INF5110 – Compiler Construction

Parsing

Spring 2016

Outline

- First and Follow set: general concepts for grammars
  - textbook looks at one parsing technique (top-down) [Louden, 1997, Chap. 4] before studying First/Follow sets
  - we: take First/Follow sets before any parsing technique
- two *transformation* techniques for grammars
- both *preserving* that accepted language
  1. removal for left-recursion
  2. left factoring

- 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 all non-terminals of a given grammar
- mostly straightforward
- one complication: *nullable* symbols (non-terminals)
- Note: those sets depend on grammar, not the language

Definition (First set)

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

$$First_G(A) = \{a \mid A \Rightarrow_G^* a\alpha, \quad a \in \Sigma_T\} + \{\epsilon \mid A \Rightarrow_G^* \epsilon\} . \tag{1}$$

Definition (Nullable)

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

Examples

- Cf. the Tiny grammar
- in Tiny, as in most languages

$$\mathit{Follow}(\mathit{if\text{-}stmt}) = \{\text{"}\textbf{if}\text{"}\}$$

- in many languages:

$$\mathit{Follow}(\mathit{assign\text{-}stmt}) = \{\textbf{identifier}, \text{"("}\}$$

- for statements:

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

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

$\Rightarrow$ recursive definition

A more algorithmic/recursive definition

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

Definition (First set of a symbol)

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

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

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

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

Definition (First set of a word)

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

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

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

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

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

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

```
for all non-terminals A do
   First[A] := {}            // counts as change
end
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] production of the form $A \to \epsilon$.

$$
\begin{aligned}
exp &\rightarrow exp\ addop\ term \;\mid\; term \\
addop &\rightarrow \mathbf{+} \;\mid\; \mathbf{-} \\
term &\rightarrow term\ mulop\ term \;\mid\; factor \\
mulop &\rightarrow \mathbf{*} \\
factor &\rightarrow \mathbf{(}\ exp\ \mathbf{)} \;\mid\; \mathbf{number}
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{exp}\ \mathit{addop}\ \mathit{term} \\
\mathit{exp} &\rightarrow \mathit{term} \\
\mathit{addop} &\rightarrow + \\
\mathit{addop} &\rightarrow - \\
\mathit{term} &\rightarrow \mathit{term}\ \mathit{mulop}\ \mathit{term} \\
\mathit{term} &\rightarrow \mathit{factor} \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow \textbf{(}\ \mathit{exp}\ \textbf{)} \\
\mathit{factor} &\rightarrow \textbf{number}
\end{aligned} \tag{3}
$$

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			$\{(, \textbf{number}\}$
addop	$\{+, -\}$		
term		$\{(, \textbf{number}\}$	
mulop	$\{*\}$		
factor	$\{(, \textbf{number}\}$		

- final results (at the end of pass 3):

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

Work-list formulation

```
for  all non-terminals A  do
   First[A] := {}
  WL        := P      // all productions
end
while WL ≠ ∅ do
  remove one (A → X_1 ... X_n) from WL
  if     First[A] ≠ First[A] ∪ First[X_1]
  then   First[A] := First[A] ∪ First[X_1]
      add all productions (A → X'_1 ... X'_m) to WL
  else   skip
end
```

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

Definition (Follow set (ignoring \$))

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

$$Follow_G(A) = \{a \mid S \Rightarrow_G^* \alpha_1 A a \alpha_2, \quad a \in \Sigma_T\} . \tag{4}$$

- More generally: \$ as special end-marker

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

- typically: start symbol *not* on the right-hand side of a production

Definition (Follow set of a non-terminal)

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

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

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

More imperative representation in pseudo code

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

Note! $First() = \epsilon$

Example expression grammar (expanded)

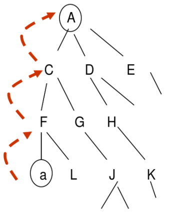
$$
\begin{aligned}
exp &\rightarrow exp\ addop\ term \\
exp &\rightarrow term \\
addop &\rightarrow + \\
addop &\rightarrow - \\
term &\rightarrow term\ mulop\ term \\
term &\rightarrow factor \\
mulop &\rightarrow * \\
factor &\rightarrow \mathbf{(}\ exp\ \mathbf{)} \\
factor &\rightarrow \mathbf{number}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

Run of the "algo"

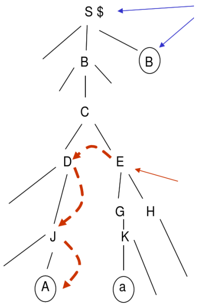
Grammar rule	Pass 1	Pass 2
exp → *exp addop term*	Follow(*exp*) = {\$, **+**, **-**} Follow(*addop*) = { **(**, **number**} Follow(*term*) = {\$, **+**, **-**}	Follow(*term*) = {\$, **+**, **-**, *****, **)** }
exp → *term*		
term → *term mulop factor*	Follow(*term*) = {\$, **+**, **-**, *****} Follow(*mulop*)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

Illustration of first/follow sets

$a \in First(A)$



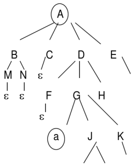
$a \in Follow(A)$



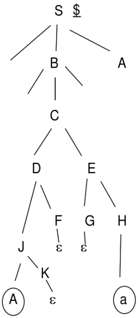
- red arrows: illustration of information flow in the algos
- run of *Follow*:
  - relies on *First*
  - in particular $a \in First(E)$ (right tree)
- $\$ \in Follow(B)$

More complex situation (nullability)

$a \in First(A)$



$a \in Follow(A)$



- left-recursive production:

$$A \to A\alpha$$

more precisely: example of *immediate* left-recursion

- 2 productions with common "left factor":

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

- left-recursion

$$exp \to exp + term$$

- classical example for common left factor: rules for conditionals

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

Transforming the expression grammar

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

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

After removing left recursion

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

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

Left-recursion removal

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

- price: ϵ-productions
- *3 cases* to consider
  - immediate (or direct) recursion
    - simple
    - general
  - *indirect* (or mutual) recursion

Left-recursion removal: simplest case

Before

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

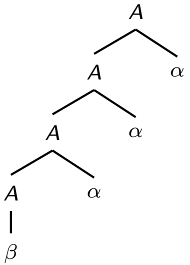
After

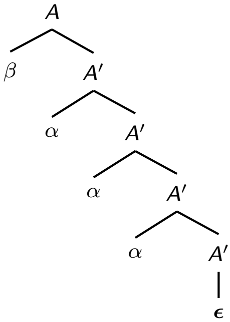
$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Schematic representation

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

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





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

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

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

Left-recursion removal: immediate recursion (multiple)

Before

$$\begin{array}{rcl} A & \to & A\alpha_1 \mid \dots \mid A\alpha_n \\ & \mid & \beta_1 \mid \dots \mid \beta_m \end{array}$$

After

$$\begin{array}{rcl} A & \to & \beta_1 A' \mid \dots \mid \beta_m A' \\ A' & \to & \alpha_1 A' \mid \dots \mid \alpha_n A' \\ & \mid & \epsilon \end{array}$$

Note, can be written in *EBNF* as:

$$A \to (\beta_1 \mid \dots \mid \beta_m)(\alpha_1 \mid \dots \mid \alpha_n)^*$$

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
    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 for A_j
  end
  { corresponds to i = j }
  remove, if necessary, immediate left recursion for A_i
end
```

"current" = rule in the current stage of algo

Example (for the general case)

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

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

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

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

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

Example (for the general case)

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

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

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

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

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

Example (for the general case)

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

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

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

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

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

Example (for the general case)

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

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

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

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

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

Left factor removal

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

$\Rightarrow$ common left factor undesirable

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

Simple situation

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

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

Example: sequence of statements

Before

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

After

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

Example: conditionals

Before

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

After

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

Example: conditionals (without else)

Before

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

After

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

Not all factorization doable in "one step"

Starting point

$$A \rightarrow \boldsymbol{a}\,\boldsymbol{b}\,\boldsymbol{c}\,B \mid \boldsymbol{a}\,\boldsymbol{b}\,C \mid \boldsymbol{a}\,E$$

After 1 step

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

After 2 steps

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

- note: we choose the *longest* common prefix (= longest left factor) in the first step

```
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
```

Outline

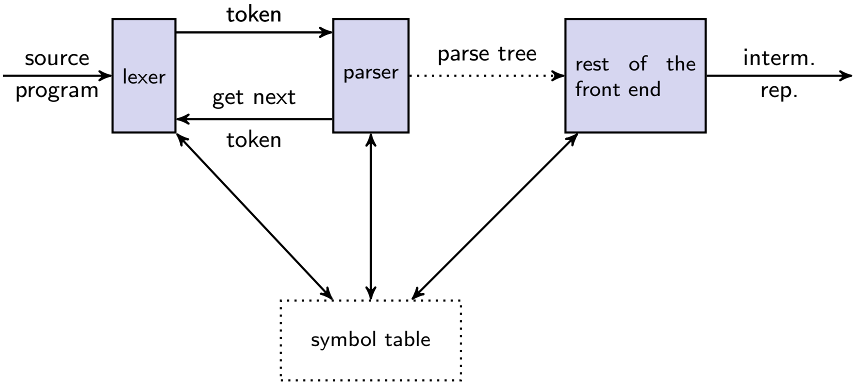
What's a parser generally doing

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 no used
- thus:
  - consider *restrictions* of CFGs, i.e., a specific subclass, and/or
  - *represented* in specific ways (no left-recursion, left-factored …)

Lexer, parser, and the rest



Top-down vs. bottom-up

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

Bottom-up

Parse tree is being grown from the leaves to the root.

Top-down

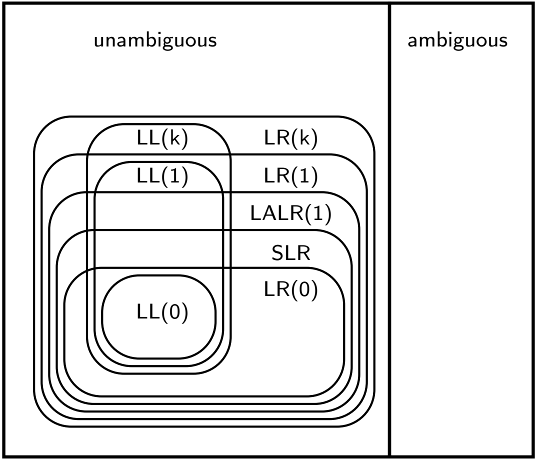
Parse tree is being grown from the root to the leaves.

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

- parser: better be "efficient"
- full complexity of CFLs: not really needed in practice[2]
- classification of CF languages vs. CF grammars, e.g.:
  - left-recursion-freedom: condition on a grammar
  - ambiguous language vs. ambiguious grammar
- classification of grammars ⇒ classification of *language*
  - a CF language is (inherently) ambiguous, if there's not 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 tool:
  - based on accepted notation for grammars: (BNF or allows EBNF etc.)

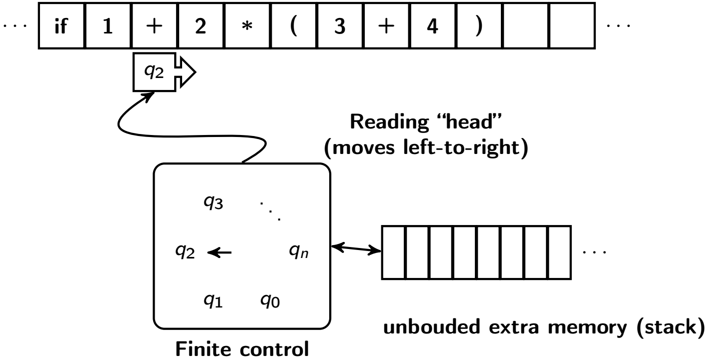
[2] Perhaps: if a parser has trouble to figure out if a program has a syntax error or not (perhaps using back-tracking), probably humans will have similar problems. So better keep it simple. And time in a compiler is better spent elsewhere (optimization, semantical analysis).

- *maaaany* have been proposed & studied, including their relationships
- lecture concentrates on
  - top-down parsing, in particular
    - LL(1)
    - recursive descent
  - bottom-up parsing
    - LR(1)
    - SLR
    - LALR(1) (the class covered by yacc-style tools)
- grammars typically written in *pure* BNF

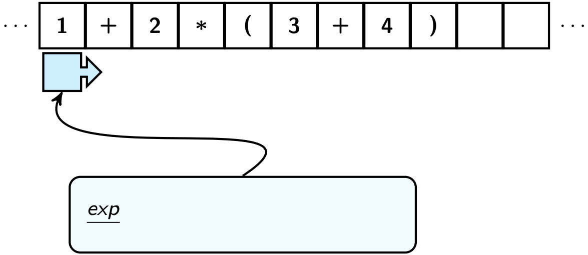


taken from [Appel, 1998]

- 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]

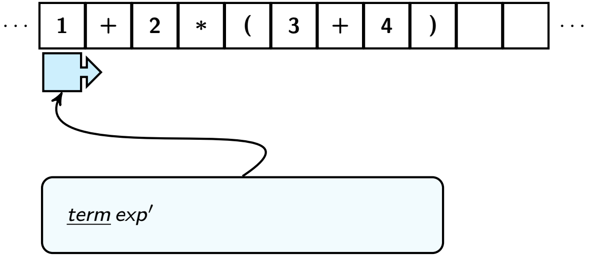


Note: sequence of *tokens* (not characters)



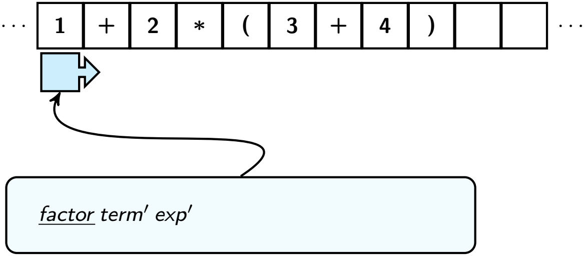
factors and terms

$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow (\ exp\ ) \mid \mathbf{number}
\end{aligned}
\tag{5}
$$



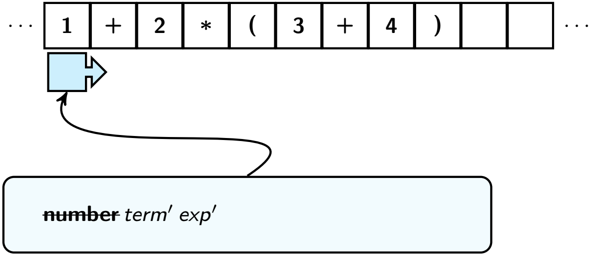
factors and terms

$$
\begin{aligned}
\textit{exp} &\rightarrow \textit{term}\ \textit{exp}' \\
\textit{exp}' &\rightarrow \textit{addop}\ \textit{term}\ \textit{exp}' \mid \epsilon \\
\textit{addop} &\rightarrow + \mid - \\
\textit{term} &\rightarrow \textit{factor}\ \textit{term}' \\
\textit{term}' &\rightarrow \textit{mulop}\ \textit{factor}\ \textit{term}' \mid \epsilon \\
\textit{mulop} &\rightarrow * \\
\textit{factor} &\rightarrow \mathbf{(}\ \textit{exp}\ \mathbf{)} \mid \mathbf{number}
\end{aligned}
\tag{5}
$$



factors and terms

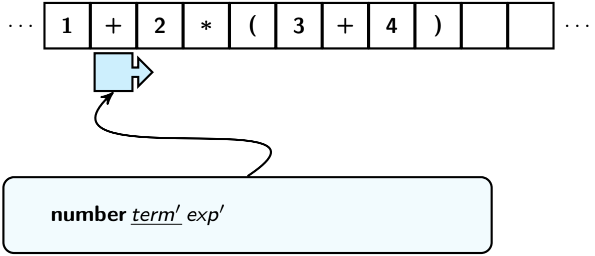
$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow (\ exp\ ) \mid \mathbf{number}
\end{aligned} \tag{5}
$$



factors and terms

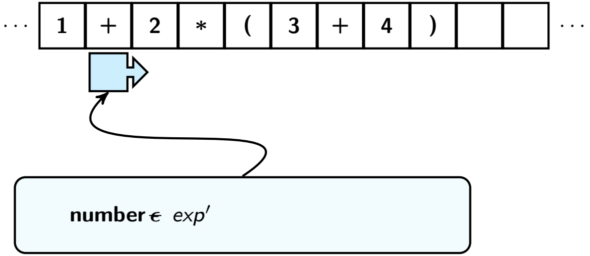
$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow \mathbf{(}\ \mathit{exp}\ \mathbf{)} \mid \mathbf{number}
\end{aligned}
\tag{5}
$$

Derivation of an expression



factors and terms

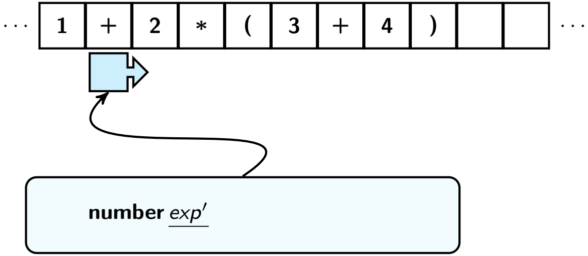
$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}
\end{aligned}
\tag{5}
$$



factors and terms

$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow (\ exp\ ) \mid \mathbf{number}
\end{aligned} \tag{5}
$$

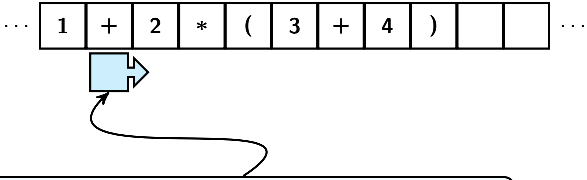
Derivation of an expression



factors and terms

$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}
\end{aligned}
\tag{5}
$$

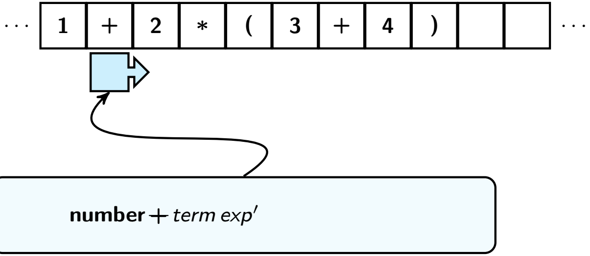
Derivation of an expression



number <u>*addop*</u> *term exp′*

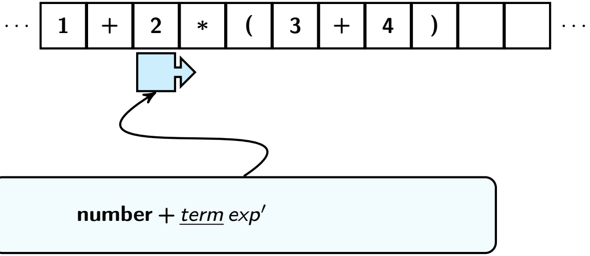
factors and terms

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\ \mathit{exp}\ ) \mid \textbf{number}
\end{aligned} \tag{5}
$$



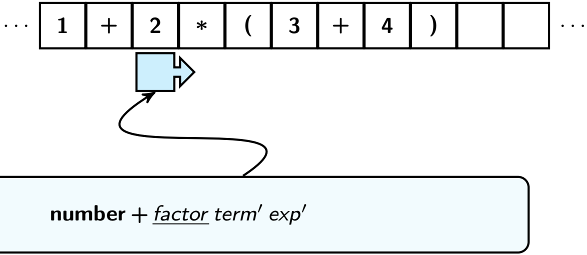
factors and terms

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



factors and terms

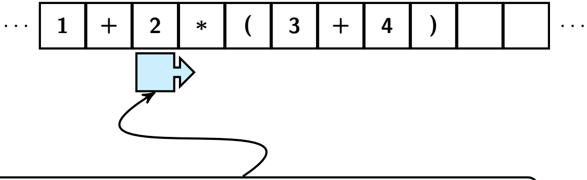
$$
\begin{array}{rcl}
exp & \rightarrow & term\ exp' \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & factor\ term' \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\
mulop & \rightarrow & * \\
factor & \rightarrow & \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}
\end{array}
\tag{5}
$$



factors and terms

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

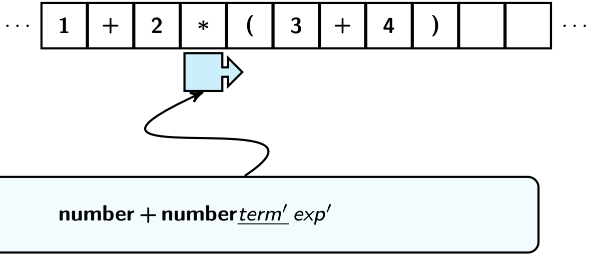
Derivation of an expression



$\textbf{number} + \cancel{\textbf{number}}\ \textit{term}'\ \textit{exp}'$

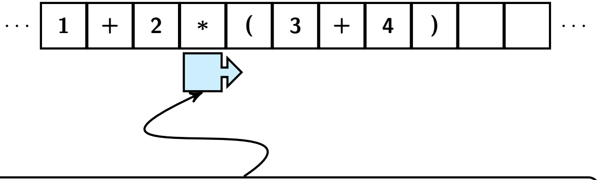
factors and terms

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



factors and terms

$$
\begin{aligned}
\textit{exp} &\rightarrow \textit{term}\ \textit{exp}' \\
\textit{exp}' &\rightarrow \textit{addop}\ \textit{term}\ \textit{exp}' \mid \epsilon \\
\textit{addop} &\rightarrow + \mid - \\
\textit{term} &\rightarrow \textit{factor}\ \textit{term}' \\
\textit{term}' &\rightarrow \textit{mulop}\ \textit{factor}\ \textit{term}' \mid \epsilon \\
\textit{mulop} &\rightarrow * \\
\textit{factor} &\rightarrow (\ \textit{exp}\ ) \mid \textbf{number}
\end{aligned}
\tag{5}
$$

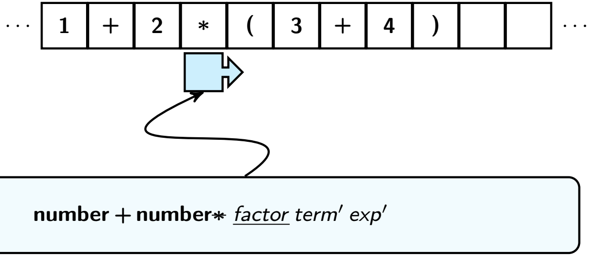


number + **number** $\underline{\textit{mulop}}$ *factor term′ exp′*

factors and terms

$$
\begin{array}{rcll}
\textit{exp} & \rightarrow & \textit{term exp}' & \\
\textit{exp}' & \rightarrow & \textit{addop term exp}' \mid \epsilon & \\
\textit{addop} & \rightarrow & + \mid - & \\
\textit{term} & \rightarrow & \textit{factor term}' & \\
\textit{term}' & \rightarrow & \textit{mulop factor term}' \mid \epsilon & \\
\textit{mulop} & \rightarrow & * & \\
\textit{factor} & \rightarrow & (\, \textit{exp} \,) \mid \textbf{number} &
\end{array}
\qquad (5)
$$

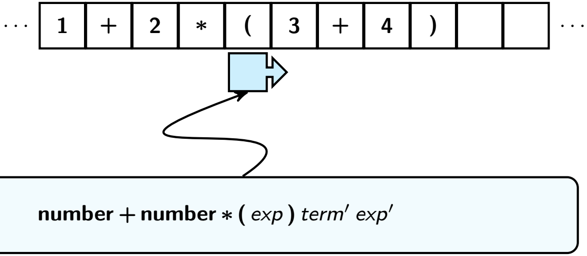
Derivation of an expression



factors and terms

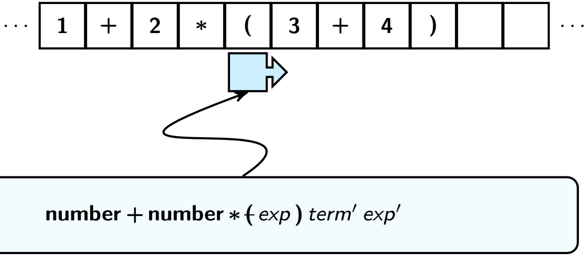
$$
\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \rightarrow & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} & \rightarrow & + \mid - \\
\mathit{term} & \rightarrow & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \rightarrow & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} & \rightarrow & * \\
\mathit{factor} & \rightarrow & (\ \mathit{exp}\ ) \mid \mathbf{number}
\end{array}
\tag{5}
$$

Derivation of an expression



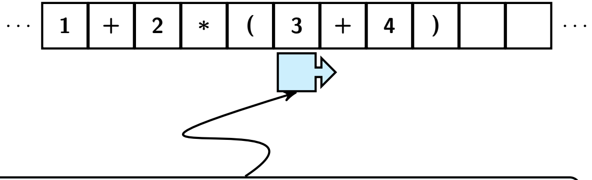
factors and terms

$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}
\end{aligned}
\tag{5}
$$



factors and terms

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

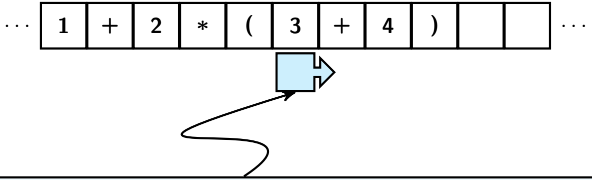


$$\textbf{number} + \textbf{number} * \textbf{(}\ \underline{\mathit{exp}}\ \textbf{)}\ \mathit{term}'\ \mathit{exp}'$$

factors and terms

$$
\begin{array}{rcl}
\mathit{exp} & \to & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \to & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} & \to & + \mid - \\
\mathit{term} & \to & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \to & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} & \to & * \\
\mathit{factor} & \to & \textbf{(}\ \mathit{exp}\ \textbf{)} \mid \textbf{number}
\end{array}
\tag{5}
$$

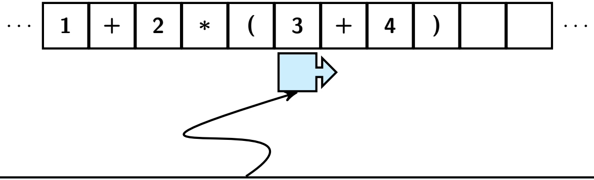
Derivation of an expression



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

factors and terms

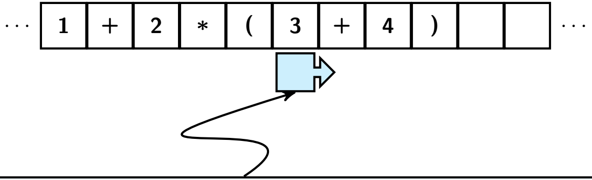
$$
\begin{array}{rcl}
exp & \rightarrow & term\ exp' \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & factor\ term' \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\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



number $+$ **number** $*$ **(** <u>*factor*</u> *term′* *exp′* **)** *term′* *exp′*

factors and terms

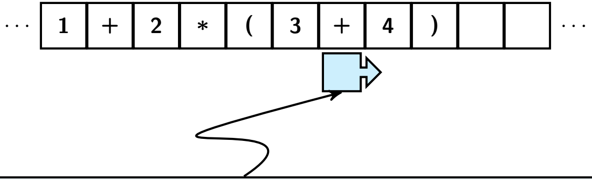
$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow \textbf{(}\ \mathit{exp}\ \textbf{)} \mid \textbf{number}
\end{aligned}
\tag{5}
$$



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

factors and terms

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\, \mathit{exp}\,) \mid \textbf{number}
\end{aligned}
\tag{5}
$$

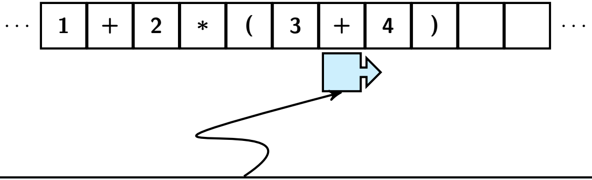


$\mathbf{number} + \mathbf{number} * (\, \mathbf{number}\, \underline{\mathit{term}'}\ \mathit{exp}'\,)\ \mathit{term}'\ \mathit{exp}'$

factors and terms

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\, \mathit{exp}\,) \mid \mathbf{number}
\end{aligned} \tag{5}
$$

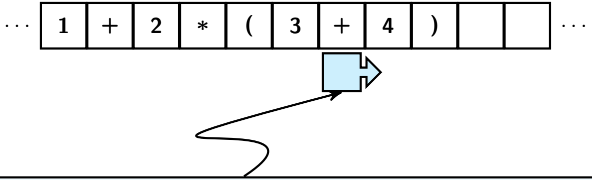
Derivation of an expression



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

factors and terms

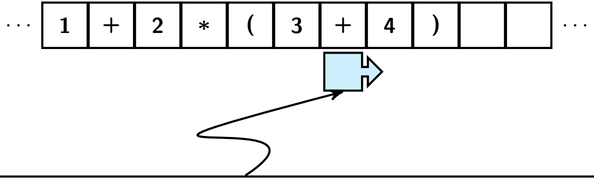
$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\ \mathit{exp}\ ) \mid \textbf{number}
\end{aligned}
\tag{5}
$$



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

factors and terms

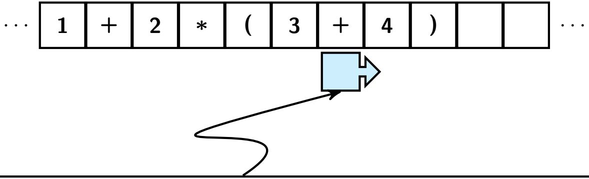
$$
\begin{array}{rcl}
exp & \rightarrow & term\ exp' \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & factor\ term' \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\, exp\,)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



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

factors and terms

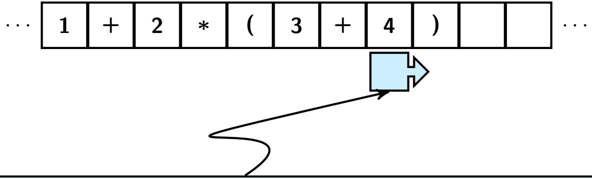
$$\begin{array}{rcl}
\textit{exp} & \rightarrow & \textit{term}\ \textit{exp}' \\
\textit{exp}' & \rightarrow & \textit{addop}\ \textit{term}\ \textit{exp}' \mid \epsilon \\
\textit{addop} & \rightarrow & + \mid - \\
\textit{term} & \rightarrow & \textit{factor}\ \textit{term}' \\
\textit{term}' & \rightarrow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



number + **number** $*$ (**number** + *term exp′*) *term′ exp′*

factors and terms

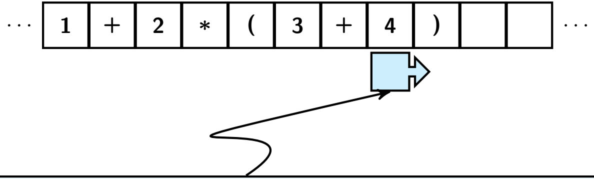
$$
\begin{array}{rcl}
exp & \rightarrow & term\ exp' \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & factor\ term' \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
\tag{5}
$$



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

factors and terms

$$
\begin{aligned}
\textit{exp} &\rightarrow \textit{term}\ \textit{exp}' \\
\textit{exp}' &\rightarrow \textit{addop}\ \textit{term}\ \textit{exp}' \mid \epsilon \\
\textit{addop} &\rightarrow + \mid - \\
\textit{term} &\rightarrow \textit{factor}\ \textit{term}' \\
\textit{term}' &\rightarrow \textit{mulop}\ \textit{factor}\ \textit{term}' \mid \epsilon \\
\textit{mulop} &\rightarrow * \\
\textit{factor} &\rightarrow (\ \textit{exp}\ ) \mid \textbf{number}
\end{aligned}
\tag{5}
$$

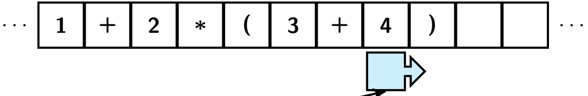


$$\textbf{number} + \textbf{number} * \textbf{(}\ \textbf{number} + \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'\ \textbf{)}\ \mathit{term}'\ \mathit{exp}'$$

factors and terms

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

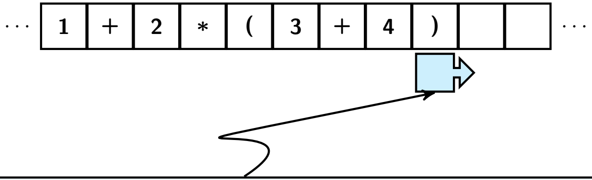
Derivation of an expression



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

factors and terms

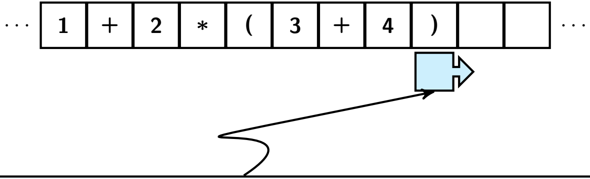
$$
\begin{aligned}
\mathit{exp} &\to \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\to \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\to + \mid - \\
\mathit{term} &\to \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\to \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\to * \\
\mathit{factor} &\to (\ \mathit{exp}\ ) \mid \textbf{number}
\end{aligned}
\tag{5}
$$



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

factors and terms

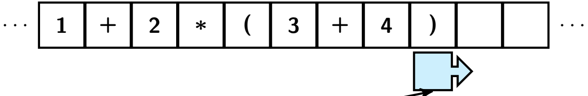
$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\,\mathit{exp}\,) \mid \textbf{number}
\end{aligned}
\tag{5}
$$



number + **number** $*$ (**number** + **number**ϵ *exp'*) *term'* *exp'*

factors and terms

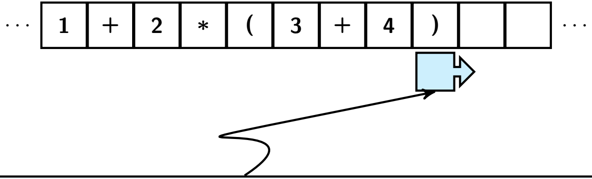
$$
\begin{array}{rcl}
exp & \rightarrow & term\ exp' \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & factor\ term' \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\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



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

factors and terms

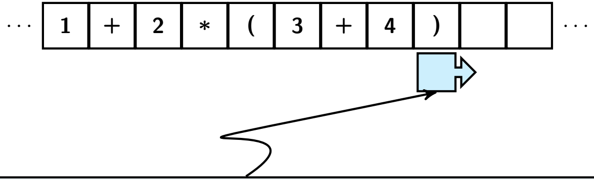
$$
\begin{aligned}
exp &\rightarrow term\ exp' \\
exp' &\rightarrow addop\ term\ exp' \mid \epsilon \\
addop &\rightarrow + \mid - \\
term &\rightarrow factor\ term' \\
term' &\rightarrow mulop\ factor\ term' \mid \epsilon \\
mulop &\rightarrow * \\
factor &\rightarrow (\, exp \,) \mid \mathbf{number}
\end{aligned}
\tag{5}
$$



$$\textbf{number} + \textbf{number} * (\, \textbf{number} + \textbf{number}\cancel{\epsilon}\,)\ \textit{term}'\ \textit{exp}'$$

factors and terms

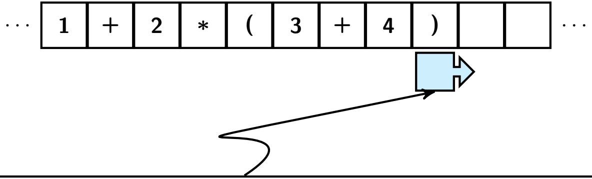
$$
\begin{array}{rcl}
\textit{exp} & \rightarrow & \textit{term}\ \textit{exp}' \\
\textit{exp}' & \rightarrow & \textit{addop}\ \textit{term}\ \textit{exp}' \mid \epsilon \\
\textit{addop} & \rightarrow & + \mid - \\
\textit{term} & \rightarrow & \textit{factor}\ \textit{term}' \\
\textit{term}' & \rightarrow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



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

factors and terms

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

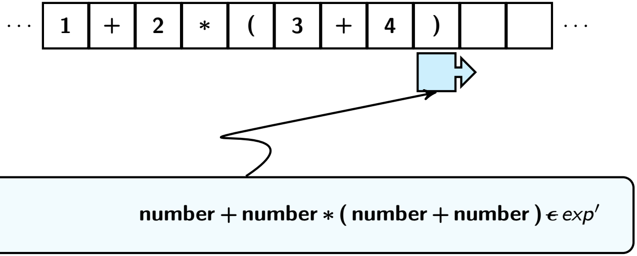


$$\mathbf{number} + \mathbf{number} * (\, \mathbf{number} + \mathbf{number} \,) \; \underline{\mathit{term}'} \; \mathit{exp}'$$

factors and terms

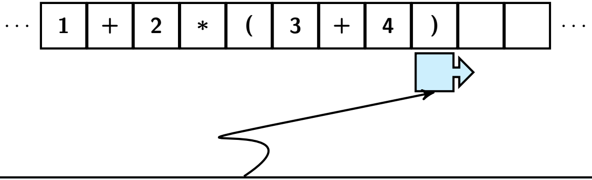
$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \;\mid\; \epsilon \\
\mathit{addop} &\rightarrow + \;\mid\; - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \;\mid\; \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\, \mathit{exp} \,) \;\mid\; \mathbf{number}
\end{aligned}
\tag{5}
$$

Derivation of an expression



factors and terms

$$
\begin{array}{rcl}
exp & \rightarrow & term\ exp' \\
exp' & \rightarrow & addop\ term\ exp' \mid \epsilon \\
addop & \rightarrow & + \mid - \\
term & \rightarrow & factor\ term' \\
term' & \rightarrow & mulop\ factor\ term' \mid \epsilon \\
mulop & \rightarrow & * \\
factor & \rightarrow & (\ exp\ ) \mid \mathbf{number}
\end{array}
\tag{5}
$$

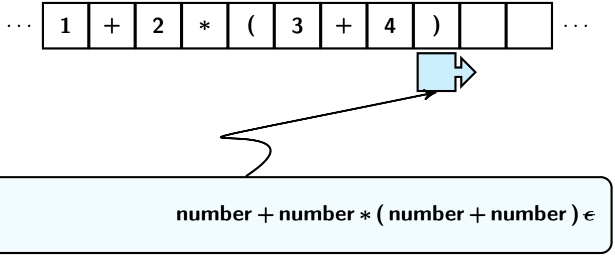


$$\textbf{number} + \textbf{number} * (\textbf{number} + \textbf{number}) \, \underline{exp'}$$

factors and terms

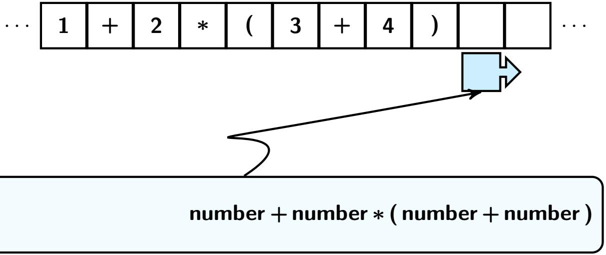
$$
\begin{array}{rcl}
exp & \to & term \; exp' \\
exp' & \to & addop \; term \; exp' \mid \epsilon \\
addop & \to & + \mid - \\
term & \to & factor \; term' \\
term' & \to & mulop \; factor \; term' \mid \epsilon \\
mulop & \to & * \\
factor & \to & (\, exp \,) \mid \textbf{number}
\end{array}
\tag{5}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

Derivation of an expression



factors and terms

$$
\begin{aligned}
\mathit{exp} &\rightarrow \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' &\rightarrow \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} &\rightarrow + \mid - \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{term}' \\
\mathit{term}' &\rightarrow \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} &\rightarrow * \\
\mathit{factor} &\rightarrow (\ \mathit{exp}\ ) \mid \textbf{number}
\end{aligned} \tag{5}
$$



factors and terms

$$
\begin{array}{rcl}
\mathit{exp} & \to & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \to & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} & \to & + \mid - \\
\mathit{term} & \to & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \to & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} & \to & * \\
\mathit{factor} & \to & \mathbf{(}\ \mathit{exp}\ \mathbf{)} \mid \mathbf{number}
\end{array}
\tag{5}
$$

Note:

- input = stream of tokens
- there: **1** . . . stands for token class **number** (for readability/concreteness), in the grammar: just **number**
- in full detail: pair of token class and token value $\langle \textbf{number}, 5\rangle$

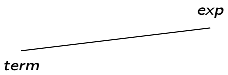
Notation:

- 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

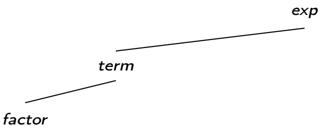
$$
\begin{array}{ll}
\underline{\mathit{exp}} & \Rightarrow \\
\underline{\mathit{term}}\ \mathit{exp}' & \Rightarrow \\
\underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ \underline{\mathit{term}'}\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

Best viewed as a tree

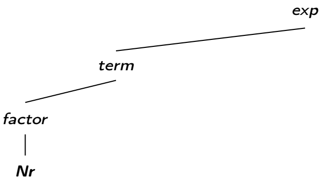
exp


exp
term

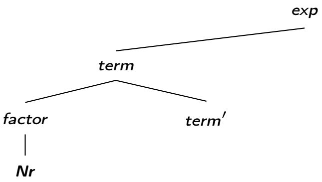
Best viewed as a tree



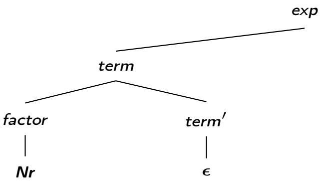
Best viewed as a tree



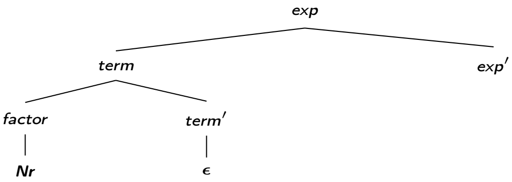
Best viewed as a tree



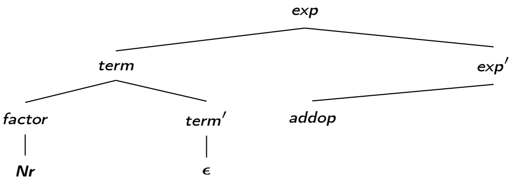
Best viewed as a tree



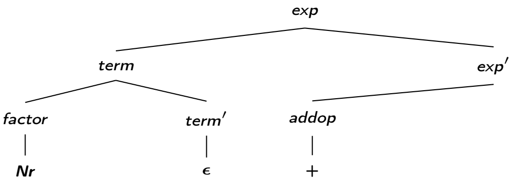
Best viewed as a tree



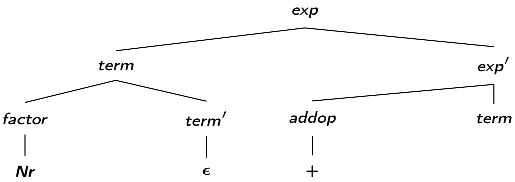
Best viewed as a tree



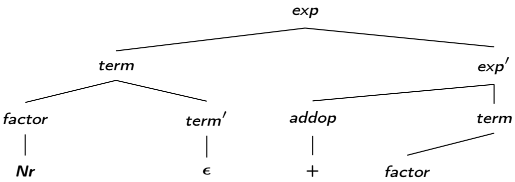
Best viewed as a tree



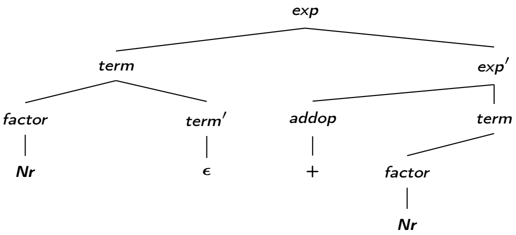
Best viewed as a tree



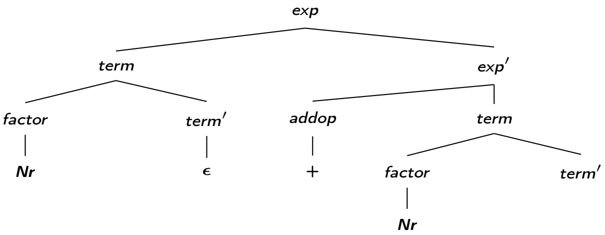
Best viewed as a tree



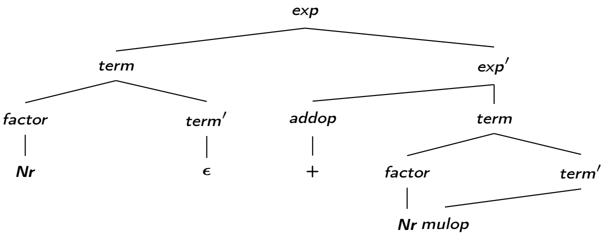
Best viewed as a tree



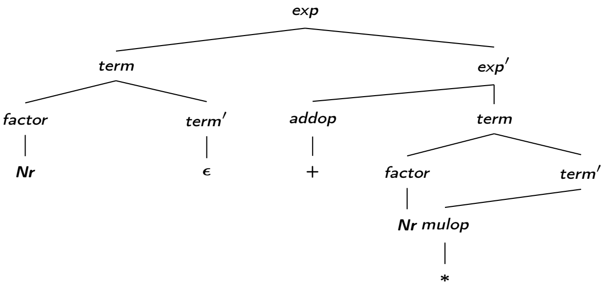
Best viewed as a tree



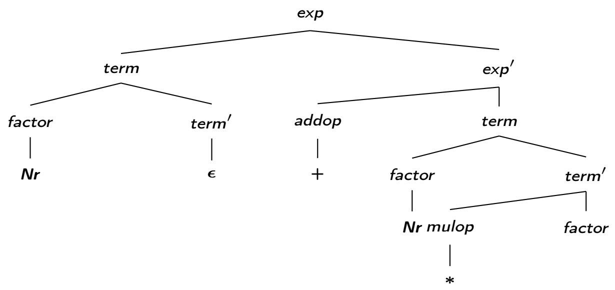
Best viewed as a tree



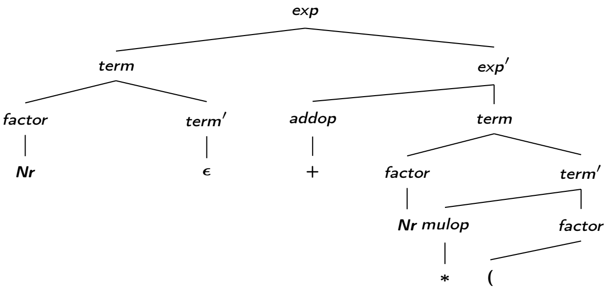
Best viewed as a tree



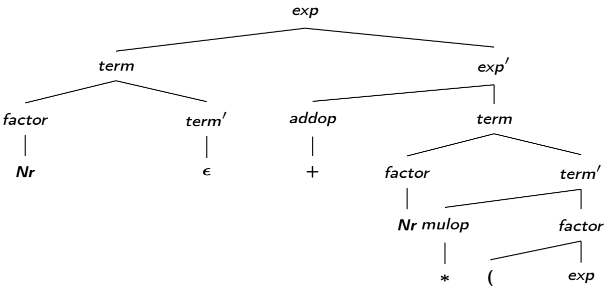
Best viewed as a tree



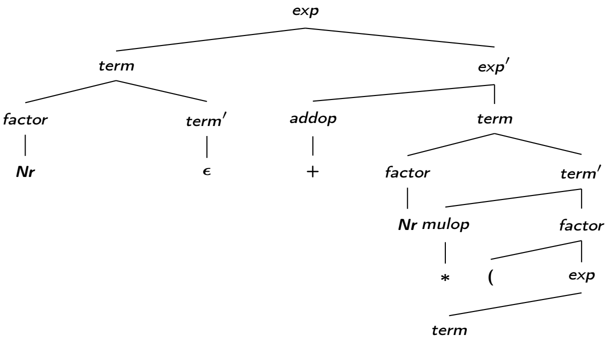
Best viewed as a tree



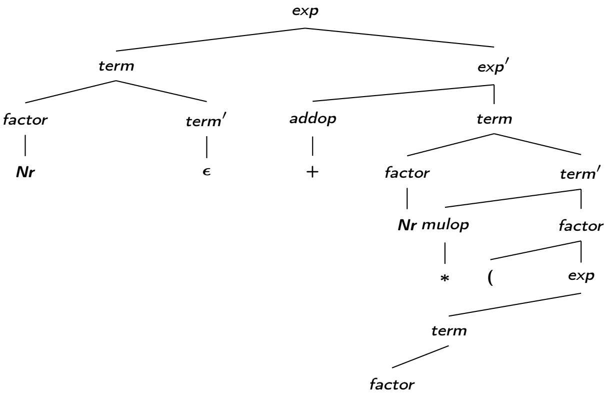
Best viewed as a tree



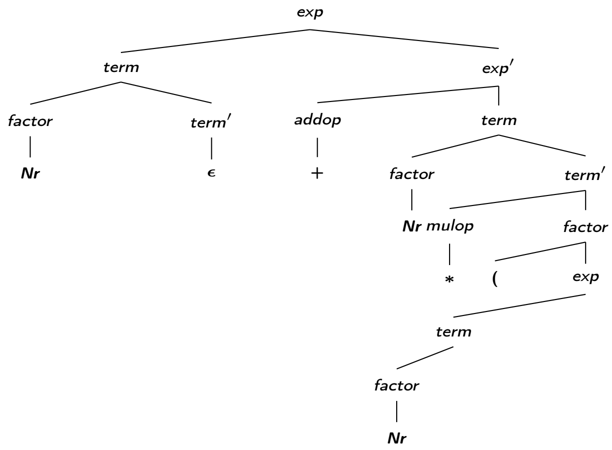
Best viewed as a tree



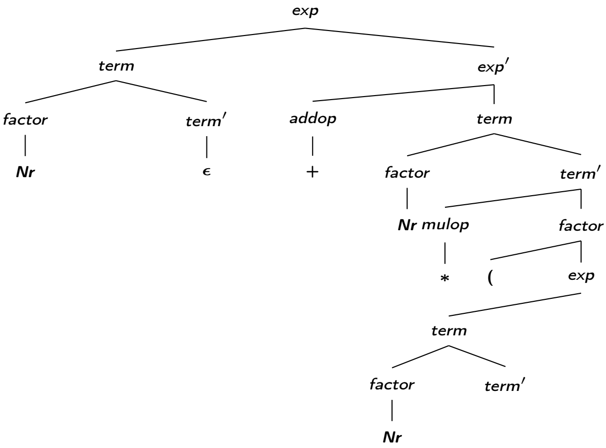
Best viewed as a tree



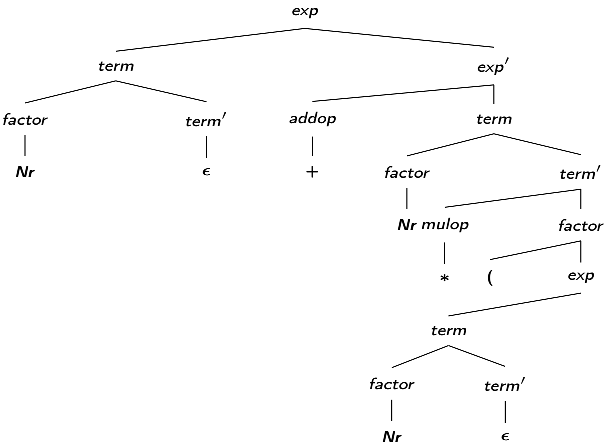
Best viewed as a tree



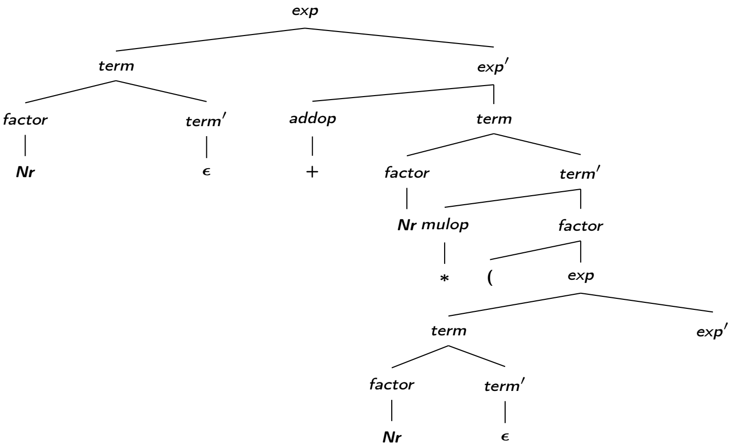
Best viewed as a tree



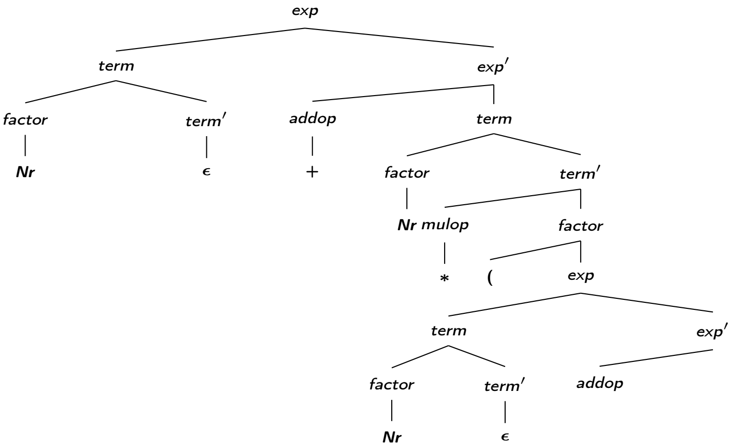
Best viewed as a tree

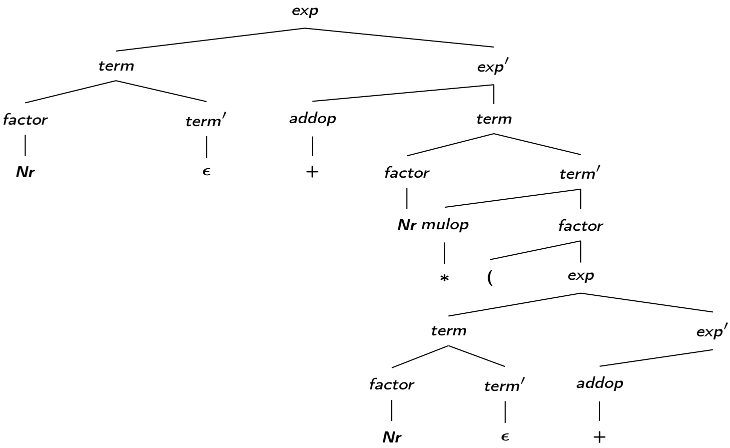


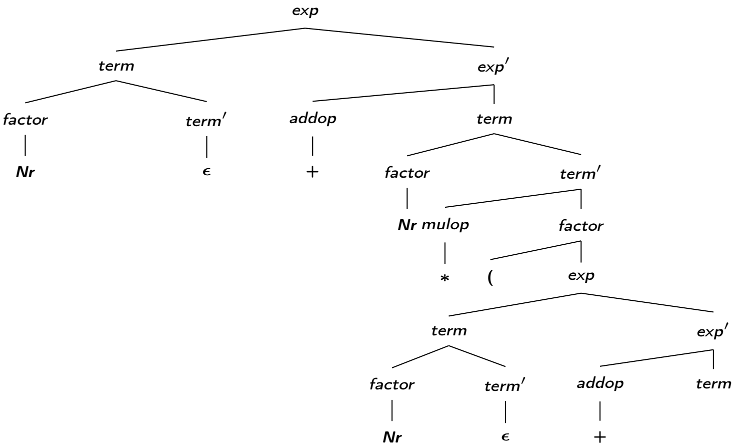
Best viewed as a tree

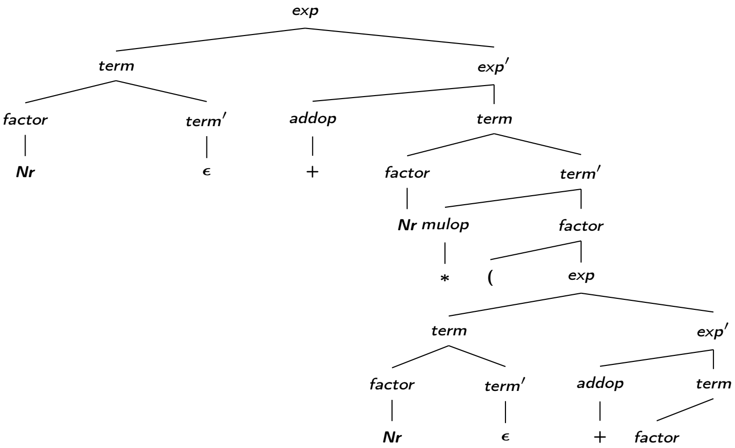


Best viewed as a tree

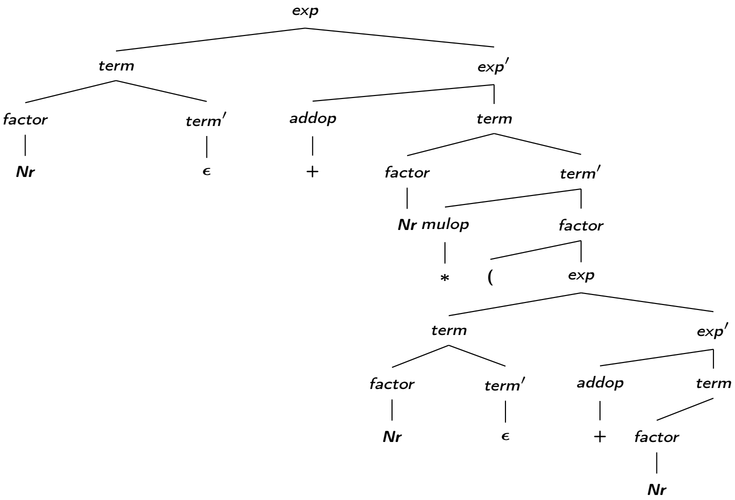



exp
term
exp′
factor
term′
addop
term
Nr
ϵ
+
factor
term′
Nr mulop
factor
*
(
exp
term
exp′
factor
term′
addop
Nr
ϵ
+

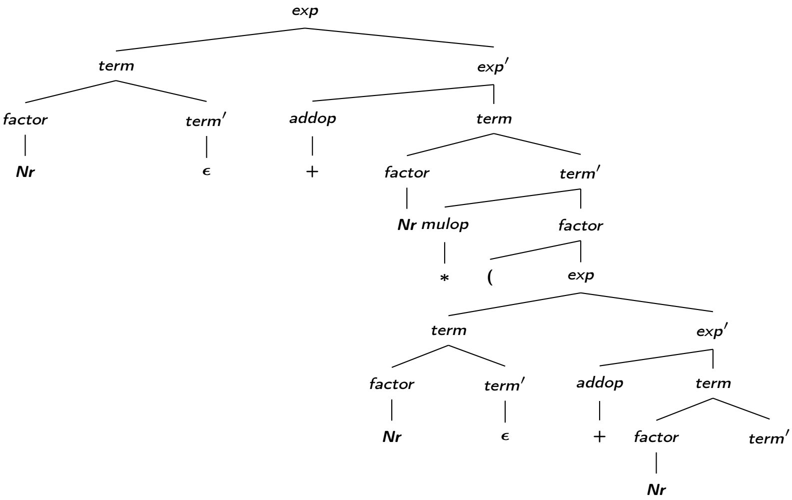

exp
term
exp′
factor
term′
addop
term
Nr
ϵ
+
factor
term′
Nr
mulop
factor
*
(
exp
term
exp′
factor
term′
addop
term
Nr
ϵ
+


exp
term
exp′
factor
term′
addop
term
Nr
ϵ
+
factor
term′
Nr
mulop
factor
*
(
exp
term
exp′
factor
term′
addop
term
Nr
ϵ
+
factor

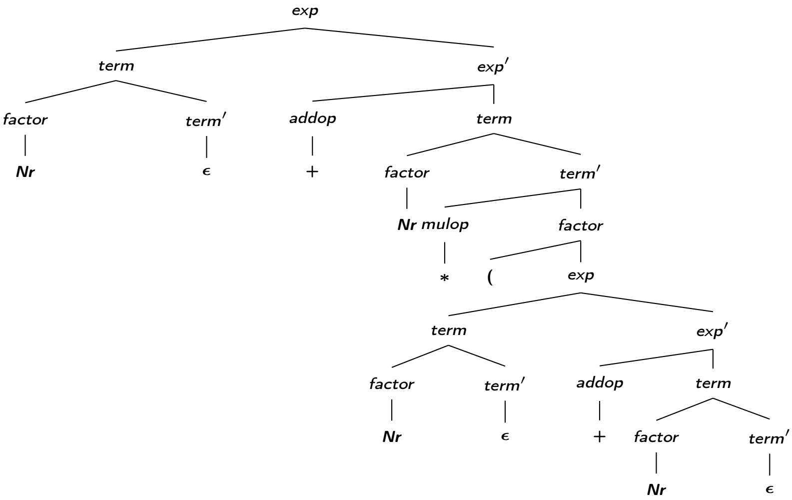
Best viewed as a tree



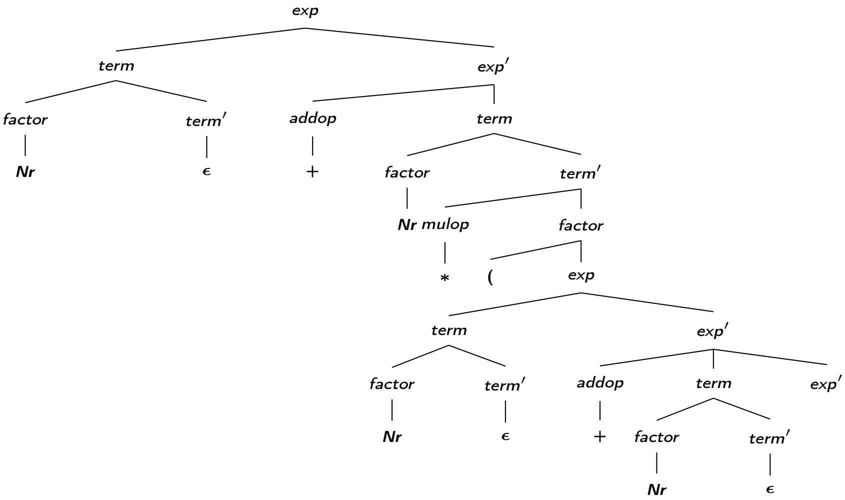
Best viewed as a tree



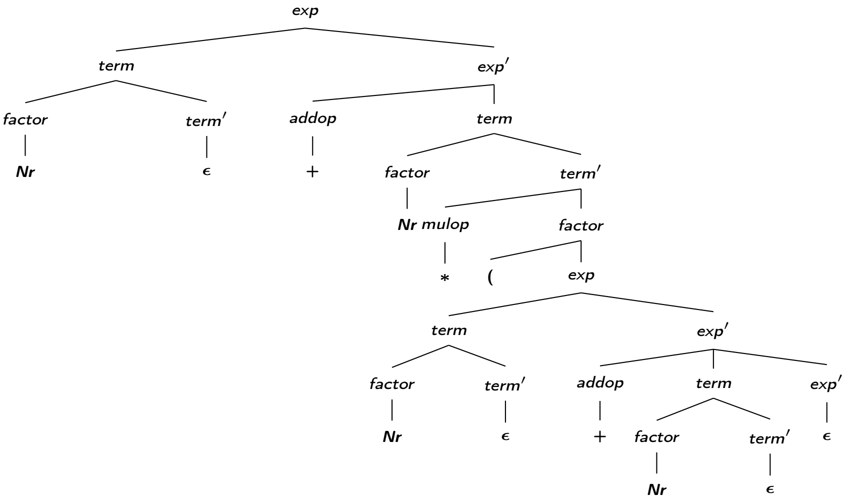
Best viewed as a tree



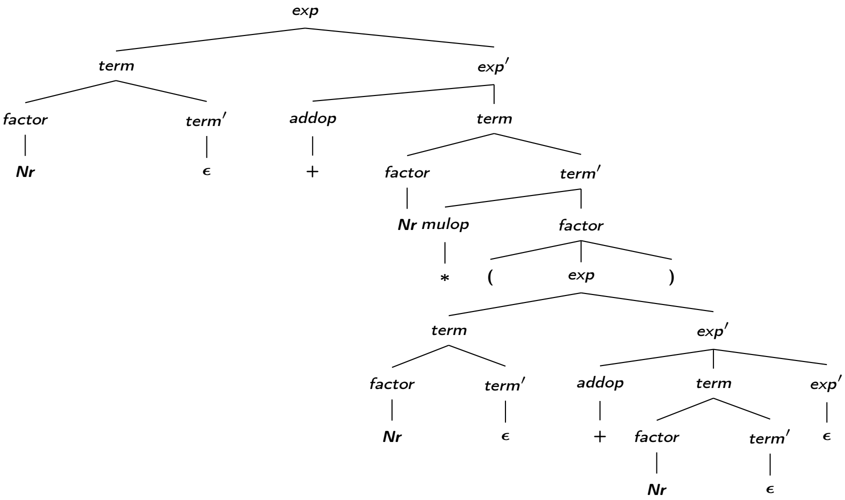
Best viewed as a tree



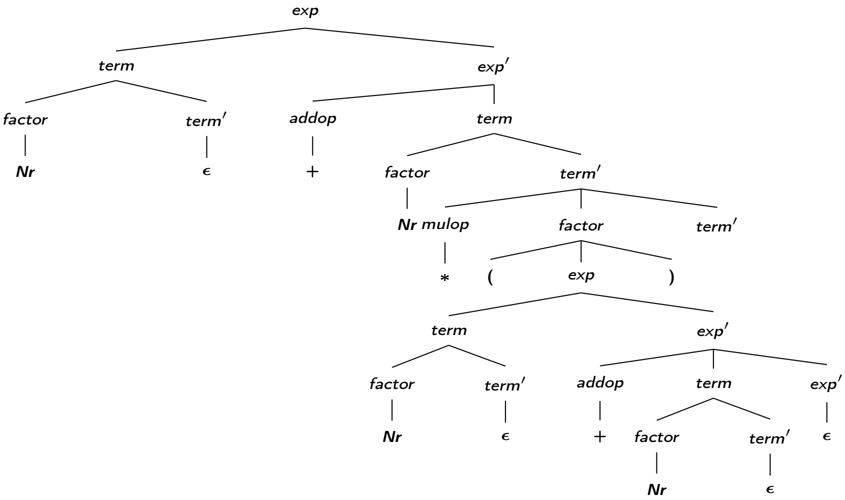
Best viewed as a tree



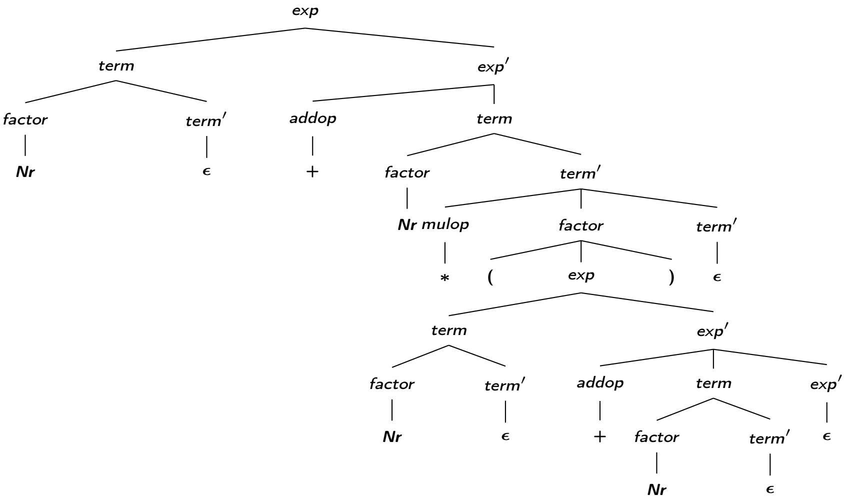
Best viewed as a tree



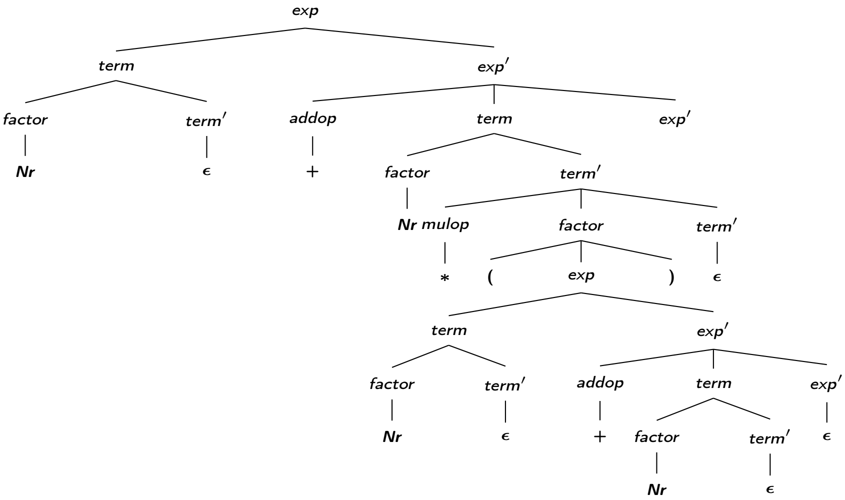
Best viewed as a tree



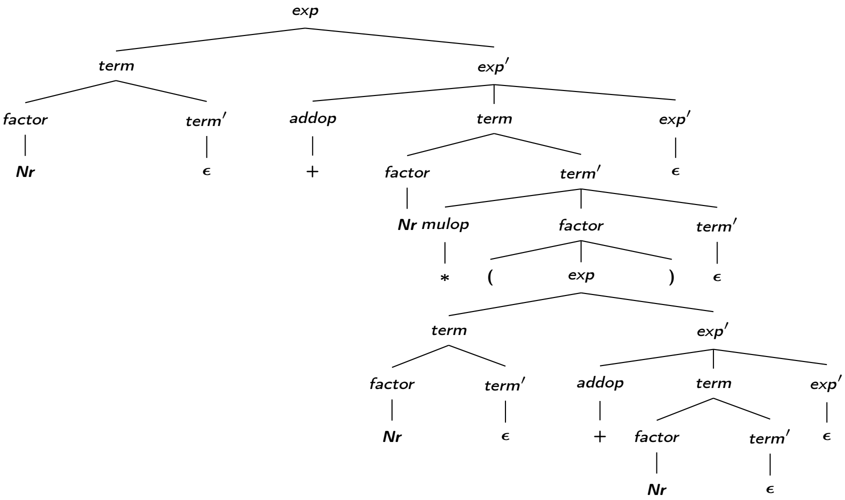
Best viewed as a tree



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?

Two principle sources of non-determinism here

Using production $A \to \beta$

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

- $\alpha_1, \alpha_2, \beta$: 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[a]
2. which production to apply (for the chosen non-terminal).

[a] Note that α_1 and α_2 may contain non-terminals, including further occurrences of A

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

- Note: the "where-to-reduce"-non-determinism $\neq$ ambiguitiy of a grammar[3]
- in a way ("theoretically"): where to reduce next is *irrelevant*:
  - the order in the sequence of derivations *does not matter*
  - what does matter: the derivation tree (aka the parse tree)

Lemma (left or right, who cares)

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

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

Using production $A \rightarrow \beta$

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

[3] A CFG is ambiguous, if there exist a word (of terminals) with 2 different parse trees.

Non-determinism vs. ambiguity

- Note: the “where-to-reduce”-non-determinism $\neq$ ambiguitiy of a grammar[3]
- in a way (“theoretically”): where to reduce next is *irrelevant*:
  - the order in the sequence of derivations *does not matter*
  - what does matter: the derivation tree (aka the parse tree)

Lemma (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 $A \rightarrow \beta$

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

[3] A CFG is ambiguous, if there exist a word (of terminals) with 2 different parse trees.

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, the target w_2 must *determine* which of the two rules to take!

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

- 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.[a]

[a] of course, one can always write a parser that "just makes some decision" based on looking ahead k symbols. The question is: will that allow to capture *all* words from the grammar and *only* those.

- in *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 looking 3) 1 symbol ahead.

- two flavors for LL(1) parsing here (both are top-down parsers)
  - *recursive descent*[4]
  - *table-based* LL(1) parser
- *predictive* parsers

[4] If one wants to be very precise: it's recursive descent with one look-ahead and without back-tracking. It's the single most common case for recursive descent parsers. Longer look-aheads are possible, but less common. Technically, even allowing back-tracking can be done using recursive descent as principle (even if not done in practice).

Sample expr grammar again

factors and terms

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

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

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

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

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

$\Rightarrow$ read the next token if there is, and decide based on the token *or else* the fact that there's none left[5]

Example: 2 productions for non-terminal *factor*

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

that situation is *trivial*, but that's not all to LL(1) ...

[5] sometimes "special terminal" \$ used to mark the end

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

Idea

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

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

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

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);
    }
}
```

```
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)
```

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[a]

[a] It must be the next token/terminal in the sense of *First*, but it need not be a token *directly* mentioned on the right-hand sides of the corresponding rules.

$\Rightarrow$ 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* $\alpha_1 \neq \alpha_2$*:*

$$\mathit{First}_1(\alpha_1) \cap \mathit{First}_1(\alpha_2) = \emptyset \ .$$

- often: common *left factors* problematic

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

- requires a look-ahead of (at least) 2
- ⇒ try to rearrange the grammar
  1. *Extended* BNF ([Louden, 1997] suggests that)

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

  1. *left-factoring*:

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

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

factors and terms

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

- consider treatment of *exp*: *First*(*exp*)?
  - whatever is in *First*(*term*), is in *First*(*exp*)[6]
  - even if only *one* (left-recursive) production $\Rightarrow$ *infinite* recursion.

Left-recursion

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

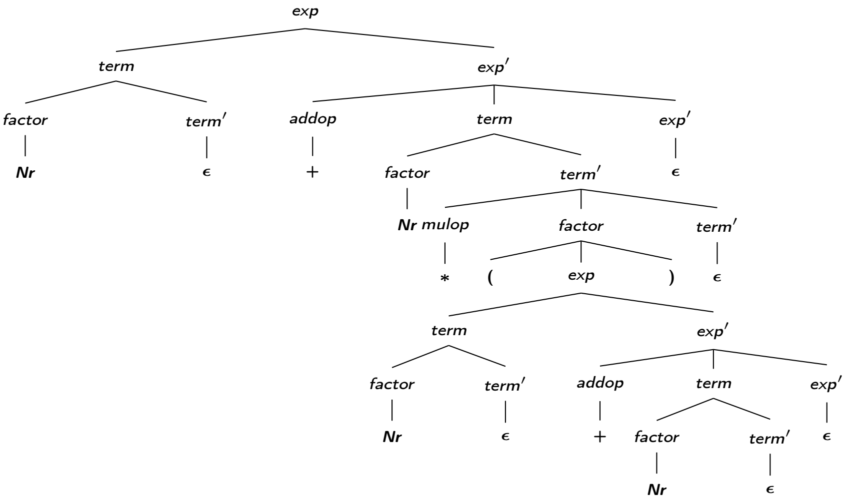
[6]And it would not help to *look-ahead* more than 1 token either.

Removing left recursion may help

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

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

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



. . . who wants this form of trees?

The two expression grammars again

Precedence & assoc.

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

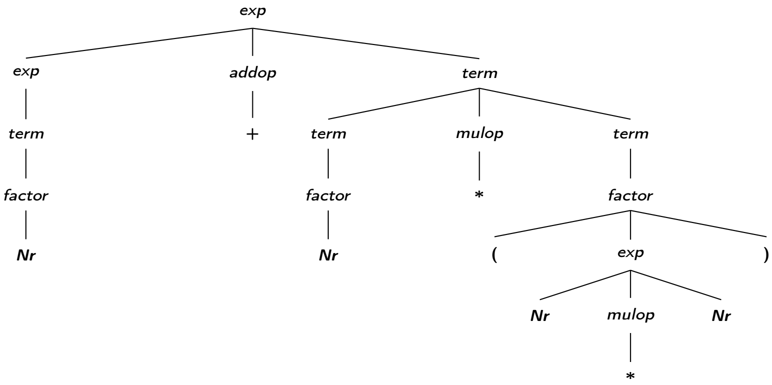
- clean and straightforward rules
- left-recursive

no left-rec.

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

- no left-recursion
- assoc. / precedence ok
- rec. descent parsing ok
- but: just "unnatural"
- non-straightforward parse-trees

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

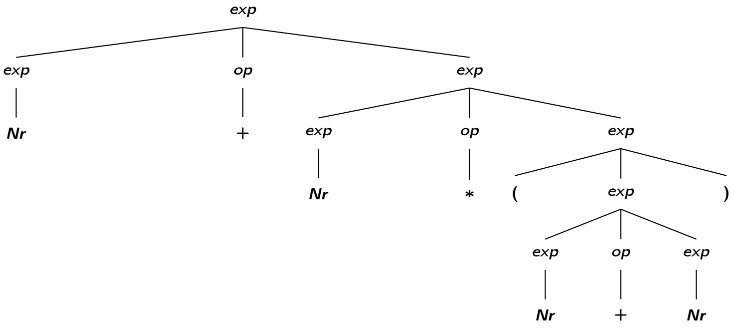

exp
exp
addop
term
term
+
term
mulop
term
factor
factor
*
factor
Nr
Nr
(
exp
)
Nr
mulop
Nr
*

The simple "original" expression grammar (even nicer)

Flat expression grammar

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

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



Associtivity problematic

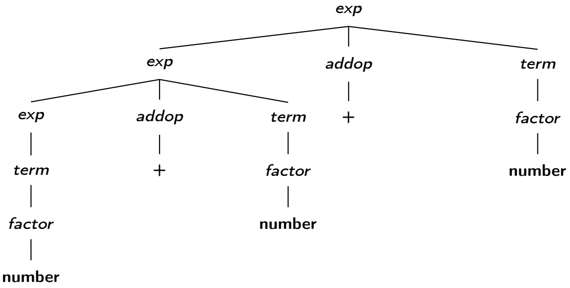
Precedence & assoc.

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

$3 + 4 + 5$

parsed “as”

$(3 + 4) + 5$



Associtivity problematic

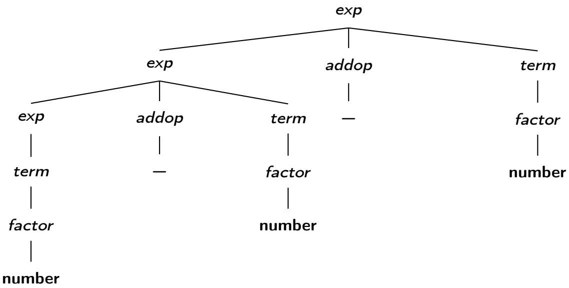
Precedence & assoc.

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

$3 - 4 - 5$

parsed "as"

$(3 - 4) - 5$

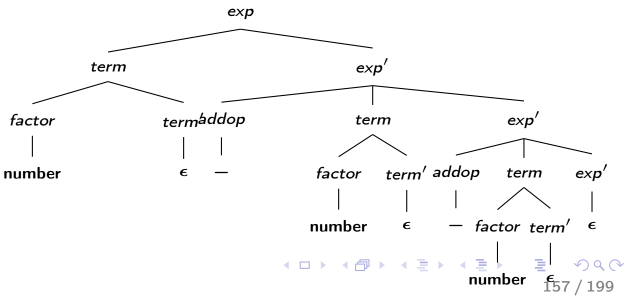


Now use the grammar without left-rec (but right-rec instead)

No left-rec.

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

$3 - 4 - 5$



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

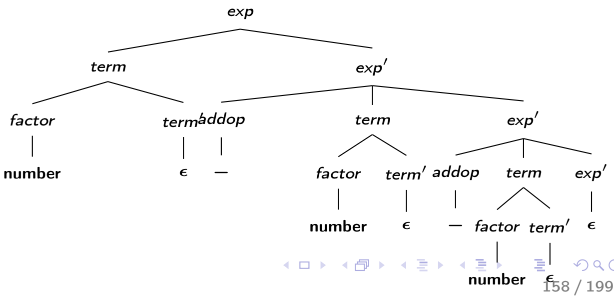
No left-rec.

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

$3 - 4 - 5$

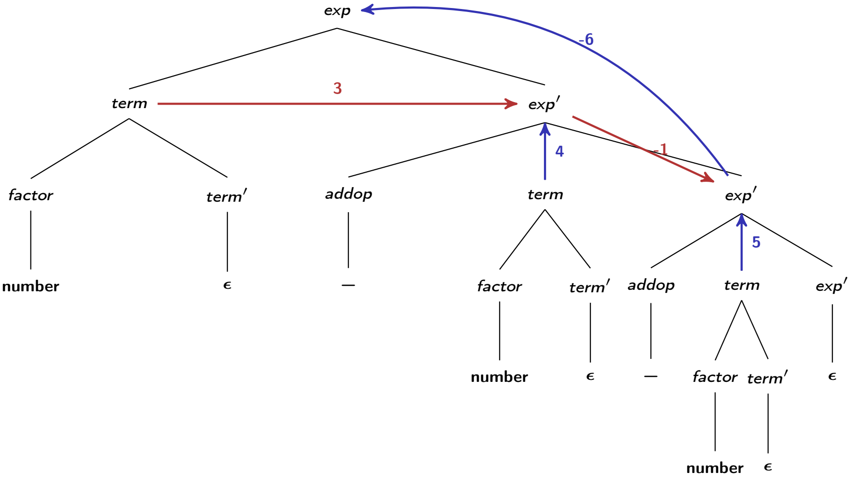
parsed "as"

$3 - (4 - 5)$



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

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



Code to "evaluate" ill-associated such trees correctly

```
function exp' ( valsofar : integer ) : integer ;
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 exp' ;
```

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

- many trade offs:
  1. starting from: design of the language, how much of the syntax is left "implicit"[7]
  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

[7]Lisp is famous/notorious in that its surface syntax is more or less an explicit notation for the ASTs. Not that it was originally planned like this . . .

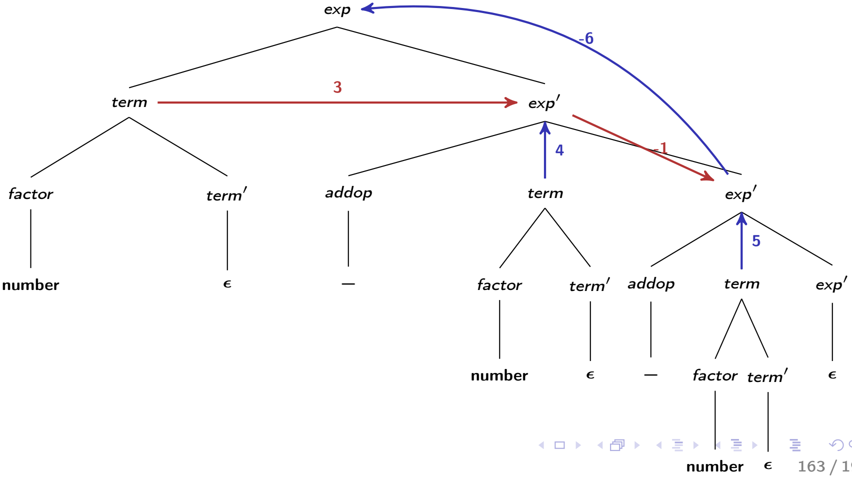
- once steps 1.–3. are fixed: *parse-trees* fixed!
- parse-trees = *essence* of a grammatical derivation process
- often: parse trees only "conceptually" present in a parser
- AST:
  - *abstractions* of the parse trees
  - *essence* of the parse tree
  - actual tree data structure, as output of the parser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

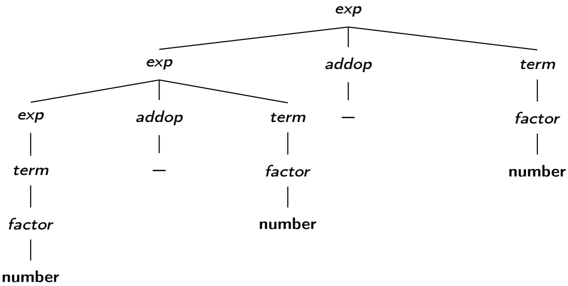
The parser "builds" the AST data structure while "doing" the parse tree.

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",

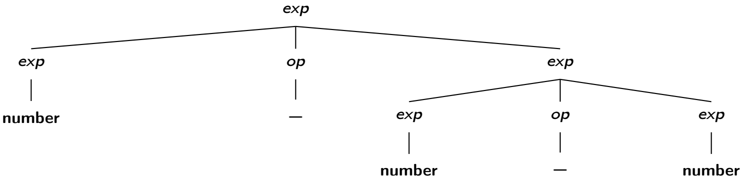


- AST: only thing relevant for later phases ⇒ better be *clean* ...
- AST "=" CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed "weirdly",



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

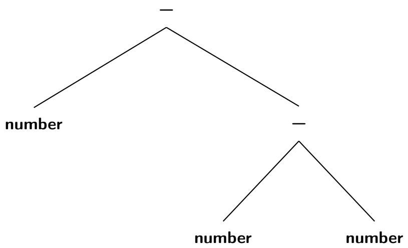
- AST: only thing relevant for later phases ⇒ better be *clean* . . .
- AST "=" CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed "weirdly",



That parse tree looks reasonable clear and intuitive

AST: How “far away” from the CST?

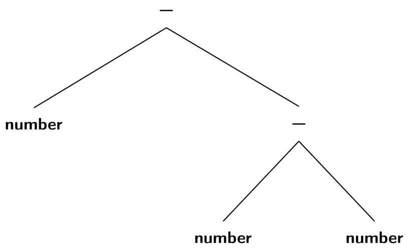
- AST: only thing relevant for later phases ⇒ better be *clean* . . .
- AST “=” CST?
  - building AST becomes straightforward
  - possible choice, if the grammar is not designed “weirdly”,



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”,

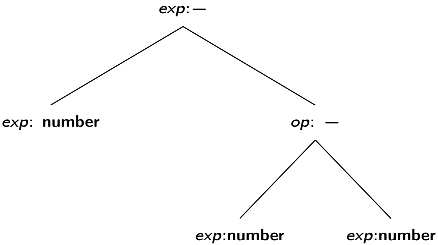


Wouldn’t that be the best AST here?

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!*

Assume, one has a "non-weird" grammar

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

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

Remember (independent from parsing)

BNF describe trees

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

Recipe

- turn each non-terminal to an abstract class
- turn each right-hand side of a given non-terminal as (non-abstract) 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*

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

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

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

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

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

Example in Java

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

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

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

Pragmatic deviations from the recipe

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

$\Rightarrow$ that class is *not* needed

- some might prefer an implementation of

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

as simply integers, for instance arranged like

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

and used as `BinExpr.PLUS` etc.

Recipe for ASTs, final words:

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

Do it systematically

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

- a word on [Louden, 1997] His C-based representation of the AST is a bit on the "bit-squeezing" side of things . . .

Extended BNF may help alleviate the pain

BNF

$$
\begin{array}{rcl}
exp & \to & exp\ addop\ term \mid term \\
term & \to & term\ mulop\ factor \mid factor
\end{array}
$$

EBNF

$$
\begin{array}{rcl}
exp & \to & term\{\ addop\ term\ \} \\
term & \to & factor\{\ mulop\ factor\ \}
\end{array}
$$

but remember:

- EBNF just a notation, just because we do not see (left or right) recursion in $\{\ \ldots\ \}$, does not mean that there is no recursion.
- not all parser generators support EBNF
- however: often easy to translate into loop [8]
- does not offer a *general* solution if associativity etc is problematic

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

Pseudo-code representing the EBNF productions

```
procedure exp;
begin
  term ;          { recursive call }
  while token = "+" or token = "-"
  do
     match(token);
     term;        // recursive call
  end
end
```

```
procedure term;
begin
  factor;         { recursive call }
  while token = "*"
  do
     match(token);
     factor;      // recursive call
  end
end
```

How to produce "something" during RD parsing?

Recursive descent

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

[a] 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.

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

Evaluating an *exp* during RD parsing

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

Building an AST: expression

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

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

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

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

Building an AST: conditionals

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

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

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

- 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 2 dimensional array)[9]

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

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

[9] Often, the entry in the parse table does not contain a full rule as here, needed is only the *right-hand-side*. In that case the table is of type $\Sigma_N \times \Sigma_T \to (\Sigma^* + \mathtt{error})$. We follow the convention of this book

Table recipe

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

Table recipe (again)

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

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

- grammars is left-factored and not left recursive

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

	First	*Follow*
stmt	**other**, **if**	\$, **else**
if-stmt	**if**	\$, **else**
else_part	**else**, ϵ	\$, **else**
exp	**0**, **1**	**)**

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

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

LL(1) table based algo

```
while the top of the parsing stack ≠ $
   if the top of the parsing stack is terminal a
       and the next input token = a
   then
       pop the parsing stack;
       advance the input;
   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 → X_1 X_2 ... X_n
        then (* generate *)
             pop the parsing stack
             for i := n to 1 do
             push X onto the stack;
      else error
   if   the top of the stack = $
   then accept
end
```

LL(1): illustration of run of the algo

Table 4.3

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

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

Original grammar

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

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

Original grammar

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

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

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

Left-rec removed

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

	First	*Follow*
exp	(, **number**	\$,)
exp'	+, −, ϵ	\$,)
addop	+, −	(, **number**
term	(, **number**	\$,), +, −
term'	∗, ϵ	\$,), +, −
mulop	∗	(, **number**
factor	(, **number**	\$,), +, −, ∗

Expressions: LL(1) parse table

M[N, T]	(	**number**	)	+	-	*	$
exp	$exp \to term\ exp'$	$exp \to term\ exp'$					
exp'			$exp' \to \varepsilon$	$exp' \to addop\ term\ exp'$	$exp' \to addop\ term\ exp'$		$exp' \to \varepsilon$
addop				$addop \to$ **+**	$addop \to$ **-**		
term	$term \to factor\ term'$	$term \to factor\ term'$					
term'			$term' \to \varepsilon$	$term' \to \varepsilon$	$term' \to \varepsilon$	$term' \to mulop\ factor\ term'$	$term' \to \varepsilon$
mulop						$mulop \to$ *****	
factor	$factor \to$ **(** exp **)**	$factor \to$ **number**					

- 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

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

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

	Synch-set (stack or parameter):
<program>	$
<u>begin</u> <deklsekv> ; <setningssekv> <u>end</u>	<u>end</u>
<stmnt> ; <stmnt> ; <stmnt> ;	; First(stmt)
	<u>name</u> <u>if</u> <u>while</u> <u>for</u> ...
<u>if</u> <expr> <u>then</u> <stmnt> [<u>else</u> <stmnt>]	<u>then</u> First(stmt) <u>else</u>
<term> ± <term> ± <term>	+ - First(term)
	(<u>integer</u> <u>name</u>
<factor> * <factor> * <factor> ...	* First(factor)
(<expr>)	)
<term> ± <term> ± <term>	+ - (<u>tall</u> <u>navn</u>

From the sketch at the previous page we can easily find:

- Which call should continue the execution?
- What input symbol should this method search for before resuming?
- We assume that $ is added to the synch. stack only by the outermost method (for the start symbol)
- The union of everything on the stack is called the "synch. set", SS

The algorithm for this goes is as follows:

For each coming input symbol, test if it is a member of SS

If so:

- Look through the SS stack from newest to oldest, and find the newest method
    - that are willing to resume at one of these symbol
- This method will itself know how to resume after the actual input symbol

What is *not* easy is to program this without destroing the nich program structure occuring from pure recursive descent.

2

Procedures for expression with "error recovery"

```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match (token) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number }) ;
  end if;
end exp ;
```

?

Also { + , - } ?

if token in {(,number} then ...

Main philosophy

The method "checkinput" is called twice: First to check that the construction starts correctly, and secondly to check that the symbol after the construction is legal.

Uses parameters, not a stack

The procedures must themselves resume execution at the right place inside themselves when they get the control back,

or it must terminate immediately if it cannot resume execution on the current symbol.

```
procedure factor ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    case token of
    ( : match( ( ) ;
        exp ( { ) } ) ;
        match( ) ) ;
    number :
        match(number) ;
    else error ;
    end case ;
    checkinput ( synchset, { (, number }) ;
  end if ;
end factor ;
```

Why not the full"synchset"?

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { $ }) do
    getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset ∪ followset ) ;
  end if ;
end;
```

27

References I

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

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