INF5110 – Compiler Construction

Parsing

Spring 2016

- 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

First and Follow sets

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

First-set of A

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

Follow-set of A

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

- sentential form: word *derived from* grammar's starting symbol
- later: different algos for First and Follow sets, for 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 $A \Rightarrow^* \epsilon$.

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

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

- in many languages:

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

- for statements:

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

Remarks

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

$$X \to X_1 X_2 \ldots X_n$$

   2.1 $First(X)$ contains $First(X_1) \setminus \{\epsilon\}$
   2.2 If, for some $i < n$, all $First(X_1), \ldots, First(X_i)$ contain ϵ, then $First(X)$ contains $First(X_i) \setminus \{\epsilon\}$.
   2.3 If all $First(X_1), \ldots, First(X_n)$ 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 $First(\alpha)$ is defined inductively as follows:

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

```
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
```

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

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 $A \to \epsilon$.

Example expression grammar (from before)

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

Example expression grammar (expanded)

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

Run of the "algo"

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

- results per pass:

	1	2	3
exp			$\{(, \textbf{\textit{number}}\}$
addop	$\{+, -\}$		
term		$\{(, \textbf{\textit{number}}\}$	
mulop	$\{*\}$		
factor	$\{(, \textbf{\textit{number}}\}$		

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

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

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 also possible

Follow sets

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

Follow sets, recursively

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

```
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
```

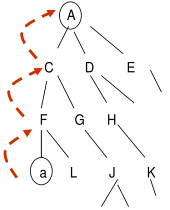
Note! $\mathit{First}() = \epsilon$

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

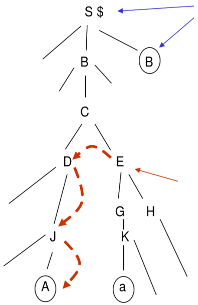
Run of the "algo"

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

$a \in First(A)$



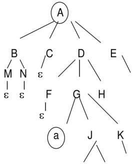
$a \in Follow(A)$



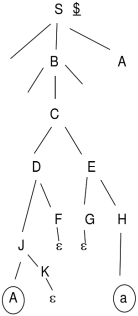
- red arrows: illustration of information flow in the algos
- run of *Follow*:
  - relies on *First*
  - in particular $a \in First(E)$ (right tree)
- $\$ \in Follow(B)$

More complex situation (nullability)

$a \in First(A)$



$a \in Follow(A)$



- left-recursive production:

$$A \rightarrow A\alpha$$

more precisely: example of *immediate* left-recursion

- 2 productions with common "left factor":

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

Some simple examples

- left-recursion

$$exp \rightarrow exp + term$$

- classical example for common left factor: rules for conditionals

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

Transforming the expression grammar

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

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

After removing left recursion

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

- 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

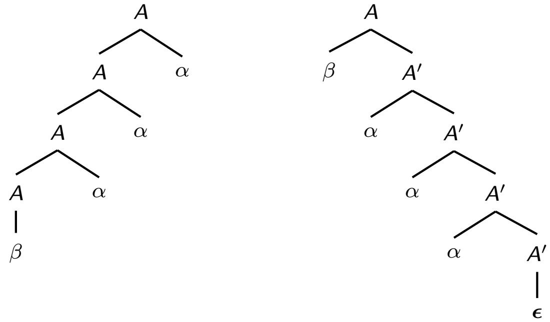
$$A \rightarrow A\alpha \mid \beta$$

After

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

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

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



- 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

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

After

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

Note, can be written in *EBNF* as:

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

```
for i := 1 to m do
   for j := 1 to i−1 do
      replace each grammar rule of the form A → 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
   remove, if necessary, immediate left recursion for A_i
end
```

Example (for the general case)

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

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

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

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

Left factor removal

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

$\Rightarrow$ common left factor undesirable

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

Simple situation

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

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

Example: sequence of statements

Before

$$
\begin{aligned}
\textit{stmt-seq} \quad &\rightarrow \quad \textit{stmt}\ \textbf{;}\ \textit{stmt-seq} \\
&\mid \quad \textit{stmt}
\end{aligned}
$$

After

$$
\begin{aligned}
\textit{stmt-seq} \quad &\rightarrow \quad \textit{stmt}\ \textit{stmt-seq}' \\
\textit{stmt-seq}' \quad &\rightarrow \quad \textbf{;}\ \textit{stmt-seq} \ \mid\ \epsilon
\end{aligned}
$$

Example: conditionals

Before

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

After

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

Example: conditionals

Before

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

After

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

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

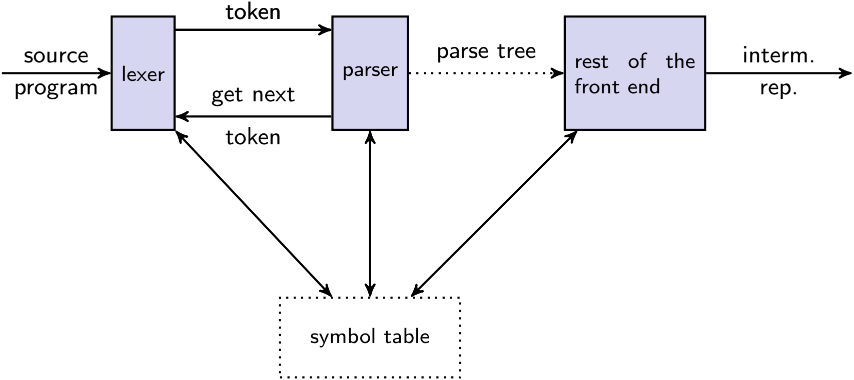
```
while  there are changes to the grammar  do
  for  each nonterminal A  do
    let  α be a prefix of max. length that is shared
           by two or more productions for A
    if    α ≠ ϵ
    then
        let  A → α1 | ... | αn be all
        prod. for A and suppose that α1, ..., αk share α
        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
```

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 ...)


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

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.

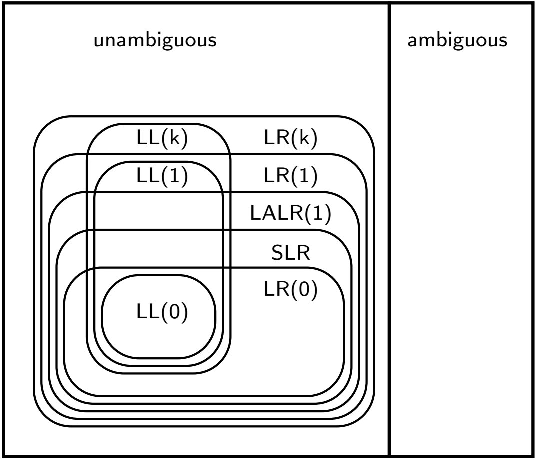
- while parse tree mostly conceptual: parsing build up the concrete data structure of AST bottom-up vs. top-down.

- 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 $\Rightarrow$ 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

Relationship of some classes

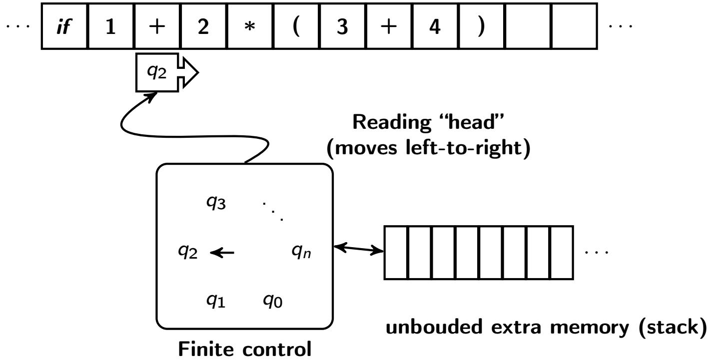


taken from [Appel, 1998]

General task (once more)

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

Schematic view on "parser machine"



Note: sequence of *tokens* (not characters)

Derivation of an expression

$\underline{exp}$ $\quad$ $\underline{term}\ exp'$ $\quad$ $\underline{factor}\ term'\ exp'$ $\quad$ $\cancel{\textbf{number}}\ term'\ exp'$ $\quad$ $\textbf{number}\ \underline{term'}\ exp'$ $\quad$ $\textbf{number}\ \cancel{\epsilon}\ exp'$ $\quad$ $\textbf{n}$

factors and terms

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

Note:

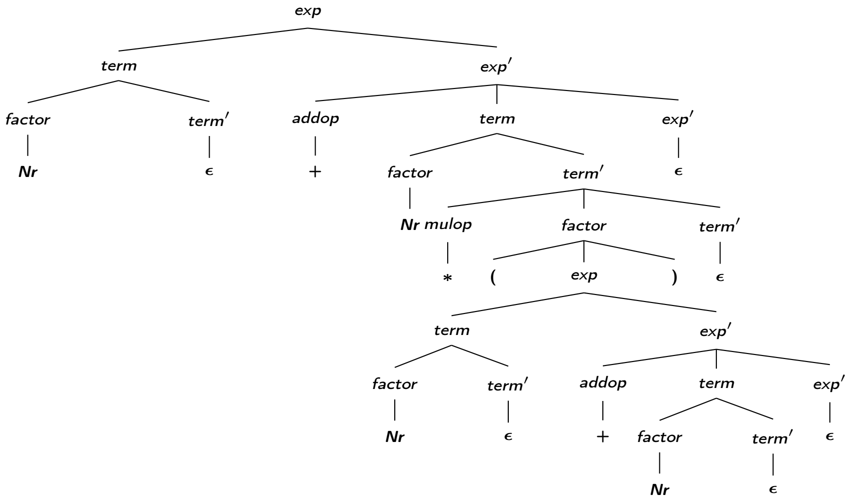
- input = stream of tokens
- there: $\mathbf{1}\ldots$ stands for token class ***number*** (for readability/concreteness), in the grammar: just ***number***
- in full detail: pair of token class and token value $\langle \textbf{\textit{number}}, 5\rangle$

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

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

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


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

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

Left-most derivation

- taking care of "where-to-reduce" non-determinism: *left-most* derivation
- notation $\Rightarrow_l$
- the example derivation used that
- second look at the "guided" derivation proccess: ?

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)

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

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

Using production $A \rightarrow \beta$

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

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

[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!

Deterministic, yes, but still impractical

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

- 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

[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

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

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

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

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

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

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

Example: 2 productions for non-terminal *factor*

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

that's *trivial*, but that's not all ...

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

Recursive descent: general set-up

- global variable, say `tok`, representing the "current token"
- 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 *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)
```

Recursive descent principle

- recursive descent: aka *predictive* parser

Princple

one *function* (method/procedure) for each non-terminal and *one case* for each production.

-

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, 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 $A \to \alpha_1$ *and* $A \to \alpha_2$ *with* $\alpha_1 \neq \alpha_2$*:*

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

- sometimes: common left factors are problematic

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

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

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

  1. *left-factoring*:

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

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

Left recursion is a no-go

factors and terms

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

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

Left-recursion

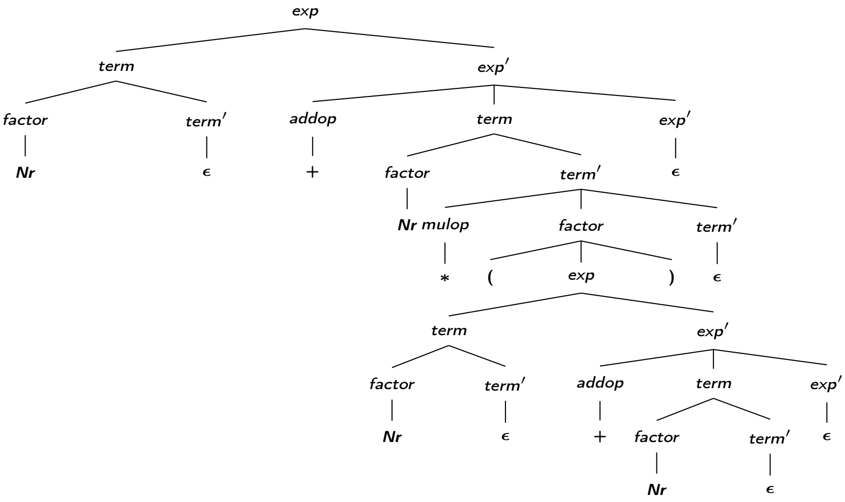
Left-recursive grammar *never* works for recursive descent.

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

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

```
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.

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

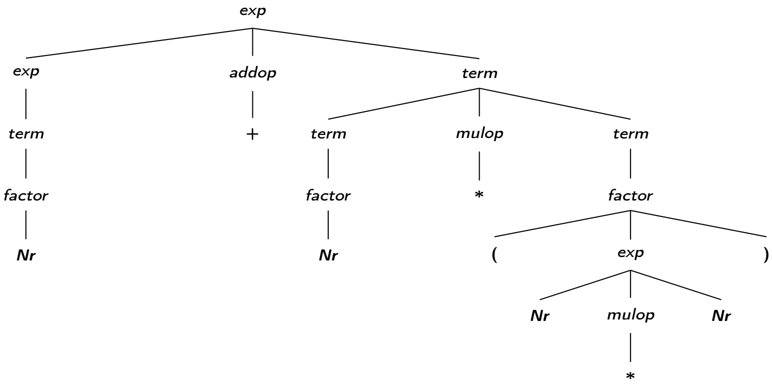
- clean and straightforward rules
- left-recursive

no left-rec.

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

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

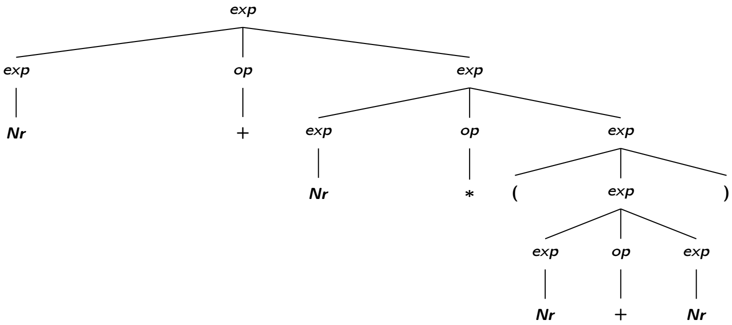
$$1 + 2 * (3 + 4)$$


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

Flat expression grammar

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

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



Associtivity problematic

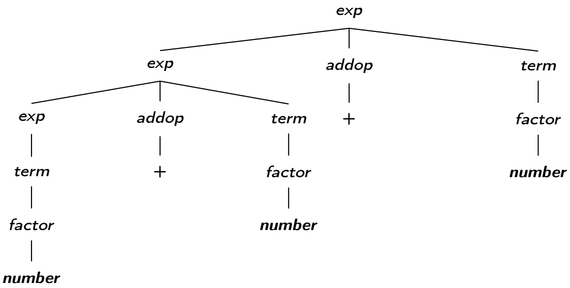
Precedence & assoc.

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

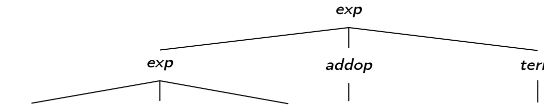
$3 + 4 + 5$

parsed "as"

$(3 + 4) + 5$



$3 - 4 - 5$



Now use the grammar without left-rec

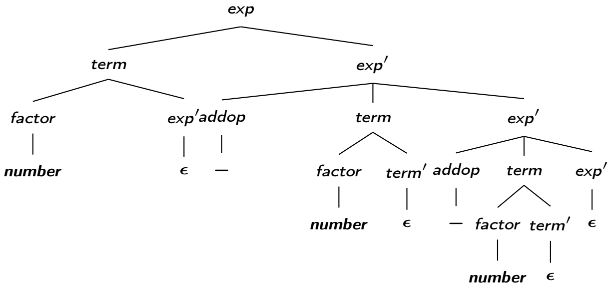
No left-rec.

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

$3 - 4 - 5$

parsed "as"

$3 - (4 - 5)$



- 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 ...

AST vs. CST

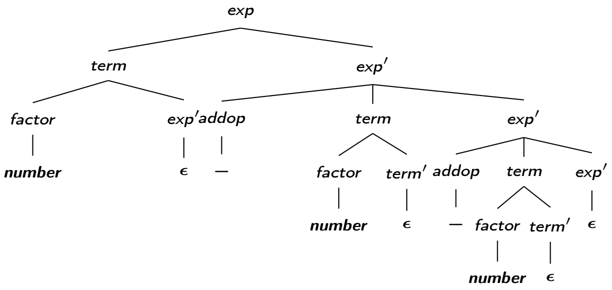
- once steps 1.–3. are fixed: *parse-trees* fixed!
- parse-trees = *essence* of a grammatical derivation process
- often: parse trees only "conceptually" present in a parser
- AST:
  - *abstractions* of the parse trees
  - *essence* of the parse tree
  - actual tree data structure, as output of the parser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

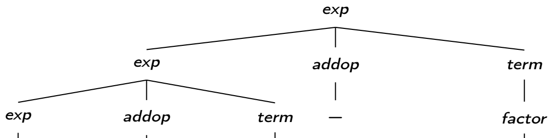
The parser "builds" the AST data structurea while "doing" the parse tree.

AST: How “far away” from the CST?

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



parse-trees like that better be cleaned up as AST



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

Assume, one has a "non-weird" grammar, like

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

- 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 a parser-generator that allows to *specify* things like " "∗" *binds stronger* than "+", it *associates* to the left ..." without cluttering the grammar.

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*

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

```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;}
}
```

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

3 − (4 − 5)

```
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

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

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 ...

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.