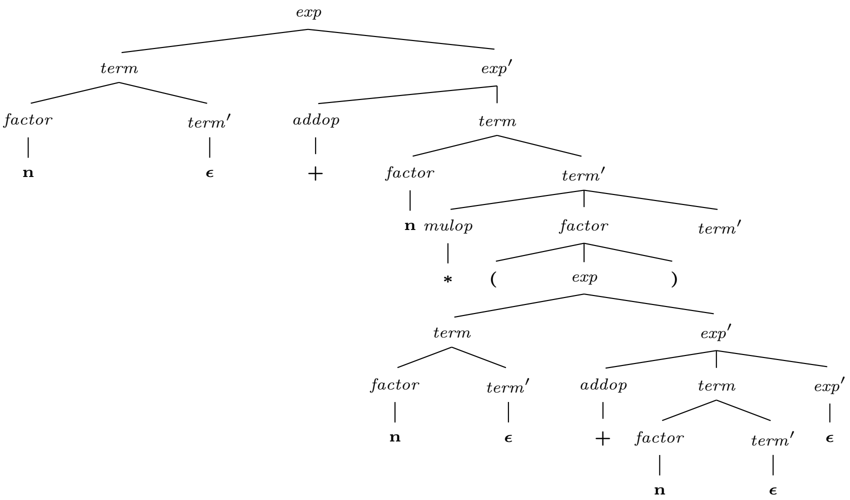
# Best viewed as a tree



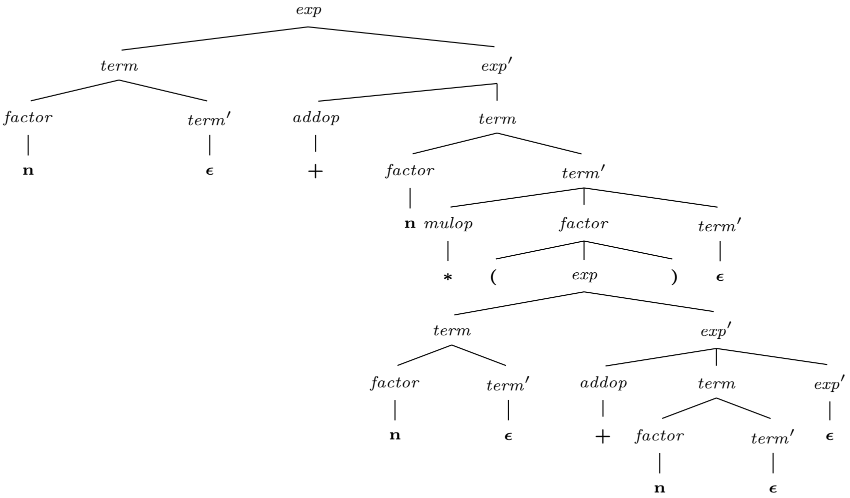
# Best viewed as a tree



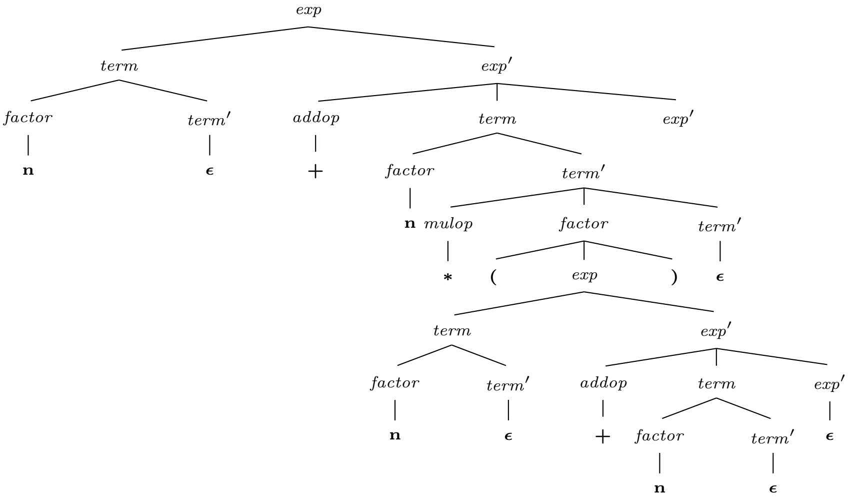
# Best viewed as a tree



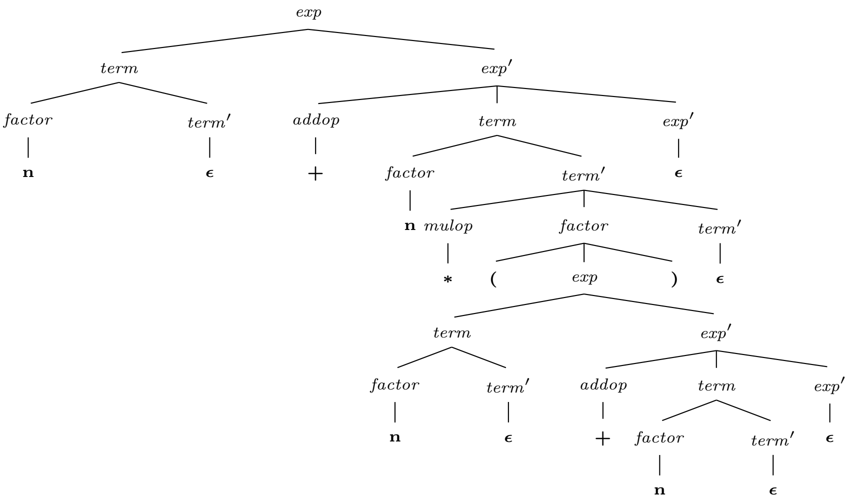
# Best viewed as a tree



# 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?

# Oracular derivation

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

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

# 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).

# Left-most derivation

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

# Non-determinism vs. ambiguity

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

**Lemma (Left or right, who cares)**

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

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

**Using production $A \to \beta$**

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

# 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)**

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

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

**Using production $A \to \beta$**

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

# 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!

# 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.

# Section

## First and follow sets

Chapter 4 “Parsing (will be polished/updated )”
Course “Compiler Construction”
Martin Steffen
Spring 2024

# 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 $X$**

The **first-set** of a symbol $X$ is the set of terminal symbols can appear at the **start** of strings *derived from* $X$.

**Follow-set of $A$**

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

- sentential form: word *derived from* 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)

# First sets

## Definition (First set)

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

$$First_G(X) = \{a \mid X \Rightarrow^*_G a\alpha, \quad a \in \Sigma_T\} \, . \qquad (2)$$

## 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:

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

# Deceptively simple example (from before)

- no nullable symbols
- another crucial aspect that oversimplifies the problem

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

# Conditions on first-sets (no nullability)





## Constraints

1. $$First(a) \supseteq \{a\}.$$
2. For $A \rightarrow X\beta$ then $First(A) \supseteq First(X)$.

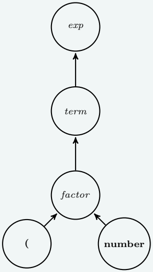
$$\begin{array}{rcl}
exp & \rightarrow & exp + term \mid exp - term \mid term \\
term & \rightarrow & term * factor \mid factor \\
factor & \rightarrow & (\, exp \,) \mid \mathbf{number}
\end{array}$$

# Conditions on first-sets (no nullability)

## Constraints

1. $$First(a) \supseteq \{a\}.$$
2. For $A \to X\beta$ then $First(A) \supseteq First(X)$.

## Dependencies for $First$




## "calculation"

```
F_factor := { "(",  } ∪ { "number" }
F_term   :=   F_factor
F_expr   :=   F_term
```

# More complex variation of previous example

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

but still no nullability

# Conditions on first-sets (no nullability)

1. $$First(a) \supseteq \{a\}.$$
2. For $A \to X\beta$ then $First(A) \supseteq First(X)$.

# Constraints for the example

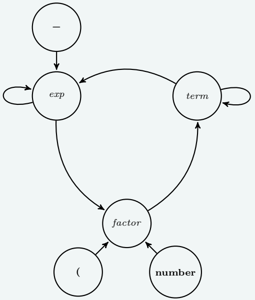
## Grammar

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

## constraints

$$
\begin{array}{rcl}
exp & \supseteq & \{-\} \\
exp & \supseteq & exp \\
exp & \supseteq & term \\
term & \supseteq & term \\
term & \supseteq & factor \\
factor & \supseteq & \{(\} \\
factor & \supseteq & \{\mathbf{number}\} \\
factor & \supseteq & exp
\end{array}
$$

## Dependencies for *First*



# When to terminate?

```
F_factor := { "(",  } ∪ { "number" }
F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp
```

# When to terminate?

```
F_factor := { "(",  } ∪ { "number" }
F_term   := F_factor
F_expr   := { "-" } ∪ F_term
F_factor := F_factor ∪ F_exp

F_term   := F_factor
```

# When to terminate?

```
F_factor := { "(",  } ∪ { "number" }
F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp

F_term   := F_factor
F_expr   := { "−" } ∪ F_term
```

INF5110 – Compiler Construction

# When to terminate?

```
F_factor := { "(",  } ∪ { "number" }
F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp

F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_expr   := { "−" } ∪ F_term
```

# When to terminate?

```
F_factor := { "(",  } ∪ { "number" }
F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp

F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp
```

# When to terminate?

```
F_factor := { "(",  } ∪ { "number" }
F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp

F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp
 // continue??
```

## Some observations

- No point to continue, continuing the update don't add need information
- actually, after updating `F_factor` the second time (line 4), the information has stabilized
- all constraints satisfied d.h. solved, (after line 4)

# When to terminate?

```
F_factor := { "(",  } ∪ { "number" }
F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_factor := F_factor ∪ F_exp
```

## Some observations

- No point to continue, continuing the update don't add need information
- actually, after updating `F_factor` the second time (line 4), the information has stabilized
- all constraints satisfied d.h. solved, (after line 4)

# When to terminate?

```
F_factor := F_factor ∪ F_exp
F_term   := F_factor
F_expr   := { "−" } ∪ F_term
F_factor := { "(",  } ∪ { "number" }
```

## Some observations

- No point to continue, continuing the update don't add need information
- actually, after updating `F_factor` the second time (line 4), the information has stabilized
- all constraints satisfied d.h. solved, (after line 4)

# When to terminate?

```
F_factor  :=  F_factor ∪ F_exp
F_term    :=  F_factor
F_expr    :=  { "−" } ∪ F_term
F_factor  :=  { "(",  } ∪ { "number" }
```

## Some observations

- No point to continue, continuing the update don't add need information
- actually, after updating `F_factor` the second time (line 4), the information has stabilized
- all constraints satisfied d.h. solved, (after line 4)
- whether updating `F_factor` 2 times is enough, depends on the order of updates

# Section

## First and follow sets

Chapter 4 “Parsing (will be polished/updated )”
Course “Compiler Construction”
Martin Steffen
Spring 2024

# 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

## Definition (First set)

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

$$First_G(A) = \{a \mid A \Rightarrow^*_G a\alpha, \quad a \in \Sigma_T\} + \{\epsilon \mid A \Rightarrow^*_G \epsilon\} \ . \tag{3}$$

## 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:

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

# Remarks

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

$\Rightarrow$ recursive definition

# A more algorithmic/recursive definition (HERE)

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

input../script/parsing/definitions/firstset-symbol-rec

# For words

**Definition (First set of a word)**

Given a grammar $G$ and word $\alpha$. The *first-set* of

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

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

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

```
initialize(First);
while there are changes to any First[A] do
  for each production A → X_1 ... X_n do
       First[A] := First[A] ∪ First[X_1]
  end;
end
```

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

# Initialization

```
for all X ∈ Σ_T ∪ {ϵ} do
    First[X] := {X}
end;

for all non-terminals A do
   First[A] := {}
end
```

# Pseudo code

```
initialize(First);
while there are changes to any First[A] do
  for each production A → X_1 ... X_n do
    k := 1;
    continue := true
    while continue = true and k ≤ n do
      First[A] := First[A] ∪ First[X_k] \ {ε}
      if ε ∉ First[X_k] then continue := false
      k := k + 1
    end;
    if   continue = true
    then First[A] := First[A] ∪ {ε}
  end;
end
```

# Example expression grammar (from before)

$$
\begin{array}{rcl}
exp & \rightarrow & exp \;\; addop \;\; term \mid term \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & term \;\; mulop \;\; factor \mid factor \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\, exp \,) \mid \mathbf{number}
\end{array}
\tag{4}
$$

# Example expression grammar (expanded)

$$
\begin{array}{rcl}
exp & \rightarrow & exp \;\; addop \;\; term \\
exp & \rightarrow & term \\
addop & \rightarrow & + \\
addop & \rightarrow & - \\
term & \rightarrow & term \;\; mulop \;\; factor \\
term & \rightarrow & factor \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\, exp \,) \\
factor & \rightarrow & \textbf{number}
\end{array} \tag{5}
$$

| 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





**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

$$Follow_G(A) = \{a \mid S\,\$ \Rightarrow^*_G \alpha_1 A a \alpha_2, \quad a \in \Sigma_T + \{\,\$\,\}\} . \tag{6}$$

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

# Follow sets, recursively (HERE)

input../script/parsing/definitions/followset-nonterm

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

# More imperative representation in pseudo code

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

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

# Expression grammar once more

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

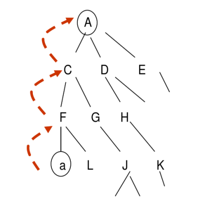
| nr | | pass 1 | pass 2 |
|---|---|---|---|
| 1 | $exp \rightarrow exp\ addop\ term$ | | |
| 2 | $exp \rightarrow term$ | | |
| 5 | $term \rightarrow term\ mulop\ factor$ | | |
| 6 | $term \rightarrow factor$ | | |
| 8 | $factor \rightarrow (\ exp\ )$ | | |

# "Run" of the algo

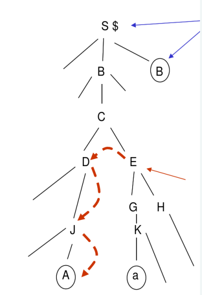
| Grammar rule | Pass 1 | Pass 2 |
|---|---|---|
| *exp* → *exp addop term* | Follow(*exp*) = {\$, +, -}<br>Follow(*addop*) = {(, **number**}<br>Follow(*term*) = {\$, +, -} | Follow(*term*) = {\$, +, -, *, )} |
| *exp* → *term* | | |
| *term* → *term mulop factor* | Follow(*term*) = {\$, +, -, *}<br>Follow(*mulop*)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

# Illustration of first/follow sets

$a \in First(A)$



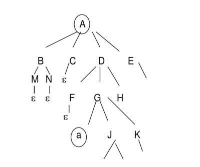

$a \in Follow(A)$



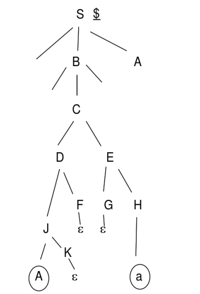

- red arrows: illustration of *information flow* in the algos
- run of $Follow$:
  - relies on $First$
  - in particular $a \in First(E)$ (right tree)
- $\$ \in Follow(B)$

# More complex situation (nullability)

**$a \in First(A)$**




**$a \in Follow(A)$**



# Section

## Massaging grammars

Chapter 4 “Parsing (will be polished/updated )”
Course “Compiler Construction”
Martin Steffen
Spring 2024

# Some forms of grammars are less desirable than others

- left-recursive production:

$$A \to A\alpha$$

more precisely: example of *immediate* left-recursion

- 2 productions with common "left factor":

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

# Some simple examples for both

- left-recursion

$$exp \to exp + term$$

- classical example for common left factor: rules for conditionals

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

# Transforming the expression grammar

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

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

# After removing left recursion

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

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

# Left-recursion removal

## Left-recursion removal

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

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

# Left-recursion removal: simplest case

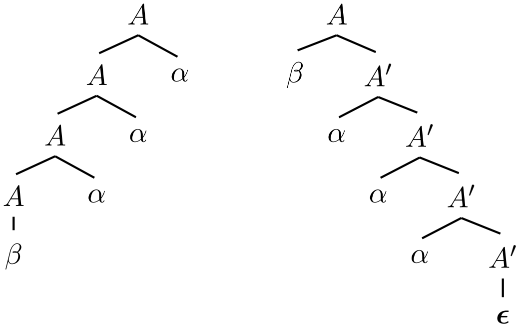
$$A \rightarrow A\alpha \mid \beta$$

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

# Schematic representation

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

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



# Remarks

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

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

- 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 $\epsilon$-productions
- more concrete example for such a production: grammar for *expressions*

# Left-recursion removal: immediate recursion (multiple)



**Before**

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

**After**

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

Note: can be written in *EBNF* as:

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

# Removal of: general left recursion

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

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

"current" = rule in the current stage of algo

# Example (for the general case)

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

# Example (for the general case)

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

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

# 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)

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

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

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

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

# Left factor removal

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

⇒ common left factor undesirable

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

## Simple situation

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

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

# Example: sequence of statements

**Before**

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

**After**

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

# 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

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

# Example: conditionals (without else)

## Before

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

## After

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

# Not all factorization doable in "one step"

## Starting point

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

## After 1 step

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

## After 2 steps

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

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

# Left factorization

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

# Section

## LL-parsing (mostly LL(1))

Chapter 4 “Parsing (will be polished/updated )”
Course “Compiler Construction”
Martin Steffen
Spring 2024

# 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)

# Sample expression grammar again

## factors and terms

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

# Look-ahead of 1: straightforward, but not trivial

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

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

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

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

**Example: 2 productions for non-terminal *factor***

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

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).

# Recursive descent: general set-up

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

**Idea**

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

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

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

# Recursive descent (in C-like)

method `factor` for nonterminal *factor*

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

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

# Recursive descent (in ocaml)

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

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

# 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* $A \rightarrow \alpha_1$ *and* $A \rightarrow \alpha_2$ *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

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

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

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

  1. *left-factoring*:

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

# Recursive descent for left-factored *if-stmt*

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

# Left recursion is a no-go

## factors and terms

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

- 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.

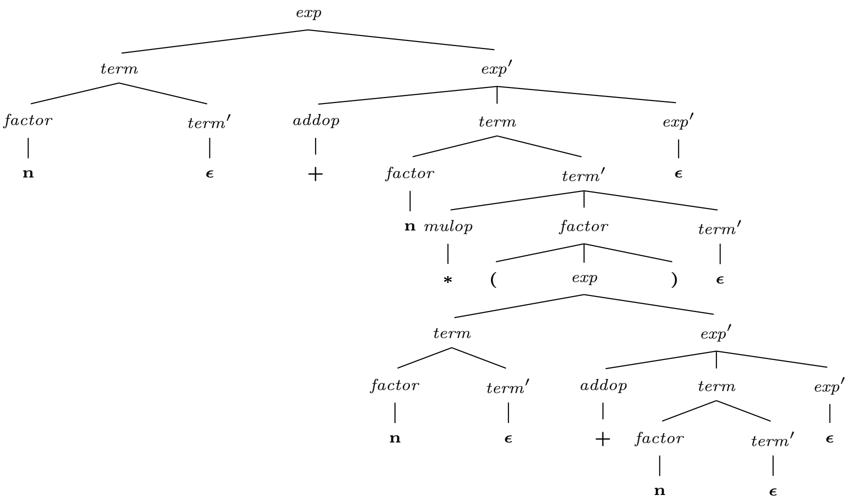
# Removing left recursion may help

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

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

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

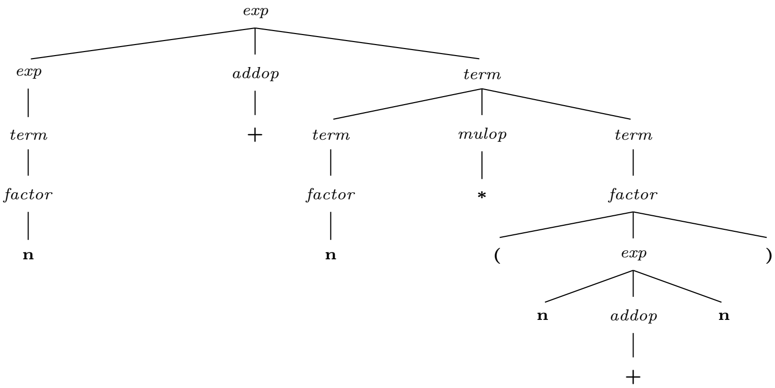
# Recursive descent works, alright, but ...




... who wants this form of trees?

# Left-recursive grammar with nicer parse trees

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



# Associtivity problematic

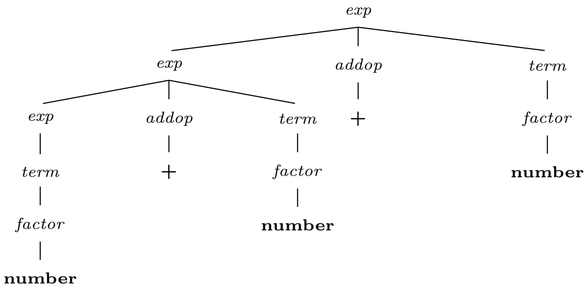
## Precedence & assoc.

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

$3 + 4 + 5$

parsed "as"

$(3 + 4) + 5$



# Associtivity problematic

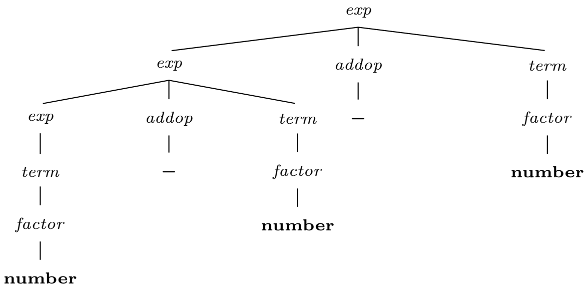
## Precedence & assoc.

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

$$3 - 4 - 5$$

parsed "as"

$$(3 - 4) - 5$$

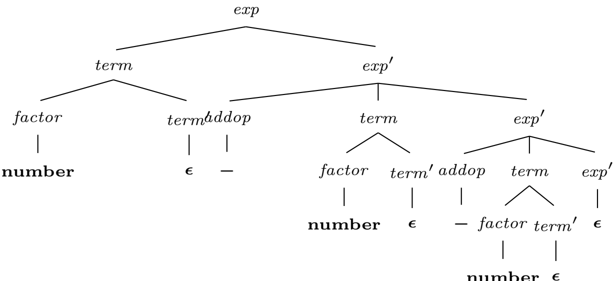


# Now use the grammar without left-rec (but right-rec instead)

## No left-rec.

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

$3 - 4 - 5$



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

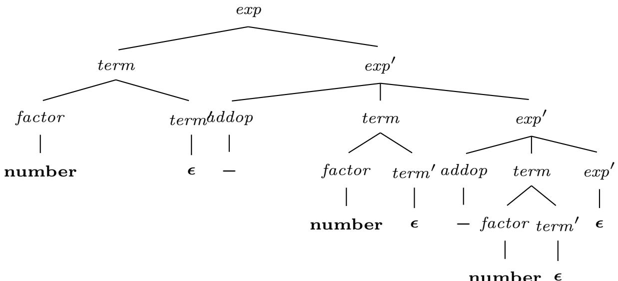
## No left-rec.

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

$3 - 4 - 5$

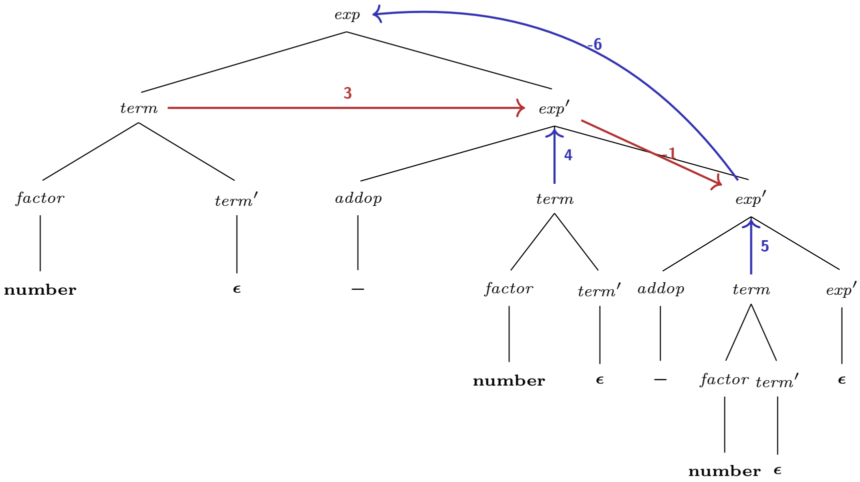
parsed "as"

$3 - (4 - 5)$



# 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

```
function exp' (valsofar: int): int;
begin
  if token = '+' or token = '-'
  then
    case token of
      '+': match ('+');
           valsofar := valsofar + term;
      '-': match ('-');
           valsofar := valsofar - term;
    end case;
    return exp'(valsofar);
  else return valsofar
end;
```

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

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

**trade offs:**

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

# AST vs. CST

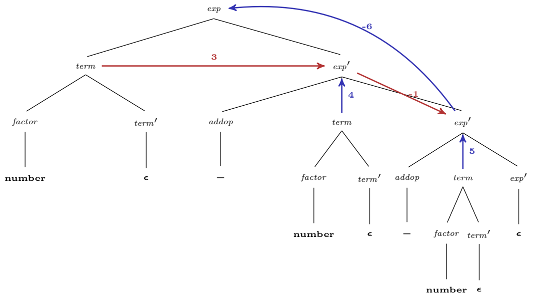
- once steps 1.–3. are fixed: *parse-trees* fixed!
- parse-trees = *essence* of grammatical derivation process
- often: parse trees only "conceptually" present in a parser
- AST:
  - *abstractions* of the parse trees
  - *essence* of the parse tree
  - actual tree data structure, as output of the parser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

**AST vs. CST/parse tree**

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

# AST: How “far away” from the CST?

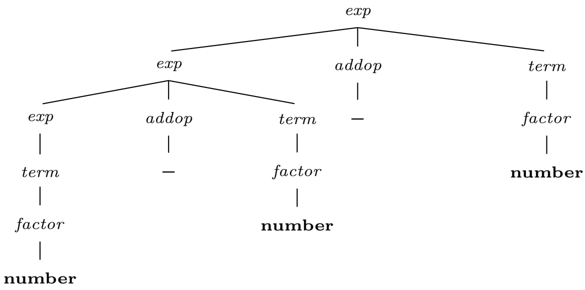
- AST: only thing relevant for later phases ⇒ better be *clean* ...
- AST “=” CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed “weirdly”,




parse-trees like that better be cleaned up as AST

# AST: How “far away” from the CST?

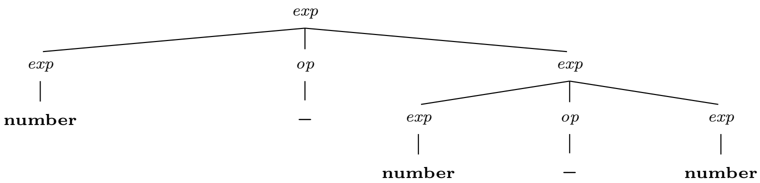
- AST: only thing relevant for later phases ⇒ better be *clean* ...
- AST “=” CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed “weirdly”,




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

# AST: How "far away" from the CST?

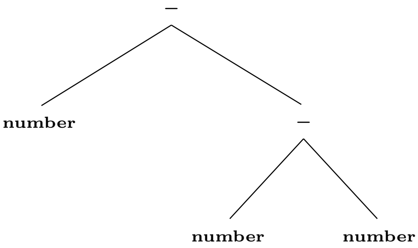
- AST: only thing relevant for later phases ⇒ better be *clean* ...
- AST "=" CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed "weirdly",




That parse tree looks reasonable clear and intuitive

# AST: How "far away" from the CST?

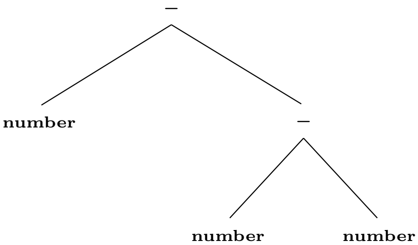
- AST: only thing relevant for later phases ⇒ better be *clean* ...
- AST "=" CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed "weirdly",




**Wouldn't that be the best AST here?**

# AST: How “far away” from the CST?

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

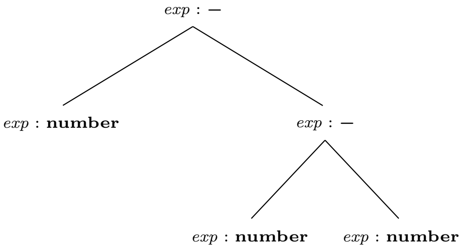



**Wouldn’t that be the best AST here?**

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 ⇒ 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**

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

- 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

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

## Recipe

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

# Example in Java

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

```
abstract public class Exp {
}
```

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

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

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

# Example in Java

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

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

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

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

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

# $3 - (4 - 5)$

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

# Pragmatic deviations from the recipe

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

⇒ that class is *not* needed

- some might prefer an implementation of

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

as simply integers, for instance arranged like

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

and used as `BinExpr.PLUS` etc.

# Recipe for ASTs, final words:

- space considerations for AST representations are 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)

# 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.

# Evaluating an *exp* during RD parsing

```
function exp() : int;
var temp: int
begin
  temp := term ();        { recursive call }
  while token = "+" or token = "−"
    case token  of
      "+": match ("+");
           temp := temp + term();
      "−": match ("−")
           temp := temp − term();
    end
  end
  return temp;
end
```

# Building an AST: expression

```
function exp() : syntaxTree;
var temp, newtemp: syntaxTree
begin
  temp := term ();          { recursive call }
  while token = "+" or token = "−"
    case token  of
      "+": match ("+");
           newtemp := makeOpNode("+");
           leftChild(newtemp)  := temp;
           rightChild(newtemp) := term();
           temp  := newtemp;
      "−": match ("−")
           newtemp := makeOpNode("−");
           leftChild(newtemp)  := temp;
           rightChild(newtemp) := term();
           temp  := newtemp;
    end
  end
  return temp;
end
```

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

# Building an AST: factor





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

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

# 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)

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

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

# Construction of the parsing table



## Table recipe

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

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

# Example: if-statements

- grammars is left-factored and not left recursive

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

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

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

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

```
push the start symbol of the parsing stack;
while  the top of the parsing stack ≠ $
  and the next input ≠ $
  if  the top of the parsing stack is terminal a
      and the next input token = a
  then
      pop the parsing stack;
      advance the input; // ``match'' ``eat''
  else if    the top the parsing is non-terminal A
       and   the next input token is a terminal or $
       and   parsing table M[A, a] contains
             production A → X1 X2 ... Xn
       then (* generate *)
             pop the parsing stack
             for i := n to 1 do
             push Xi onto the stack;
     else error
  if     the top of the stack = $
         and the next input token is $
```

# 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 |

INF5110 – Compiler Construction

# Expressions

## Original grammar

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

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

# Expressions

## Original grammar

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

left-recursive ⇒ not LL(k)

| | *First* | *Follow* |
|---|---|---|
| $exp$ | $(, \textbf{number}$ | $\$, )$ |
| $exp'$ | $+, -, \epsilon$ | $\$, )$ |
| $addop$ | $+, -$ | $(, \textbf{number}$ |
| $term$ | $(, \textbf{number}$ | $\$, ), +, -$ |
| $term'$ | $*, \epsilon$ | $\$, ), +, -$ |
| $mulop$ | $*$ | $(, \textbf{number}$ |
| [illegible] | [illegible] | [illegible] |

# Expressions

## Left-rec removed

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

| | *First* | *Follow* |
|---|---|---|
| *exp* | (, **number** | \$, ) |
| *exp′* | +, −, $\epsilon$ | \$, ) |
| *addop* | +, − | (, **number** |
| *term* | (, **number** | \$, ), +, − |
| *term′* | ∗, $\epsilon$ | \$, ), +, − |
| *mulop* | ∗ | (, **number** |
| *factor* | (, **number** | \$, ), +, −, ∗ |

# Expressions: LL(1) parse table

| M[N, T] | ( | **number** | ) | + | - | * | $ |
|---|---|---|---|---|---|---|---|
| *exp* | *exp* → *term exp'* | *exp* → *term exp'* | | | | | |
| *exp'* | | | *exp'* → ε | *exp'* → *addop term exp'* | *exp'* → *addop term exp'* | | *exp'* → ε |
| *addop* | | | | *addop* → + | *addop* → - | | |
| *term* | *term* → *factor term'* | *term* → *factor term'* | | | | | |
| *term'* | | | *term'* → ε | *term'* → ε | *term'* → ε | *term'* → *mulop factor term'* | *term'* → ε |
| *mulop* | | | | | | *mulop* → * | |
| *factor* | *factor* → ( *exp* ) | *factor* → **number** | | | | | |

# Section

## Error handling

Chapter 4 “Parsing (will be polished/updated )”
Course “Compiler Construction”
Martin Steffen
Spring 2024

# Error handling

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

# Error messages

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

# Section

## Bottom-up parsing

Chapter 4 “Parsing (will be polished/updated )”
Course “Compiler Construction”
Martin Steffen
Spring 2024

# Bottom-up parsing: intro

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

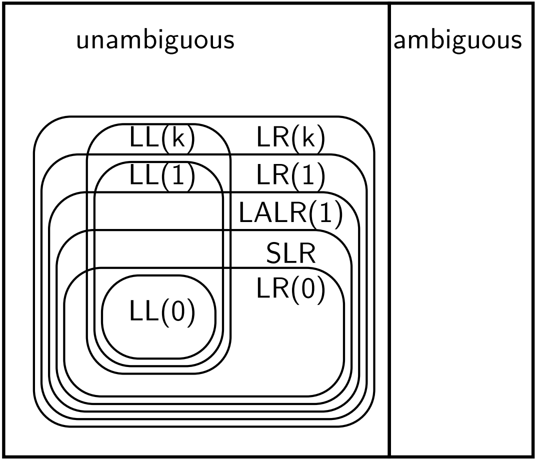
**LR(0)**
- only for very simple grammars
- approx. 300 states for standard programming languages
- only as warm-up for SLR(1) and LALR(1)

**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

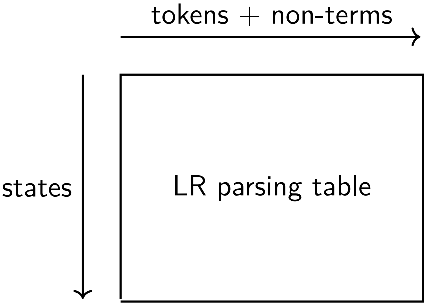
**LR(1)** covers all grammars, which can in principle be parsed by looking at the next token

# Grammar classes overview (again)



# LR-parsing and its subclasses

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

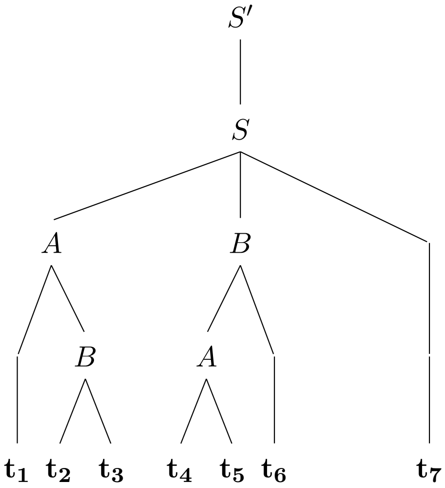


# Example grammar

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

- assume: grammar unambiguous
- assume word of terminals $\mathbf{t_1 t_2 \ldots t_7}$ 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 $S'$)

# Parse tree for $t_1 \dots t_7$



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

**Potentially puzzling question at first sight:**

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

**"Reduction"**

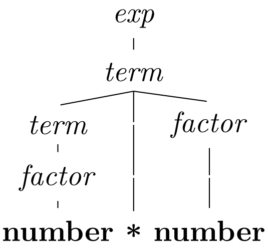
- short answer: parser builds the parse tree bottom-up
- derivation:
  - replacement of nonterminals by right-hand sides
  - *derivation*: builds (implicitly) a parse-tree *top-down*
- reduce step = bottom-up move = reverse derive step

**Right-sentential form: right-most derivation**

$$S \Rightarrow_r^* \alpha$$

# Example expression grammar (from before)

$$
\begin{aligned}
exp &\rightarrow exp \;\; addop \;\; term \mid term \\
addop &\rightarrow + \mid - \\
term &\rightarrow term \;\; mulop \;\; factor \mid factor \\
mulop &\rightarrow * \\
factor &\rightarrow (\, exp \,) \mid \mathbf{number}
\end{aligned}
\tag{10}
$$



# Bottom-up parse: Growing the parse tree

**number ∗ number**

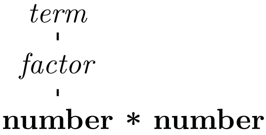
<u>**number**</u> **∗ number**

# Bottom-up parse: Growing the parse tree

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

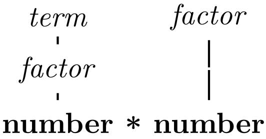
$$\underline{\textbf{number}} * \textbf{number} \quad \hookrightarrow \quad \underline{factor} * \textbf{number}$$

# Bottom-up parse: Growing the parse tree



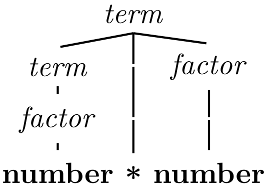

$$
\begin{aligned}
\underline{\textbf{number}} * \textbf{number} &\hookrightarrow \underline{factor} * \textbf{number} \\
&\hookrightarrow term * \underline{\textbf{number}}
\end{aligned}
$$

# Bottom-up parse: Growing the parse tree



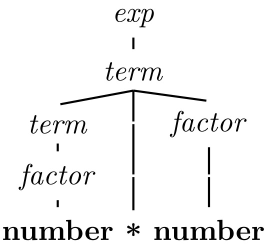

$$
\begin{aligned}
\underline{\textbf{number}} * \textbf{number} &\hookrightarrow \underline{\mathit{factor}} * \textbf{number} \\
&\hookrightarrow \mathit{term} * \underline{\textbf{number}} \\
&\hookrightarrow \underline{\mathit{term} * \mathit{factor}}
\end{aligned}
$$

# Bottom-up parse: Growing the parse tree




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

# Bottom-up parse: Growing the parse tree




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

# Reduction in reverse = right derivation

## Reduction

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

## Right derivation

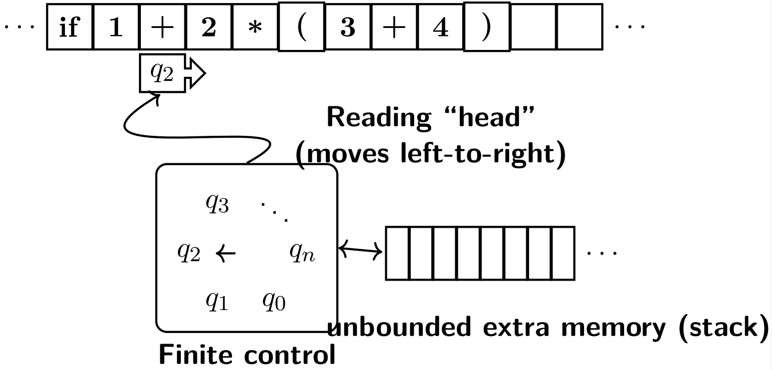
$$
\begin{array}{lll}
\mathbf{n} * \mathbf{n} & \Leftarrow_r & \underline{\mathit{factor}} * \mathbf{n} \\
 & \Leftarrow_r & \underline{\mathit{term}} * \mathbf{n} \\
 & \Leftarrow_r & \mathit{term} * \underline{\mathit{factor}} \\
 & \Leftarrow_r & \underline{\mathit{term}} \\
 & \Leftarrow_r & \underline{\mathit{exp}}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

## Underlined part

- *different* in reduction vs. derivation
- represents the "part being replaced"
  - for derivation: right-most non-terminal
  - for reduction: so-called handle (or part of it)

all intermediate words are *right-sentential forms*

# Schematic picture of parser machine (again)



# General LR "parser machine" configuration

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

# Schematic run (reduction: from top to bottom)

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

# 2 basic steps: shift and reduce

- parsers reads input and uses stack as intermediate storage
- so far: no mention of look-ahead, but that will play a role, as well

**Shift**

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

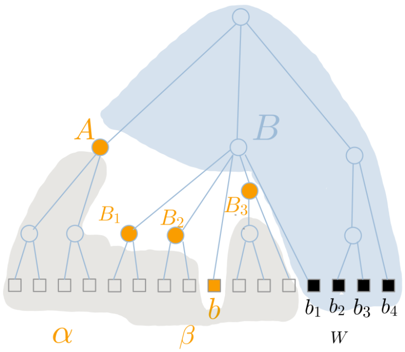
**Reduce**

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

# A typical situation during LR-parsing




$$B \to \underbrace{B_1\ B_2\ b\ B_3}_{\beta}$$

# Handle

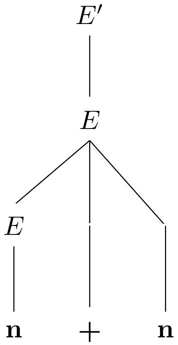
**Definition (Handle)**

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

- $w$ (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 ($\beta$ 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 $\beta$ (= bottom-up!)
- "handle"-part $\beta$ can be *empty* ($= \epsilon$)

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

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




| | 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 |

*note*: line 3 vs line 6!; both contain $E$ on (top of) the stack

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

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

# Example with $\epsilon$-transitions: parentheses

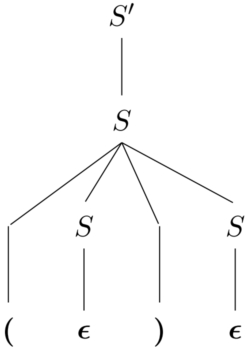
$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & (\, S\,)\, S \;\mid\; \epsilon \end{array}$$

side remark: unlike previous grammar, here:

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

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

# Parentheses: run and right-most derivation




| | 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 $\epsilon$ 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**

$$S \Rightarrow_r^* \alpha$$

right-sentential forms: part of the "run", 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 |

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

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

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

# 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

# But what are the states of an LR-parser?

### State

1. the state is determined by the “past”.

# But what are the states of an LR-parser?

## State

1. the state is determined by the "past".
2. the memory of the parser machine: stack (unbounded!)

# But what are the states of an LR-parser?

## State

1. the state is determined by the "past".
2. the memory of the parser machine: stack (unbounded!)
3. make it finite state: FSA on stack content.

# But what are the states of an LR-parser?

## State

1. the state is determined by the "past".
2. the memory of the parser machine: stack (unbounded!)
3. make it finite state: FSA on stack content.

## General idea

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

**State of parser $\hat{=}$ state of the thusly constructed FSA.**

# 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

# LR(0) items

## LR(0) item

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

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

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

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

## complete and initial items

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

# Grammar for parentheses: 3 productions

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

## 8 items

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

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

# Grammar for addition: 3 productions

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

## (coincidentally also:) 8 items

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

# 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

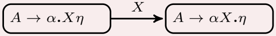
In a state marked by/containing item

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

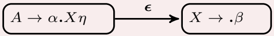
- $\beta$ on the *stack*
- $\gamma$: to be treated next (terminals on the input, but can contain also non-terminals(!))

# 2 kind of state transitions of the NFA

**Terminal or non-terminal**

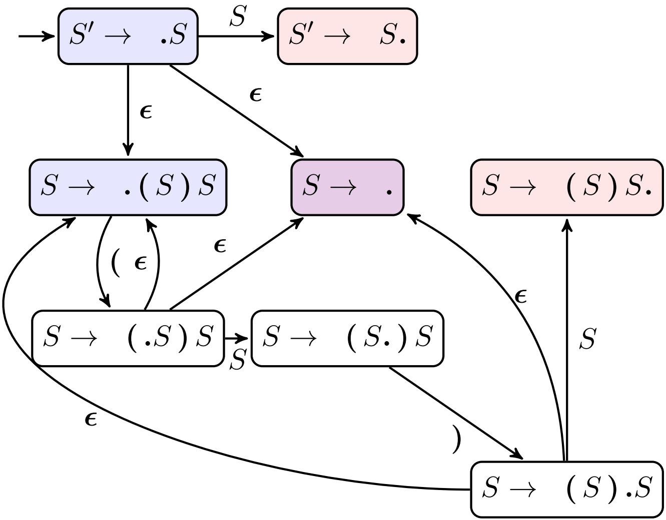



**$\epsilon$ $(X \to \beta)$**




- $X \in \Sigma$
- In case $X =$ *terminal* (i.e. token) $=$
  - the step on the left corresponds to a shift step
- for non-terminals: in that case, item $A \to \alpha.X\eta$ has two (kinds of) outgoing transitions

# NFA: parentheses



# Initial and final states

**initial states:**

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

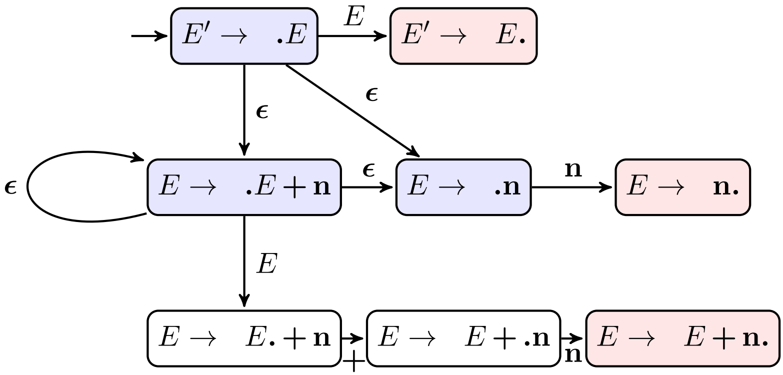
$\Rightarrow$ initial item $S' \to .S$ as (only) initial state

**final states/accepting actions:**

acceptance 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)

# NFA: addition

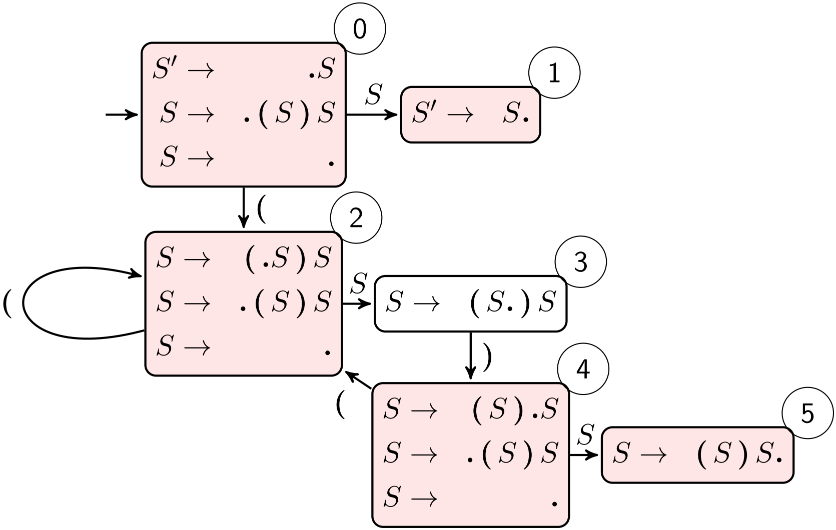


# Determinizing: from NFA to DFA

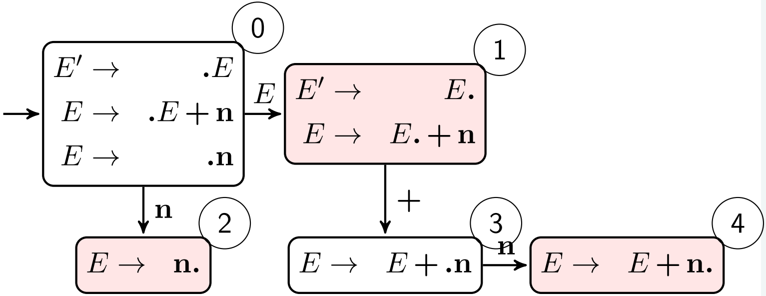
- standard subset-construction[5]
- states then contain *sets* of items
- important: $\epsilon$-closure
- also: *direct* construction of the DFA possible

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

# DFA: parentheses



# DFA: addition



# Direct construction of an LR(0)-DFA

- quite easy: just build in the closure directly…

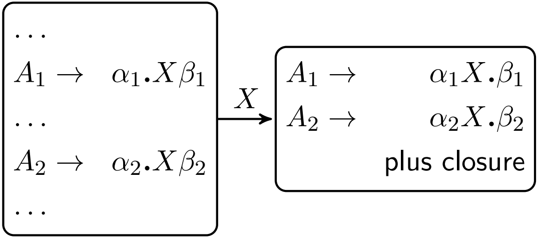
## $\epsilon$-closure

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

## initial state

→ [ $S' \to .S$ plus closure ]

# Direct DFA construction: transitions




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

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

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

**But: how does it hang together?**

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

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

and the reaction better be uniquely determined . . . .

# Stack contents and state of the automaton

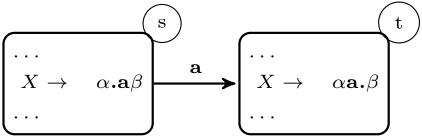
- remember: at any config. of stack/input in a run
  1. stack contains words from $\Sigma^*$
  2. DFA operates deterministically on such words
- the stack contains "abstraction of the past":
- when feeding that "past" on the stack into the automaton
  - starting with the oldest symbol (not in a LIFO manner)
  - starting with the DFA's initial state
  
  $\Rightarrow$ stack content determines state of the DFA
- actually: each prefix also determines uniquely a state
- top state:
  - state after the complete stack content
  - corresponds to the current state of the stack-machine
  
  $\Rightarrow$ crucial when determining *reaction*

# State transition corresponding to a shift

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

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

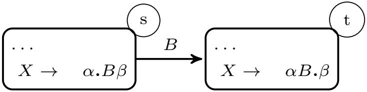
- construction thus has transition as follows




- shift possible (if $s$ is top-state)
- 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

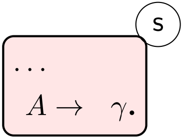
$X \to \alpha.B\beta$




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

# State (not transition) where reduce possible

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




- a complete right-hand side (“handle”) $\gamma$ 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 $\gamma$:
  - new top state!
  - remember the “goto-transition” (shift of a non-terminal)

# Remarks: states, transitions, and reduce steps

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

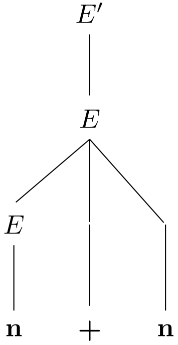
**No edges to represent (all of) a reduce step!**

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

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

# Example: LR parsing for addition (given the tree)

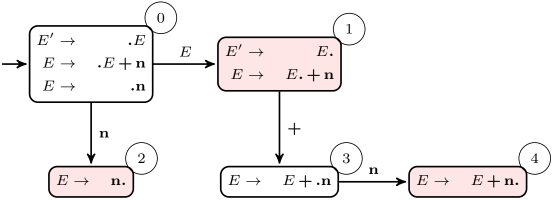
$$E' \rightarrow E$$
$$E \rightarrow E + \mathbf{n} \mid \mathbf{n}$$



| | parse stack | input | action |
|---|---|---|---|
| 1 | $\$$ | $\mathbf{n} + \mathbf{n}\,\$$ | shift |
| 2 | $\$\,\mathbf{n}$ | $+\mathbf{n}\,\$$ | red:. $E \rightarrow \mathbf{n}$ |
| **3** | $\$\,E$ | $+\mathbf{n}\,\$$ | shift |
| 4 | $\$\,E+$ | $\mathbf{n}\,\$$ | shift |
| 5 | $\$\,E+\mathbf{n}$ | $\$$ | 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) the 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

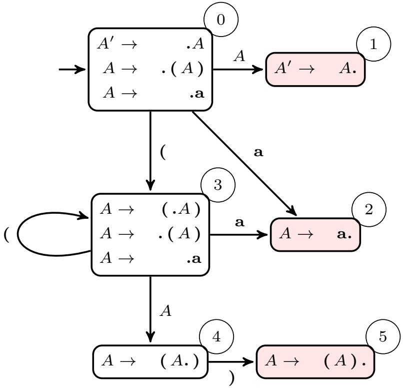
**LR(0) grammar**

The top-state alone determines the next step.

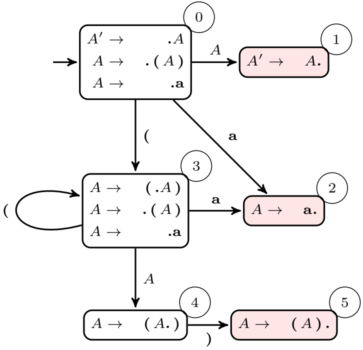
- thus: previous addition-grammar is *not LR(0)*

# Simple parentheses

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

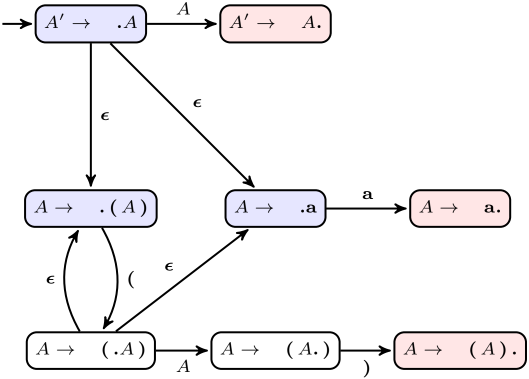


# Simple parentheses is LR(0)




| 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

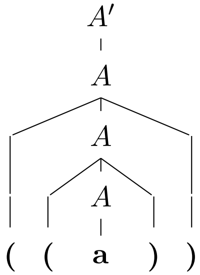
- table structure: slightly different for SLR(1), LALR(1), and LR(1) (see later)
- note: the "goto" part: "shift" on non-terminals (only 1 non-terminal $A$ here)
- corresponding to the $A$-labelled transitions

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

# Parsing of ( ( a ) )

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

# 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 $A \to \mathbf{a}$ |
| 5 | $\$_0(_3(_3A_4$ | $)\$$ | shift |
| 6 | $\$_0(_3(_3A_4)_5$ | $\$$ | reduce $A \to (A)$ |
| 7 | $\$_0(_3A_4$ | $\$$ | ???? |

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

**Invariant**

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

# LR(0) parsing algo, given DFA

let $s$ be the current head 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 head *state*: 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 $A \to \gamma$:
   - Reduction by $S' \to S$: accept, if input is empty; else error:
   - else:

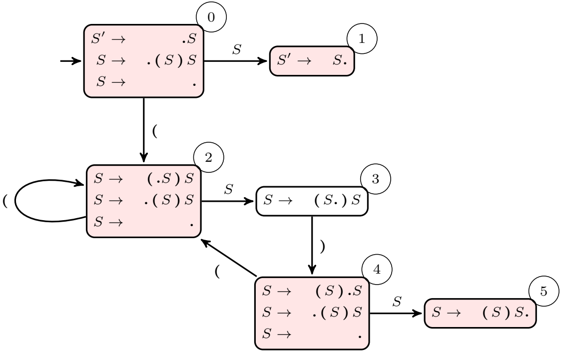
     **pop:** remove $\gamma$

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

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

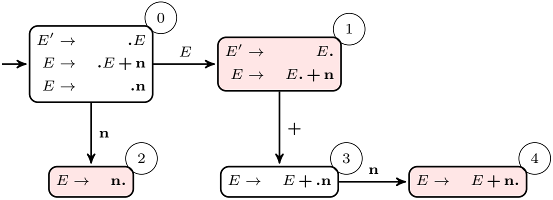
# DFA parentheses again: LR(0)?

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

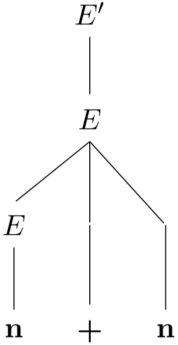


# DFA addition again: LR(0)?

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



# Non-deterministic choices



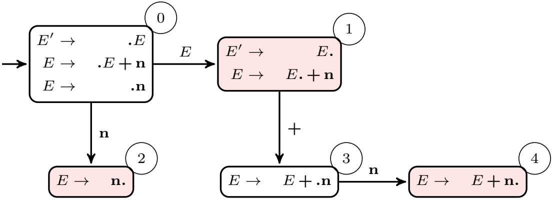

| | parse stack | input | action |
|---|---|---|---|
| 1 | $\mathbf{\$}$ | $\mathbf{n}+\mathbf{n}\,\mathbf{\$}$ | shift |
| 2 | $\mathbf{\$}\,\mathbf{n}$ | $+\mathbf{n}\,\mathbf{\$}$ | red:. $E \to \mathbf{n}$ |
| **3** | $\mathbf{\$}\,E$ | $+\mathbf{n}\,\mathbf{\$}$ | shift |
| 4 | $\mathbf{\$}\,E+$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

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

⇒ look-ahead on the input (without building the tree yet)

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

# Addition grammar (again)




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

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

# One look-ahead

- LR(0) too weak
- add look-ahead, here of *1 input symbol* (= token)
- different variations of that idea (with slight difference in expresiveness)
- tables slightly changed (compared to LR(0))
- but: *still* can use the LR(0)-DFAs

# Resolving LR(0) reduce/reduce conflicts

## LR(0) reduce/reduce conflict:

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

# Resolving LR(0) reduce/reduce conflicts

## LR(0) reduce/reduce conflict:

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

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

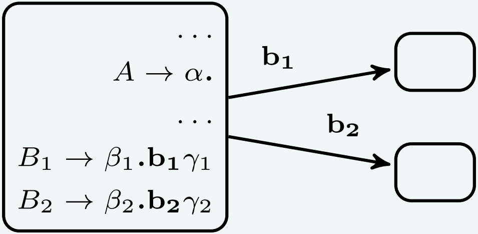
- If $Follow(A) \cap Follow(B) = \emptyset$

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

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

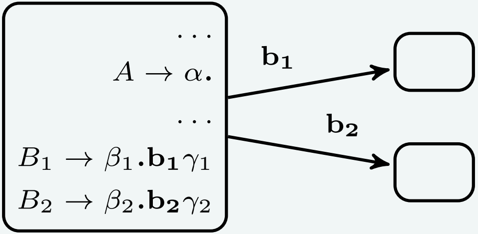
# Resolving LR(0) shift/reduce conflicts

## LR(0) shift/reduce conflict:



# Resolving LR(0) shift/reduce conflicts

## LR(0) shift/reduce conflict:



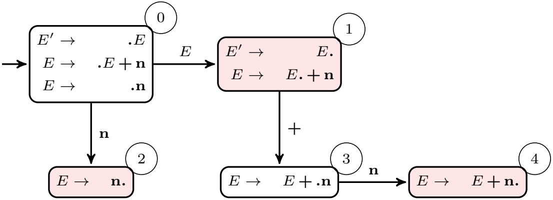

## SLR(1) solution: again: use follow sets of non-terms

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

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

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

# Revisit addition one more time




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

⇒
  - 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

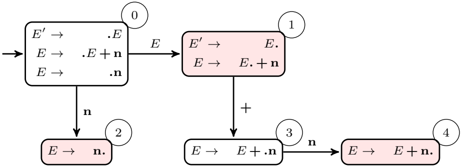
1. $s$ contains $A \to \alpha.X\beta$, where $X$ is a terminal and $X$ is the next token on the input, then
   - shift $X$ from input to top of stack. The new *state* pushed on the stack: state $t$ where $s \xrightarrow{X} t$
2. $s$ contains a *complete* item (say $A \to \gamma.$) and the next token 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 $\gamma$

     **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 | | $r:(E \to \mathbf{n})$ | | |
| 3 | $s:4$ | | | |
| 4 | | $r:(E \to E + \mathbf{n})$ | $r:(E \to E + \mathbf{n})$ | |

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

# Parsing table: remarks

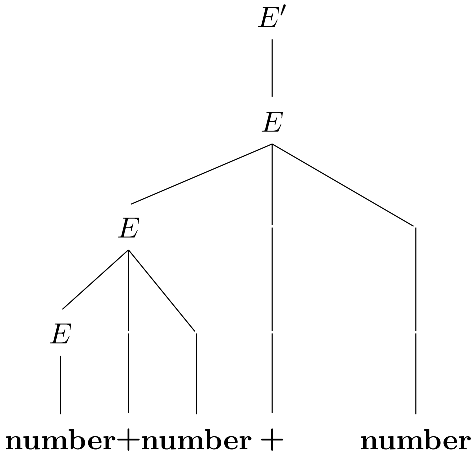
- SLR(1) parsing table: rather similar-looking to the LR(0) one
- differences: reflect the differences in: LR(0)-algo vs. SLR(1)-algo
- same number of rows in the table ( = same number of states in the DFA)
- only: rows "arranged" differently
  - LR(0): each state uniformely: either shift or else reduce (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) *reduce/reduce* and/or SLR(1) *shift-reduce* conflicts
  - would result in non-unique entries in SLR(1)-table

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

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

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

# Corresponding parse tree



# Revisit the parentheses again: SLR(1)?

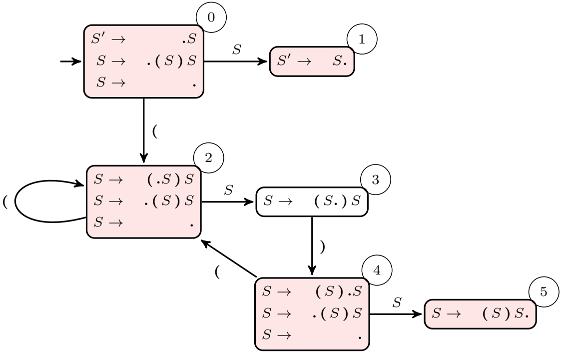
**Grammar: parentheses**

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

**Follow set**

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

# DFA for parentheses



# SLR(1) parse table

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

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

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

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

# 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)

# 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.

# Simplified conditionals

## Simplified "schematic" if-then-else

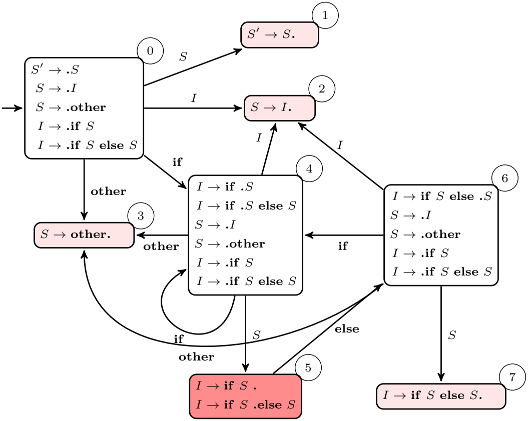
$$
\begin{array}{rcl}
S & \rightarrow & I \mid \textbf{other} \\
I & \rightarrow & \textbf{if}\ S \mid \textbf{if}\ S\ \textbf{else}\ S
\end{array}
$$

## Follow-sets

| | *Follow* |
|---|---|
| $S'$ | $\{\$\}$ |
| $S$ | $\{\$, \textbf{else}\}$ |
| $I$ | $\{\$, \textbf{else}\}$ |

- construction of LR(0)-DFA: non-SLR(1)
- since ambiguous: at least one conflict must be somewhere

# DFA of LR(0) items



# Simple conditionals: parse table

## Grammar

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

## SLR(1)-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 grammar

# Parser run (= reduction)

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

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

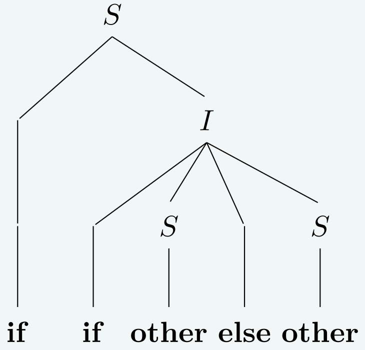
# Parser run, different choice

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

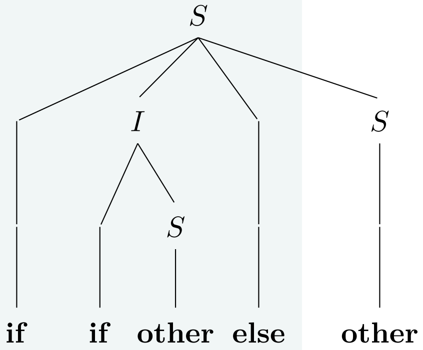
| *stage* | parsing stack | input | action |
|---|---|---|---|
| 1 | $\$_0$ | **if if other else other $** | shift: 4 |
| 2 | $\$_0\mathbf{if}_4$ | **if other else other $** | shift: 4 |
| 3 | $\$_0\mathbf{if}_4\mathbf{if}_4$ | **other else other $** | shift: 3 |
| 4 | $\$_0\mathbf{if}_4\mathbf{if}_4\mathbf{other}_3$ | **else other $** | reduce: 2 |
| 5 | $\$_0\mathbf{if}_4\mathbf{if}_4S_5$ | **else other $** | reduce 3 |
| 6 | $\$_0\mathbf{if}_4I_2$ | **else other $** | reduce 1 |
| 7 | $\$_0\mathbf{if}_4S_5$ | **else other $** | shift 6 |
| 8 | $\$_0\mathbf{if}_4S_5\mathbf{else}_6$ | **other $** | shift 3 |
| 9 | $\$_0\mathbf{if}_4S_5\mathbf{else}_6\mathbf{other}_3$ | **$** | reduce 2 |
| 10 | $\$_0\mathbf{if}_4S_5\mathbf{else}_6S_7$ | **$** | reduce 4 |
| 11 | $\$_0S_1$ | **$** | accept |

# Parse trees for the "simple conditions"

**shift-precedence: conventional**




**"wrong" tree**




**standard "dangling else" convention**

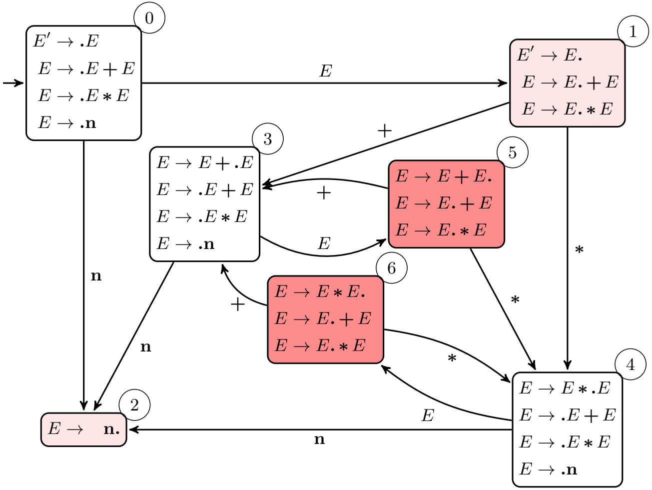
"an **else** belongs to the last previous, still open (= dangling) if-clause"

# 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` ...

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

# DFA for $+$ and $\times$



# States with conflicts

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

# Parse table + and ×

| 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

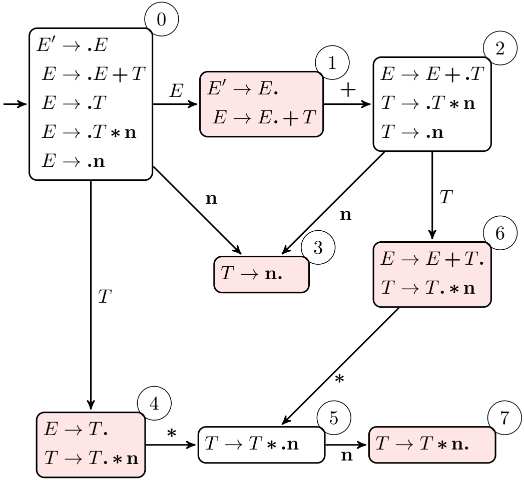
# Compare: unambiguous grammar for $+$ and $*$

## Unambiguous grammar: precedence and left-assoc built in

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

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

# DFA for unambiguous + and ×



# DFA remarks

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

$$\begin{aligned} \textbf{state 1:} &\quad \mathit{Follow}(E') = \{\$\} \\ \textbf{state 4:} &\quad \mathit{Follow}(E) = \{\$, +\} \\ \textbf{state 6:} &\quad \mathit{Follow}(E) = \{\$, +\} \\ \textbf{state 3/7:} &\quad \mathit{Follow}(T) = \{\$, +, *\} \end{aligned}$$

  - 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): look-ahead as afterthought**

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.

# Limitations of SLR(1) grammars

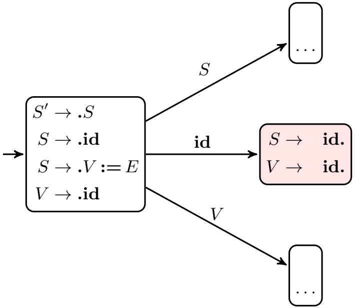
## Assignment grammar fragment

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

## Assignment grammar fragment, simplified

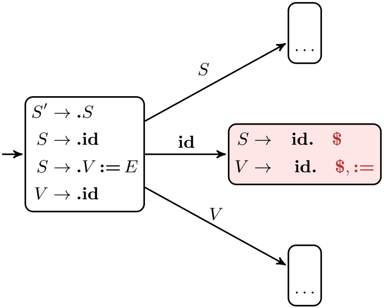
$$
\begin{aligned}
S &\rightarrow \textbf{id} \mid V \mathrel{\textbf{:=}} E \\
V &\rightarrow \textbf{id} \\
E &\rightarrow V \mid \textbf{n}
\end{aligned}
$$

# Non-SLR(1): Reduce/reduce conflict



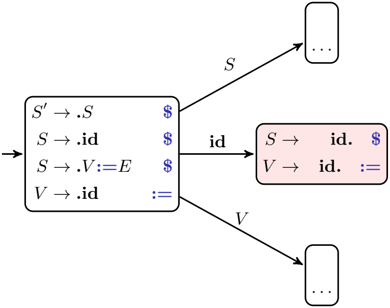

| | *First* | *Follow* |
|---|---|---|
| $S$ | **id** | **\$** |
| $V$ | **id** | **\$**, **:=** |
| $E$ | **id**, **number** | **\$** |

# Non-SLR(1): Reduce/reduce conflict




| | *First* | *Follow* |
|---|---|---|
| $S$ | **id** | **\$** |
| $V$ | **id** | **\$**, **:=** |
| $E$ | **id**, **number** | **\$** |

# Situation can be saved: more look-ahead



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

- LR(0)-items: too "indiscriminate" wrt. the follow sets
- remember the definition of SLR(1) conflicts
- LR(0)/SLR(1)-states:
  - sets of items[6] 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

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

# LR(1) items

- main idea: simply make the look-ahead part of the item
- obviously: proliferation of states[7]

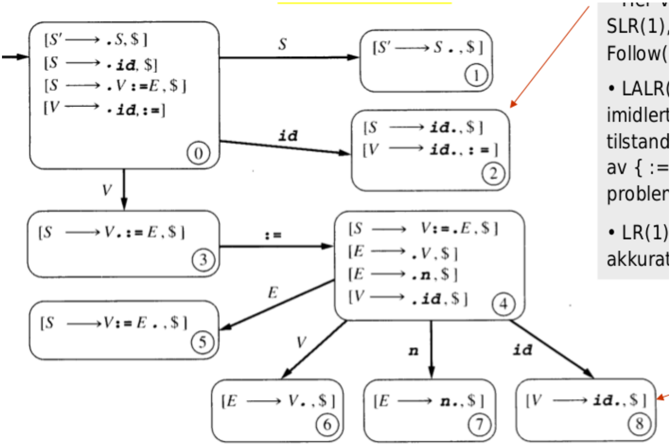
## LR(1) items

$$[A \to \alpha.\beta, \mathbf{a}] \tag{11}$$

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

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

# LR(1)-DFA


[S' ⟶ .S, $]
[S ⟶ .id, $]
[S ⟶ .V := E, $]
[V ⟶ .id, :=]
0
S
[S' ⟶ S ., $]
1
id
[S ⟶ id., $]
[V ⟶ id., :=]
2
V
[S ⟶ V. := E, $]
3
:=
[S ⟶ V:=.E, $]
[E ⟶ .V, $]
[E ⟶ .n, $]
[V ⟶ .id, $]
4
E
[S ⟶ V:= E ., $]
5
V
n
id
[E ⟶ V., $]
6
[E ⟶ n., $]
7
[V ⟶ id., $]
8
SLR(1),
Follow(
• LALR(
imidlert
tilstand
av { :=
probler
• LR(1)
akkurat

# 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

# 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**

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

# LR(1) transitions: $\epsilon$

## $\epsilon$-transition

for all

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

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

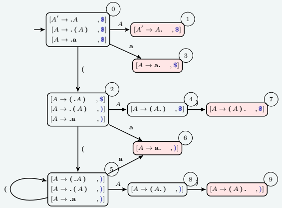
## including special case ($\gamma = \epsilon$)

$$\text{for all } B \to \beta \;\mid\; \ldots$$

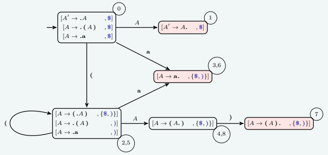
$$[A \to \alpha.B \quad ,\mathbf{a}] \xrightarrow{\epsilon} [B \to .\beta \quad ,\mathbf{a}]$$

# LALR(1) vs. LR(1)

## LR(1)




## LALR(1)




INF5110 – Compiler Construction

# Core of LR(1)-states

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

**Core of an LR(1) state**

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

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

# LALR(1)-DFA by 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[8]
  - 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)

[8] 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 specific *table* (LR(0), . . . ) is set-up, the parsing algorithms all work *the same*

# Error handling

## Minimal requirement

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

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

# Error messages

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

# Error recovery in bottom-up parsing

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

⇒ heuristic needed (like panic mode recovery)

## Panic mode idea

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

# Possible error situation

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

| state | input | | | | | goto | | | |
|---|---|---|---|---|---|---|---|---|---|
| | ... | ) | $\mathbf{f}$ | $\mathbf{g}$ | ... | ... | $A$ | $B$ | ... |
| ... | | | | | | | | | |
| 3 | | | | | | | $u$ | $v$ | |
| 4 | | | — | | | | — | — | |
| 5 | | | — | | | | — | — | |
| 6 | | — | — | | | | — | — | |
| ... | | | | | | | | | |
| $u$ | | — | — | reduce ... | | | | | |
| $v$ | | — | — | shift : 7 | | | | | |
| ... | | | | | | | | | |

# Possible error situation

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

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

# Panic mode recovery

## Algo

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

# Example again

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

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

# 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 | $\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3B_v$ | $\mathbf{gh}\ \ldots\$$ | back to normal |
| 3 | $\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3B_v\mathbf{g}_7$ | $\mathbf{h}\ \ldots\$$ | ... |

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

# Panic mode may loop forever

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

# Panicking and looping

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

- 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 detect 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 books 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 anyhow, and error recovery is not part of the oblig (halfway decent error *handling* is).

# 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.