Syntax Analysis

Syntax Analysis

- Syntax analysis checks the correctness of a program according to the grammar:
  - gets scanner generated stream of tokes as an input;
  - outputs a syntax-tree corresponding to the program;
  - in the presence of syntactic errors, locates them:
  - ...reports possible causes of the errors;
  - ...tries to recover and continue the analysis (in order to discover more errors).
- Syntax analysis is called parsing and the corresponding analyzer parser.

Grammars

- Syntax is usually described by context-free grammars.
- Grammar is a quadruple $G = \langle N, T, P, S \rangle$, where
  - N is a finite alphabet of non-terminal symbols;
  - T is a finite alphabet of terminal symbols;
  - $N \cap T = \emptyset$ and $V = N \cup T$;
  - $P \subset \{\alpha \to \beta \mid \alpha \in V^+,\ \beta \in V^\star\}$ is a finite set of production rules;
  - $S \in N$ is a start symbol.
- Grammar is context-free if production rules are in the form $A \to \alpha$, where $A \in N$ and $\alpha \in V^\star$.

Grammars

- A sequence $w \in V^\star$ is called a sentential form.
- The sentential form $v \in V^\star$ is directly derivable from the sentential form $u \in V^\star$ (notation $u \Longrightarrow v$), if there are $w_1, w_2, \alpha, \beta \in V^\star$ such, that $u = w_1 \alpha w_2$, $v = w_1 \beta w_2$ and $\alpha \to \beta \in P$.
- Reflexive transitive closure of the relation $\Longrightarrow$ is called derivation (notation $\Longrightarrow^\star$).
- The grammar $G = \langle N, T, P, S \rangle$ generates a language

$$L(G) = \{w \in T^\star \mid S \Longrightarrow^\star w\}$$

- Grammars G_1 and G_2 are equivalent if $L(G_1) = L(G_2)$.

Grammars

Chomsky hierarchy:

	Productions	Languages	Automata
L_0	$\alpha \to \beta$	Semi-Thue systems	Turing machines
L_1	$\alpha A \beta \to \alpha \gamma \beta$	Context-dependent	Bounded TM-s
L_2	$A \to \alpha$	Context-free	Push-down automata
L_3	$A \to w,\ A \to wB$	Regular	Finite automata
(L_4)	$A \to w$	Finite	FA without cycles

where $A, B \in N,\ \alpha, \beta, \gamma \in V^\star$ and $w \in T^\star$.

Lemma: Chomsky hierarchy is strict; ie.:

$$(L_4) \subset L_3 \subset L_2 \subset L_1 \subset L_0$$

Context-free Grammars

- From now on we consider only context-free grammars.
- Production rules of context-free grammars are usually described using Backus-Naur Form (BNF).
- Example: let $N = \{\text{Exp}\}$ and $T = \{+, *, (,), id\}$, then

$$\begin{array}{rcl} \text{Exp} & \to & \text{Exp} + \text{Exp} \\ & | & \text{Exp} * \text{Exp} \\ & | & (\ \text{Exp}\ ) \\ & | & id \end{array}$$

describes the set of production rules

$$\begin{array}{rcll} P & = & \{ & \text{Exp} \to \text{Exp} + \text{Exp}, \quad \text{Exp} \to (\ \text{Exp}\ ), \\ & & & \text{Exp} \to \text{Exp} * \text{Exp}, \quad \text{Exp} \to id \qquad \}. \end{array}$$

Context-free Grammars

- Non-terminal A is productive if there exists $w \in T^\star$ such that $A \Longrightarrow^\star w$.
- Non-terminal A is reachable if there exist sentential forms $u, v \in V^\star$ such that $S \Longrightarrow^\star uAv$.
- CF-grammar $G = \langle N, T, P, S \rangle$ is reduced if every non-terminal is productive and reachable.
- Lemma: Every CF-grammar can be transformed into an equivalent reduced CF-grammar.

Context-free Grammars

- A sentential form may have several derivations.
- Canonical derivations:
  - left-derivation – on every derivation step the leftmost non-terminal is replaced;
  - right-derivation – on every derivation step the rightmost non-terminal is replaced.
- Example:

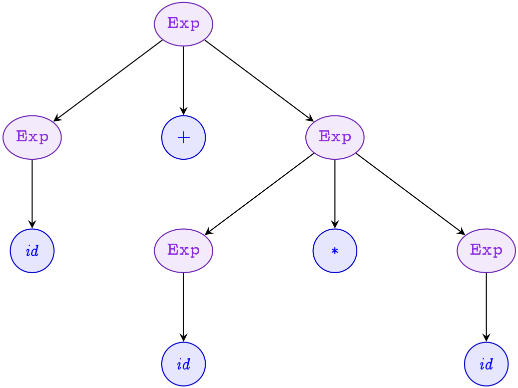
$$
\begin{array}{lll}
\text{Exp} & \Longrightarrow_{lm} & \text{Exp} + \text{Exp} \\
 & \Longrightarrow_{lm} & id + \text{Exp} \\
 & \Longrightarrow_{lm} & id + \text{Exp} * \text{Exp} \\
 & \Longrightarrow_{lm} & id + id * \text{Exp} \\
 & \Longrightarrow_{lm} & id + id * id
\end{array}
\qquad
\begin{array}{lll}
\text{Exp} & \Longrightarrow_{rm} & \text{Exp} + \text{Exp} \\
 & \Longrightarrow_{rm} & \text{Exp} + \text{Exp} * \text{Exp} \\
 & \Longrightarrow_{rm} & \text{Exp} + \text{Exp} * id \\
 & \Longrightarrow_{rm} & \text{Exp} + id * id \\
 & \Longrightarrow_{rm} & id + id * id
\end{array}
$$

Context-free Grammars

- Every derivation determines a unique syntax-tree (or parse-tree) – a tree with ordered nodes, where:
  - the root is labelled by the start symbol S;
  - intermediate nodes are labelled by non-terminals;
  - leaves are labelled by terminals or the empty symbol ε;
  - when intermediate node is labelled by the non-terminal A and roots of its subtrees (from left to right) $t_1, \dots, t_n$ are labelled by $A_1, \dots, A_n$, then $A \to A_1 \dots A_n \in P$.
- Labels of leaves (read from left to right) form the derived sentential form.
- Syntax-tree uniquely determines which production rules were used, but not the order of their application.

Context-free Grammars

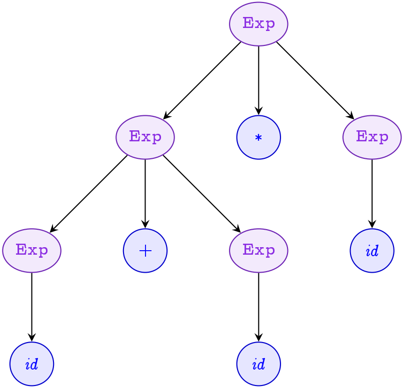
Example: previously given left- and right-derivations correspond to the same syntax-tree



Context-free Grammars

NB! A sentence may have several syntax-trees!

$$\begin{aligned}
\text{Exp} &\Longrightarrow_{lm} \text{Exp} * \text{Exp} \\
&\Longrightarrow_{lm} \text{Exp} + \text{Exp} * \text{Exp} \\
&\Longrightarrow_{lm} id + \text{Exp} * \text{Exp} \\
&\Longrightarrow_{lm} id + id * \text{Exp} \\
&\Longrightarrow_{lm} id + id * id
\end{aligned}$$



Context-free Grammars

- CF-grammar is ambiguous if for the same sentence there are several syntax-trees.
- For every syntax-tree, there is exactly one left- and right-derivation; thus:
  - non-ambiguous sentence has exactly one left- and one right-derivation;
  - ambiguous sentence has at least two left- and right-derivations..
- Different syntax-trees of a sentence usually correspond to different semantic interpretations of the sentence.
- An ambiguous grammar can sometimes (but not always) be transformed to an equivalent non-ambiguous one.

Context-free Grammars

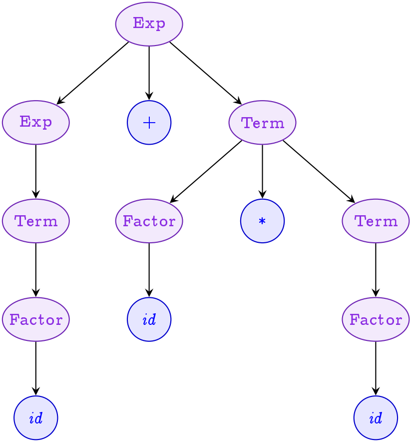
- Elimination of ambiguity – binary operators:
  - every priority level introduces a new non-terminal;
  - left-associative operators use left-recursion; right-associative operators right-recursion;
  - rules corresponding to operators of higher priorities are placed "deeper".
- Example:

$$
\begin{array}{lcl}
\text{Exp} & \rightarrow & \text{Exp} + \text{Term} \\
 & \mid & \text{Term} \\
\text{Term} & \rightarrow & \text{Factor} * \text{Term} \\
 & \mid & \text{Factor} \\
\text{Factor} & \rightarrow & (\ \text{Exp}\ ) \\
 & \mid & id
\end{array}
$$

Context-free Grammars

Example:

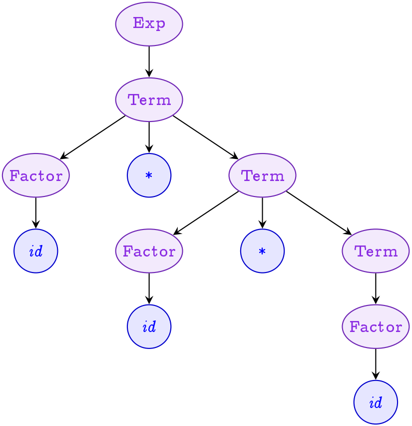
$$
\begin{aligned}
\text{Exp} &\Longrightarrow_{lm} \text{Exp} + \text{Term} \\
&\Longrightarrow_{lm} \text{Term} + \text{Term} \\
&\Longrightarrow_{lm} \text{Factor} + \text{Term} \\
&\Longrightarrow_{lm} id + \text{Term} \\
&\Longrightarrow_{lm} id + \text{Factor} * \text{Term} \\
&\Longrightarrow_{lm} id + id * \text{Term} \\
&\Longrightarrow_{lm} id + id * \text{Factor} \\
&\Longrightarrow_{lm} id + id * id
\end{aligned}
$$



Context-free Grammars

Example:

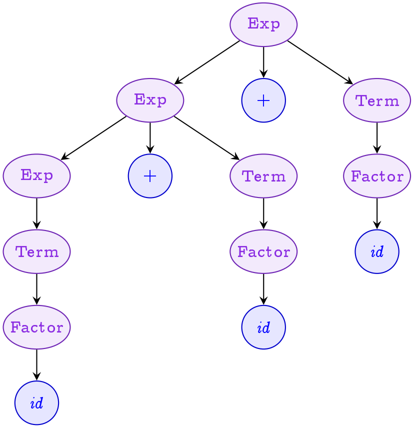
$$
\begin{aligned}
\text{Exp} &\Longrightarrow_{lm} \text{Term} \\
&\Longrightarrow_{lm} \text{Factor} * \text{Term} \\
&\Longrightarrow_{lm} id * \text{Term} \\
&\Longrightarrow_{lm} id * \text{Factor} * \text{Term} \\
&\Longrightarrow_{lm} id * id * \text{Term} \\
&\Longrightarrow_{lm} id * id * \text{Factor} \\
&\Longrightarrow_{lm} id * id * id
\end{aligned}
$$



Context-free Grammars

Example:

$$
\begin{aligned}
\text{Exp} &\Longrightarrow_{lm} \text{Exp} + \text{Term} \\
&\Longrightarrow_{lm} \text{Exp} + \text{Term} + \text{Term} \\
&\Longrightarrow_{lm} \text{Term} + \text{Term} + \text{Term} \\
&\Longrightarrow_{lm} \text{Factor} + \text{Term} + \text{Term} \\
&\Longrightarrow_{lm} id + \text{Term} + \text{Term} \\
&\Longrightarrow_{lm} id + \text{Factor} + \text{Term} \\
&\Longrightarrow_{lm} id + id + \text{Term} \\
&\Longrightarrow_{lm} id + id + \text{Factor} \\
&\Longrightarrow_{lm} id + id + id
\end{aligned}
$$



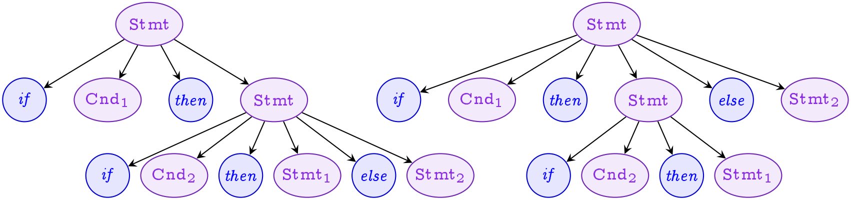
Context-free Grammars

- Elimination of ambiguity – condition statements:

$$\begin{array}{lcl} \text{Stmt} & \rightarrow & \textit{if} \ \text{Cnd} \ \textit{then} \ \text{Stmt} \\ & | & \textit{if} \ \text{Cnd} \ \textit{then} \ \text{Stmt} \ \textit{else} \ \text{Stmt} \\ & | & \text{Other} \end{array}$$

- The following sentence has two different syntax-trees:

$$\textit{if} \ \text{Cnd}_1 \ \textit{then} \ \textit{if} \ \text{Cnd}_2 \ \textit{then} \ \text{Stmt}_1 \ \textit{else} \ \text{Stmt}_2$$

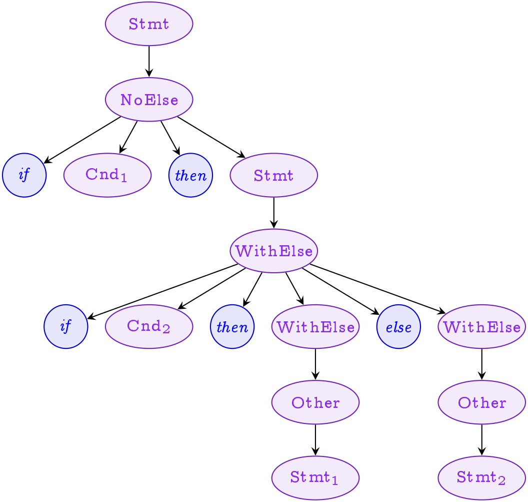


Context-free Grammars

Usually, the first one is considered to be the correct one; ie. *else* belongs to the innermost conditional sentence:

$$
\begin{array}{lcl}
\text{Stmt} & \rightarrow & \text{WithElse} \\
 & | & \text{NoElse} \\
\text{WithElse} & \rightarrow & \textit{if}\ \text{Cnd}\ \textit{then}\ \text{WithElse}\ \textit{else}\ \text{WithElse} \\
 & | & \text{Other} \\
\text{NoElse} & \rightarrow & \textit{if}\ \text{Cnd}\ \textit{then}\ \text{Stmt} \\
 & | & \textit{if}\ \text{Cnd}\ \textit{then}\ \text{WithElse}\ \textit{else}\ \text{NoElse}
\end{array}
$$

Context-free Grammars



Parsing Techniques

Top-down parsing:

- starts constructing the syntax-tree from the root downward towards leaves;
- on every step selects a production rule and tries to match it with the input string;
- if the rule doesn't match the process backtracks;
- results to the leftmost derivation.

Bottom-up parsing:

- starts constructing the syntax-tree from leaves working up toward the root;
- applies suitable rules from right to left until reaches the start symbol;
- results to the rightmost derivation.

Top-down Parsing

General algorithm of top-down parsing:

- construct a root node, label it with the start symbol, and continue construction of the tree towards leaves from left to right;
- if the node under consideration is a non-terminal A, then choose a rule in the form of $A \to \alpha$, construct nodes corresponding to its RHS, and continue with its leftmost subnode.
- if the node is a terminal which doesn't match with the input symbol, then backtrack to the choice of the production rule which introduced this terminal, and continue from there by choosing another production rule;
- if the node is terminal matching to the input symbol, then continue with the leftmost unexpanded node.

Top-down Parsing

Example:

$$
\begin{array}{lll}
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T \,/\, F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & id \\
 & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

Top-down Parsing

Example:

$$
\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{E} + \mathtt{T} \\
&\mid \mathtt{E} - \mathtt{T} \\
&\mid \mathtt{T} \\
\mathtt{T} &\rightarrow \mathtt{T} * \mathtt{F} \\
&\mid \mathtt{T} \,/\, \mathtt{F} \\
&\mid \mathtt{F} \\
\mathtt{F} &\rightarrow (\ \mathtt{E}\ ) \\
&\mid id \\
&\mid num
\end{aligned}
$$

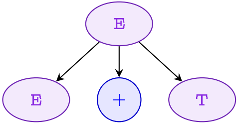
	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

Top-down Parsing

Example:

$$
\begin{aligned}
E &\rightarrow E + T \\
&\mid E - T \\
&\mid T \\
T &\rightarrow T * F \\
&\mid T \,/\, F \\
&\mid F \\
F &\rightarrow (E) \\
&\mid id \\
&\mid num
\end{aligned}
$$

	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

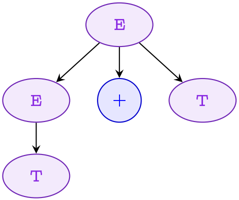


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & \mid & E - T \\
  & \mid & T \\
T & \rightarrow & T * F \\
  & \mid & T \,/\, F \\
  & \mid & F \\
F & \rightarrow & (\ E\ ) \\
  & \mid & id \\
  & \mid & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet num * id$
$T.1$	$id - \bullet num * id$
$T.3$	$id - \bullet num * id$
$F.3$	$id - \bullet num * id$
	$id - num \bullet * id$
	$id - num * \bullet id$
$F.2$	$id - num * \bullet id$
	$id - num * id \bullet$

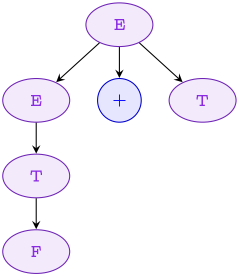


Top-down Parsing

Example:

$$
\begin{aligned}
E &\rightarrow E + T \\
&\mid\ E - T \\
&\mid\ T \\
T &\rightarrow T * F \\
&\mid\ T\ /\ F \\
&\mid\ F \\
F &\rightarrow (\ E\ ) \\
&\mid\ id \\
&\mid\ num
\end{aligned}
$$

	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id\ \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num\ \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id\ \bullet$

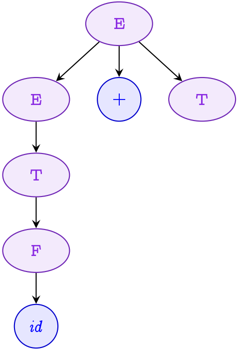


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & | & E - T \\
  & | & T \\
T & \rightarrow & T * F \\
  & | & T \;/\; F \\
  & | & F \\
F & \rightarrow & (E) \\
  & | & id \\
  & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

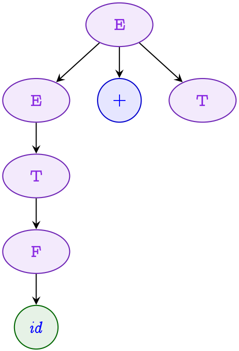


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & \mid & E - T \\
  & \mid & T \\
T & \rightarrow & T * F \\
  & \mid & T \,/\, F \\
  & \mid & F \\
F & \rightarrow & (\ E\ ) \\
  & \mid & id \\
  & \mid & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

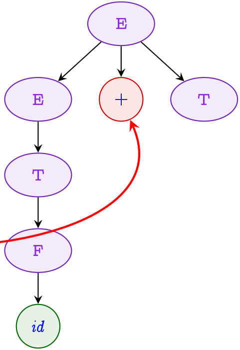


Top-down Parsing

Example:

$$
\begin{array}{lcl}
\mathtt{E} & \rightarrow & \mathtt{E} + \mathtt{T} \\
& | & \mathtt{E} - \mathtt{T} \\
& | & \mathtt{T} \\
\mathtt{T} & \rightarrow & \mathtt{T} * \mathtt{F} \\
& | & \mathtt{T} \,/\, \mathtt{F} \\
& | & \mathtt{F} \\
\mathtt{F} & \rightarrow & (\ \mathtt{E}\ ) \\
& | & id \\
& | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - \ num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$



Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & \mid & E - T \\
  & \mid & T \\
T & \rightarrow & T * F \\
  & \mid & T \,/\, F \\
  & \mid & F \\
F & \rightarrow & (\ E\ ) \\
  & \mid & id \\
  & \mid & num
\end{array}
$$

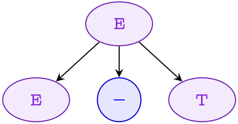
	$\bullet\ id - num * id$
$E.1$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

Top-down Parsing

Example:

E	$\rightarrow$	E + T
	|	E − T
	|	T
T	$\rightarrow$	T $*$ F
	|	T / F
	|	F
F	$\rightarrow$	(E)
	|	*id*
	|	*num*

	• $id - num * id$
$E.2$	• $id - num * id$
$E.3$	• $id - num * id$
$T.3$	• $id - num * id$
$F.2$	• $id - num * id$
	id • $- num * id$
	$id -$ • $num * id$
$T.1$	$id -$ • $num * id$
$T.3$	$id -$ • $num * id$
$F.3$	$id -$ • $num * id$
	$id - num$ • $* id$
	$id - num *$ • id
$F.2$	$id - num *$ • id
	$id - num * id$ •

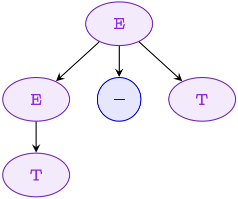


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & \mid & E - T \\
 & \mid & T \\
T & \rightarrow & T * F \\
 & \mid & T \,/\, F \\
 & \mid & F \\
F & \rightarrow & (\, E \,) \\
 & \mid & id \\
 & \mid & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

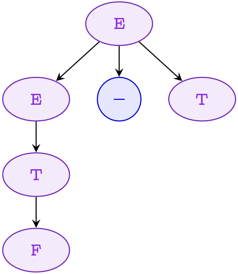


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T \;/\; F \\
 & | & F \\
F & \rightarrow & (\; E\;) \\
 & | & id \\
 & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

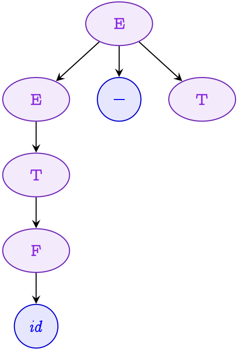


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & \mid & E - T \\
  & \mid & T \\
T & \rightarrow & T * F \\
  & \mid & T \,/\, F \\
  & \mid & F \\
F & \rightarrow & (\ E\ ) \\
  & \mid & id \\
  & \mid & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

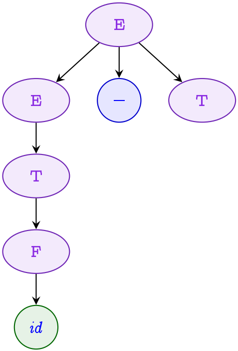


Top-down Parsing

Example:

$$
\begin{aligned}
E &\rightarrow E + T \\
&\;|\;\; E - T \\
&\;|\;\; T \\
T &\rightarrow T * F \\
&\;|\;\; T \,/\, F \\
&\;|\;\; F \\
F &\rightarrow (\, E \,) \\
&\;|\;\; id \\
&\;|\;\; num
\end{aligned}
$$

	$\bullet\; id - num * id$
$E.2$	$\bullet\; id - num * id$
$E.3$	$\bullet\; id - num * id$
$T.3$	$\bullet\; id - num * id$
$F.2$	$\bullet\; id - num * id$
	$id \bullet - num * id$
	$id - \bullet\; num * id$
$T.1$	$id - \bullet\; num * id$
$T.3$	$id - \bullet\; num * id$
$F.3$	$id - \bullet\; num * id$
	$id - num \bullet * id$
	$id - num * \bullet\; id$
$F.2$	$id - num * \bullet\; id$
	$id - num * id \bullet$

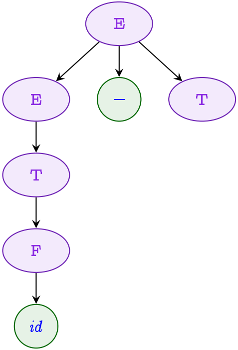


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & | & E - T \\
  & | & T \\
T & \rightarrow & T * F \\
  & | & T \ / \ F \\
  & | & F \\
F & \rightarrow & (\ E \ ) \\
  & | & id \\
  & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

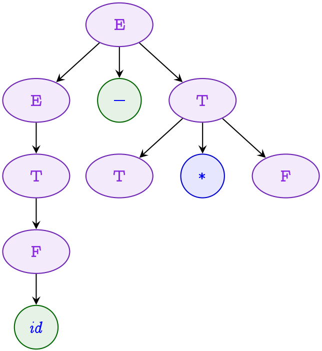


Top-down Parsing

Example:

$$
\begin{aligned}
E &\rightarrow E + T \\
  &\mid E - T \\
  &\mid T \\
T &\rightarrow T * F \\
  &\mid T \, / \, F \\
  &\mid F \\
F &\rightarrow (\, E \,) \\
  &\mid id \\
  &\mid num
\end{aligned}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

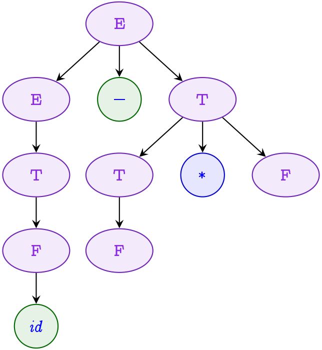


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T \,/\, F \\
 & | & F \\
F & \rightarrow & (\ E\ ) \\
 & | & id \\
 & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

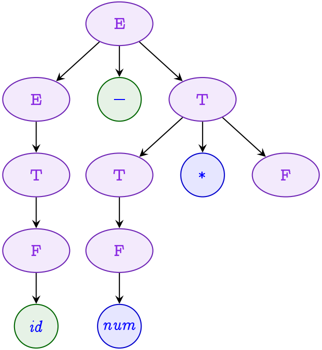


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & | & E - T \\
  & | & T \\
T & \rightarrow & T * F \\
  & | & T \;/\; F \\
  & | & F \\
F & \rightarrow & (\ E\ ) \\
  & | & id \\
  & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

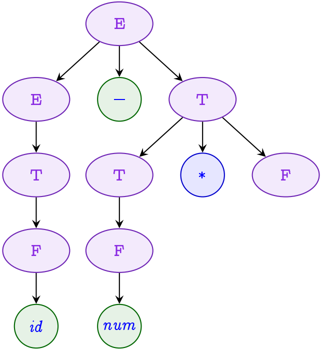


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & T \,/\, F \\
 & | & F \\
F & \rightarrow & (\ E\ ) \\
 & | & id \\
 & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

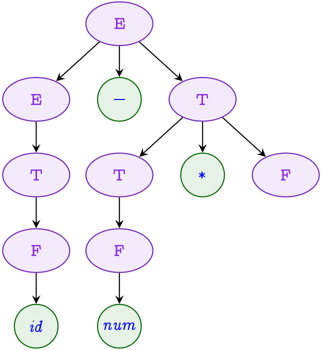


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & | & E - T \\
  & | & T \\
T & \rightarrow & T * F \\
  & | & T \,/\, F \\
  & | & F \\
F & \rightarrow & (\ E\ ) \\
  & | & id \\
  & | & num
\end{array}
$$

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

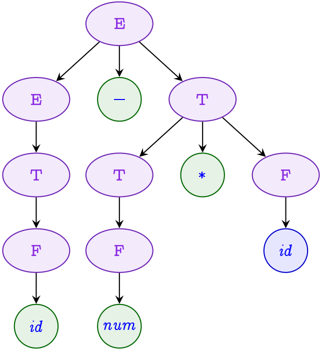


Top-down Parsing

Example:

E	$\rightarrow$	E + T
	$\mid$	E − T
	$\mid$	T
T	$\rightarrow$	T $*$ F
	$\mid$	T / F
	$\mid$	F
F	$\rightarrow$	(E)
	$\mid$	*id*
	$\mid$	*num*

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.3$	$\bullet\ id - num * id$
$T.3$	$\bullet\ id - num * id$
$F.2$	$\bullet\ id - num * id$
	$id \bullet - num * id$
	$id - \bullet\ num * id$
$T.1$	$id - \bullet\ num * id$
$T.3$	$id - \bullet\ num * id$
$F.3$	$id - \bullet\ num * id$
	$id - num \bullet * id$
	$id - num * \bullet\ id$
$F.2$	$id - num * \bullet\ id$
	$id - num * id \bullet$

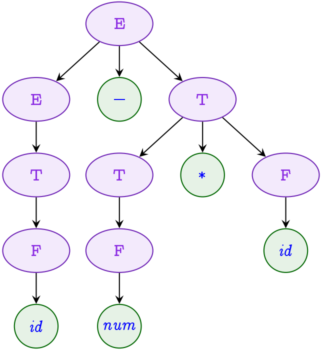


Top-down Parsing

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
  & | & E - T \\
  & | & T \\
T & \rightarrow & T * F \\
  & | & T \, / \, F \\
  & | & F \\
F & \rightarrow & (\; E \;) \\
  & | & id \\
  & | & num
\end{array}
$$

	$\bullet \; id - num * id$
$E.2$	$\bullet \; id - num * id$
$E.3$	$\bullet \; id - num * id$
$T.3$	$\bullet \; id - num * id$
$F.2$	$\bullet \; id - num * id$
	$id \bullet - num * id$
	$id - \bullet \; num * id$
$T.1$	$id - \bullet \; num * id$
$T.3$	$id - \bullet \; num * id$
$F.3$	$id - \bullet \; num * id$
	$id - num \bullet * id$
	$id - num * \bullet \; id$
$F.2$	$id - num * \bullet \; id$
	$id - num * id \bullet$



Top-down Parsing

- Efficiency of parsing strongly depends from the choice of a production rule.
- Choosing a wrong rule causes a later backtracking.
- In the case of the grammar has left-recursive rules, the top-down parsing may be non-terminating.

	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
$E.2$	$\bullet\ id - num * id$
	$\dots$

Top-down Parsing

- Efficiency of parsing strongly depends from the choice of a production rule.
- Choosing a wrong rule causes a later backtracking.
- In the case of the grammar has left-recursive rules, the top-down parsing may be non-terminating.

- $id - num * id$

$E.2$ • $id - num * id$

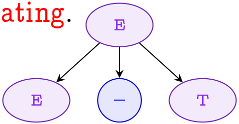
$E.2$ • $id - num * id$

$E.2$ • $id - num * id$

...

Top-down Parsing

- Efficiency of parsing strongly depends from the choice of a production rule.
- Choosing a wrong rule causes a later backtracking.
- In the case of the grammar has left-recursive rules, the top-down parsing may be non-terminating.

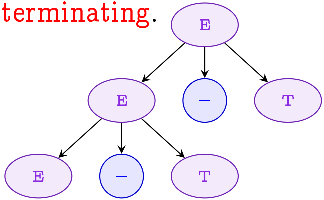


	• $id - num * id$
$E.2$	• $id - num * id$
$E.2$	• $id - num * id$
$E.2$	• $id - num * id$

...

Top-down Parsing

- Efficiency of parsing strongly depends from the choice of a production rule.
- Choosing a wrong rule causes a later backtracking.
- In the case of the grammar has left-recursive rules, the top-down parsing may be non-terminating.



- $id - num * id$

$E.2$ • $id - num * id$

$E.2$ • $id - num * id$

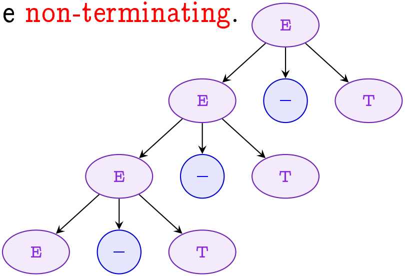
$E.2$ • $id - num * id$

...

Top-down Parsing

- Efficiency of parsing strongly depends from the choice of a production rule.
- Choosing a wrong rule causes a later backtracking.
- In the case of the grammar has left-recursive rules, the top-down parsing may be non-terminating.

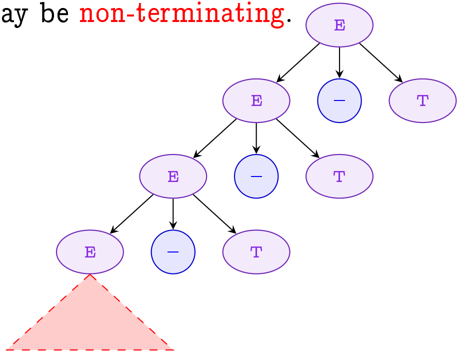
	• $id - num * id$
$E.2$	• $id - num * id$
$E.2$	• $id - num * id$
$E.2$	• $id - num * id$
	...



Top-down Parsing

- Efficiency of parsing strongly depends from the choice of a production rule.
- Choosing a wrong rule causes a later backtracking.
- In the case of the grammar has left-recursive rules, the top-down parsing may be non-terminating.

	• $id - num * id$
$E.2$	• $id - num * id$
$E.2$	• $id - num * id$
$E.2$	• $id - num * id$
	...



Left-recursion

- A grammar is left-recursive, if there is a non-terminal $A \in N$ such that

$$A \Longrightarrow^{+} A\,\alpha,$$

  where $\alpha \in V^{\star}$.

- Left-recursion is direct, if there is a rule in the form $A \rightarrow A\,\alpha$.
- Otherwise, the left-recursion is indirect.

Left-recursion Elimination

Elimination of the direct left-recursion:

- Introduce a new non-terminal and replace the left-recursion with the right-recursion

$$A \rightarrow A\,\alpha \mid \beta \quad \longrightarrow \quad \begin{array}{lll} A & \rightarrow & \beta\ A' \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \end{array}$$

- In general

$$A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \ldots$$

$$\longrightarrow \quad \begin{array}{lll} A & \rightarrow & \beta_1\ A' \mid \beta_2\ A' \mid \ldots \\ A' & \rightarrow & \alpha_2\ A' \mid \alpha_2\ A' \mid \ldots \mid \varepsilon \end{array}$$

Left-recursion Elimination

Example:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & E - T \\
 & | & T
\end{array}
\qquad
\begin{array}{lcl}
T & \rightarrow & T * F \\
 & | & T \,/\, F \\
 & | & F
\end{array}
$$

$\downarrow$ $\downarrow$

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \\
 & | & -\ T\ E' \\
 & | & \varepsilon
\end{array}
\qquad
\begin{array}{lcl}
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \\
 & | & /\ F\ T' \\
 & | & \varepsilon
\end{array}
$$

- The new grammar generates the same language, but is less intuitive.
- In both grammars the operators are left associative.

Left-recursion Elimination

Elimination of the indirect left-recursion:

- Example:

$$\begin{array}{lcl} A_1 & \rightarrow & A_2\ \alpha \mid \beta \\ A_2 & \rightarrow & A_1\ \gamma \mid A_2\ \delta \end{array}$$

- Transform the indirect left-recursion to the direct one.
- In the right-hand sides of A_2 production rules, replace all occurrences of A_1 with its definition:

$$\begin{array}{lcl} A_1 & \rightarrow & A_2\ \alpha \mid \beta \\ A_2 & \rightarrow & A_2\ \alpha\gamma \mid \beta\gamma \mid A_2\ \delta \end{array}$$

- Eliminate the immediate left-recursion:

$$\begin{array}{lcl} A_1 & \rightarrow & A_2\ \alpha \mid \beta \\ A_2 & \rightarrow & \beta\ \gamma\ A_2' \\ A_2' & \rightarrow & \alpha\ \gamma\ A_2' \mid \delta\ A_2' \mid \varepsilon \end{array}$$

Left-recursion Elimination

General algorithm for left-recursion elimination:

Assign some order to non-terminals $A_1, \ldots, A_n$

for $i \leftarrow 1$ **to** n

for $j \leftarrow 1$ **to** $i - 1$

Replace productions in the form $A_i \rightarrow A_j\ \alpha$
with productions $A_i \rightarrow \beta_1\ \alpha \mid \beta_2\ \alpha \mid \ldots \mid \beta_k\ \alpha$,
where $A_j \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_k$ are all A_j productions

Eliminate the immediate left-recursion from productions of the non-terminal A_i

NB! Assumes that the original grammar doesn't have neither ε-productions nor cycles (ie. $A_i \Longrightarrow^+ A_i$).

Left-recursion Elimination

Example:

$$\begin{array}{lcl} A & \to & C\ \alpha \mid A\ \alpha \\ B & \to & A\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & A\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & C\ \alpha\ A'\ \beta\ \delta \mid \gamma\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \to & \alpha\ A' \mid \varepsilon \\ C' & \to & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ C & \to & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \to & \alpha\ A' \mid \varepsilon \\ C' & \to & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array}$$

Left-recursion Elimination

Example:

$$\begin{array}{lcl} A & \to & C\ \alpha \mid A\ \alpha \\ B & \to & A\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & A\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & C\ \alpha\ A'\ \beta\ \delta \mid \gamma\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \to & \alpha\ A' \mid \varepsilon \\ C' & \to & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ C & \to & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \to & \alpha\ A' \mid \varepsilon \\ C' & \to & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array}$$

Left-recursion Elimination

Example:

$$\begin{array}{lcl} A & \rightarrow & C\ \alpha \mid A\ \alpha \\ B & \rightarrow & A\ \beta \mid \gamma \\ C & \rightarrow & B\ \delta \mid \varphi \end{array} \longrightarrow \begin{array}{lcl} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & A\ \beta \mid \gamma \\ C & \rightarrow & B\ \delta \mid \varphi \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \rightarrow & B\ \delta \mid \varphi \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \rightarrow & C\ \alpha\ A'\ \beta\ \delta \mid \gamma\ \delta \mid \varphi \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \rightarrow & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \\ C' & \rightarrow & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \rightarrow & C\ \alpha\ A' \\ C & \rightarrow & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \\ C' & \rightarrow & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array}$$

Left-recursion Elimination

Example:

$$\begin{aligned} A &\rightarrow C\ \alpha \mid A\ \alpha \\ B &\rightarrow A\ \beta \mid \gamma \\ C &\rightarrow B\ \delta \mid \varphi \end{aligned} \longrightarrow \begin{aligned} A &\rightarrow C\ \alpha\ A' \\ B &\rightarrow A\ \beta \mid \gamma \\ C &\rightarrow B\ \delta \mid \varphi \\ A' &\rightarrow \alpha\ A' \mid \varepsilon \end{aligned}$$

$$\longrightarrow \begin{aligned} A &\rightarrow C\ \alpha\ A' \\ B &\rightarrow C\ \alpha\ A'\ \beta \mid \gamma \\ C &\rightarrow B\ \delta \mid \varphi \\ A' &\rightarrow \alpha\ A' \mid \varepsilon \end{aligned} \longrightarrow \begin{aligned} A &\rightarrow C\ \alpha\ A' \\ B &\rightarrow C\ \alpha\ A'\ \beta \mid \gamma \\ C &\rightarrow C\ \alpha\ A'\ \beta\ \delta \mid \gamma\ \delta \mid \varphi \\ A' &\rightarrow \alpha\ A' \mid \varepsilon \end{aligned}$$

$$\longrightarrow \begin{aligned} A &\rightarrow C\ \alpha\ A' \\ B &\rightarrow C\ \alpha\ A'\ \beta \mid \gamma \\ C &\rightarrow \alpha\ A'\ \beta\ \delta\ C' \\ A' &\rightarrow \alpha\ A' \mid \varepsilon \\ C' &\rightarrow \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{aligned} \longrightarrow \begin{aligned} A &\rightarrow C\ \alpha\ A' \\ C &\rightarrow \alpha\ A'\ \beta\ \delta\ C' \\ A' &\rightarrow \alpha\ A' \mid \varepsilon \\ C' &\rightarrow \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{aligned}$$

Left-recursion Elimination

Example:

$$\begin{array}{lll} A & \rightarrow & C\ \alpha \mid A\ \alpha \\ B & \rightarrow & A\ \beta \mid \gamma \\ C & \rightarrow & B\ \delta \mid \varphi \end{array} \longrightarrow \begin{array}{lll} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & A\ \beta \mid \gamma \\ C & \rightarrow & B\ \delta \mid \varphi \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lll} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \rightarrow & B\ \delta \mid \varphi \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lll} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \rightarrow & C\ \alpha\ A'\ \beta\ \delta \mid \gamma\ \delta \mid \varphi \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lll} A & \rightarrow & C\ \alpha\ A' \\ B & \rightarrow & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \rightarrow & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \\ C' & \rightarrow & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lll} A & \rightarrow & C\ \alpha\ A' \\ C & \rightarrow & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \rightarrow & \alpha\ A' \mid \varepsilon \\ C' & \rightarrow & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array}$$

Left-recursion Elimination

Example:

$$\begin{array}{lcl} A & \to & C\ \alpha \mid A\ \alpha \\ B & \to & A\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & A\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & B\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & C\ \alpha\ A'\ \beta\ \delta \mid \gamma\ \delta \mid \varphi \\ A' & \to & \alpha\ A' \mid \varepsilon \end{array}$$

$$\longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ B & \to & C\ \alpha\ A'\ \beta \mid \gamma \\ C & \to & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \to & \alpha\ A' \mid \varepsilon \\ C' & \to & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array} \longrightarrow \begin{array}{lcl} A & \to & C\ \alpha\ A' \\ C & \to & \alpha\ A'\ \beta\ \delta\ C' \\ A' & \to & \alpha\ A' \mid \varepsilon \\ C' & \to & \gamma\ \delta\ C' \mid \varphi\ C' \mid \varepsilon \end{array}$$

Predictive Parsing

- Choosing a wrong rule causes a later backtracking.
- The "rightness" of the rule can often be decided by lookahead of some number of input symbols.
- In general case, one needs unbounded lookahead.
  - Ex.: parsing algorithms by Cocke-Younger-Kasam or Earley.
- Predictive parsing is a top-down parsing where it is always possible to choose a correct rule without backtracking.
  - A grammar must be such, that the next input symbol (or some fixed number of symbols) determines uniquely the correct rule.

Predictive Parsing

- For every sentential form $\alpha \in (N \cup T)^\star$ define a set:

$$\begin{aligned} \mathit{first}(\alpha) &= \{a \in T \mid \alpha \Longrightarrow^\star a\,\beta\} \\ &\cup \{\varepsilon \mid \alpha \Longrightarrow^\star \varepsilon\} \end{aligned}$$

  where $\beta \in (N \cup T)^\star$.

- For every non-terminal $A \in N$ define a set:

$$\begin{aligned} \mathit{follow}(A) &= \{a \in T \mid S \Longrightarrow^\star \alpha\,A\,a\,\beta\} \\ &\cup \{\$ \mid S \Longrightarrow^\star \alpha\,A\} \end{aligned}$$

  where $\alpha, \beta \in (N \cup T)^\star$ and \$ is a special end of input marker.

Predictive Parsing

Example:

$$\begin{array}{lcl} S & \rightarrow & A\,B\,C \\ A & \rightarrow & a\,A \mid \varepsilon \end{array} \qquad \begin{array}{lcl} B & \rightarrow & b \mid \varepsilon \\ C & \rightarrow & c \mid d \end{array}$$

$$\begin{array}{lcl} \mathit{first}(C) & = & \{c, d\} \\ \mathit{first}(B) & = & \{b, \varepsilon\} \\ \mathit{first}(A) & = & \{a, \varepsilon\} \\ \mathit{first}(S) & = & \mathit{first}(A\,B\,C) \\ & = & (\mathit{first}(A) \setminus \{\varepsilon\}) \\ & & \cup\ (\mathit{first}(B) \setminus \{\varepsilon\}) \\ & & \cup\ \mathit{first}(C) \\ & = & \{a, b, c, d\} \end{array}$$

$$\begin{array}{lcl} \mathit{follow}(C) & = & \{\$\} \\ \mathit{follow}(B) & = & \mathit{first}(C) \\ & = & \{c, d\} \\ \mathit{follow}(A) & = & (\mathit{first}(B) \setminus \{\varepsilon\}) \\ & & \cup\ \mathit{first}(C) \\ & = & \{b, c, d\} \end{array}$$

Predictive Parsing

- If production rules $A \to \alpha$ and $A \to \beta$ are such that $\mathit{first}(\alpha) \cap \mathit{first}(\beta) = \emptyset$, then lookahead of one input symbol is enough to decide which rule to choose.
- NB! Holds only when $\varepsilon \notin \{\mathit{first}(\alpha) \cup \mathit{first}(\beta)\}$.
- Otherwise, the set $\mathit{follow}(A)$ should also be inspected.
- For each rule $A \to \alpha$ define a set:

$$\mathit{first}^+(\alpha) = \begin{cases} (\mathit{first}(\alpha) \setminus \{\varepsilon\}) \cup \mathit{follow}(A), & \text{kui } \varepsilon \in \mathit{first}(\alpha) \\ \mathit{first}(\alpha), & \text{kui } \varepsilon \notin \mathit{first}(\alpha) \end{cases}$$

- A CF-grammar is LL(1), if for all (pairwise different) production rules $A \to \alpha$ and $A \to \beta$

$$\mathit{first}^+(\alpha) \cap \mathit{first}^+(\beta) = \emptyset$$

Predictive Parsing

- NB! LL(1) grammar cannot be neither left-recursive nor ambiguous!
- Example:

$$S \rightarrow S\,a \mid \beta$$

- If $\beta \neq \varepsilon$
  - Then $\mathit{first}(\beta) \subseteq \mathit{first}(S) = \mathit{first}(S\,a)$
  - Thus $\mathit{first}^+(S\,a) \cap \mathit{first}^+(\beta) \neq \emptyset$
- If $\beta = \varepsilon$
  - Then $a \in \mathit{first}(S\,a)$ and $a \in \mathit{follow}(S) = \{a, \$\}$
  - Thus $\mathit{first}^+(S\,a) \cap \mathit{first}^+(\beta) \neq \emptyset$

Predictive Parsing

- Often, a grammar can be transformed to LL(1) using:
  - left-recursion elimination;
  - left-factoring;
  - in worst case, one can generalize the grammar a bit (and check for removed restrictions after parsing).
- Left-factoring replaces rules with a common prefix with new ones, where the prefix is only in one RHS.
- Example:

$$\begin{array}{lcl} A & \to & B\,a\,C\,D \\ & \mid & B\,a\,C\,E \end{array} \quad \longrightarrow \quad \begin{array}{lcl} A & \to & B\,a\,C\,Z \\ Z & \to & D \\ & \mid & E \end{array}$$

Predictive Parsing

Left-factoring algorithm:

1. For every non-terminal $A \in N$ find a longest prefix α which appears in two or more right-hand sides of A production rules.
2. If $\alpha \neq \varepsilon$, then replace all productions of A

$$A \rightarrow \alpha\,\beta_1 \mid \alpha\,\beta_2 \mid \ldots \mid \alpha\,\beta_n \mid \gamma$$

   with productions

$$\begin{aligned} A &\rightarrow \alpha\,Z \mid \gamma \\ Z &\rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{aligned}$$

   where $Z \in N$ is a new non-terminal.
3. Repeat the process until none of right-hand sides have common prefix.

Predictive Parsing

- Recursive descent parsing is a top-down parsing method where:
  - parser consists of a set of (mutually recursive) procedures, one for each non-terminal, recognizing sentential forms derivable from the corresponding non-terminal;
  - depending from the input, each procedure chooses a production rule and calls one after another procedures corresponding to non-terminals in the right-hand side of the production rule.
- Recursive descent parsing is commonly used for hand-written parsers.

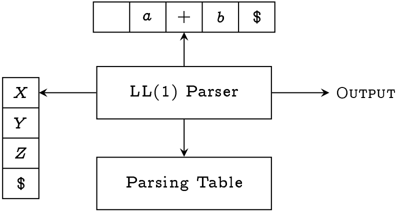
Predictive Parsing

Example: let non-terminal A have production rules
$A \to a\,B \mid b\,C\,A \mid D\,E$

$parseA()$ {
  if $token = a$ **then** {
    $token := nextWord();\ parseB();$
  } **else if** $token = b$ **then** {
    $token := nextWord();\ parseC();\ parseA();$
  } **else** {
    $parseD();\ parseE();$
  }
}

Predictive Parsing

- Automatically generated LL(1) parsers are usually table driven:
  - construct a table M where rows and columns are indexed by non-terminals and terminals respectively;
  - cells of the table contain production rules to be chosen for the given non-terminal and input symbol.
- Structure of a table driven LL(1) parser:



Predictive Parsing

LL(1) parsing algorithm:

$push(\$);\ \ push(S);$
$token := nextWord();$
while $stack \neq empty$ **do** {
  $A := pop();$
  if $A \in N$ **then** {
    if $M[A, token] = B_1 \ldots B_n$ **then** {
      $push(B_n); \ldots; push(B_1);$
    } **else** $error();$
  } **else if** $A = token$ **then** {
    $token := nextWord();$
  } **else** $error();$
}

Shift-reduce Parsing

Shift-reduce parsing is a general method for a bottom-up syntax analysis:

- constructing a tree starts from leaves working up toward the root with a goal of "reducing" the input string to the start symbol.
- during parsing there is a forest of trees, which correspond to the different, already recognized, substrings;
- two basic actions:
  - shift reads a new input symbol and pushes it to the stack;
  - reduce applies production rules in reverse to the top of the stack; ie. it replaces a sequence of symbols in the top of the stack, forming a right-hand side of a rule, with the left-hand symbol of that rule;
- construction corresponds to the right derivation.

Shift-reduce Parsing

Example:

Input String: $\bullet\, a\, b\, c\, c\, d\, e$

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

shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\, c\, A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\, A\, B\, e$

$$\begin{array}{lllll} S & \Longrightarrow_{rm} & a\, A\, B\, e & \Longrightarrow_{rm} & a\, A\, d\, e \\ & \Longrightarrow_{rm} & a\, b\, c\, A\, d\, e & \Longrightarrow_{rm} & a\, b\, c\, c\, d\, e \end{array}$$

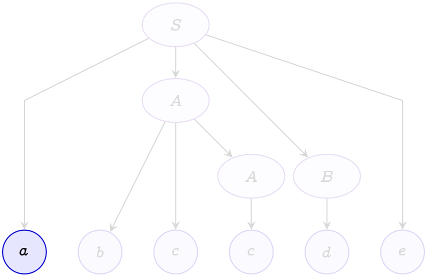
Shift-reduce Parsing

Example:

Input String: $a \bullet b\,c\,c\,d\,e$

$$S \rightarrow a\,A\,B\,e$$
$$A \rightarrow b\,c\,A \mid c$$
$$B \rightarrow d$$

shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$



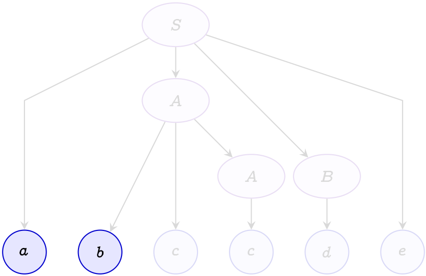
$$S \Longrightarrow_{rm} a\,A\,B\,e \Longrightarrow_{rm} a\,A\,d\,e$$
$$\Longrightarrow_{rm} a\,b\,c\,A\,d\,e \Longrightarrow_{rm} a\,b\,c\,c\,d\,e$$

Shift-reduce Parsing

Example:

Input String: $a\, b \bullet c\, c\, d\, e$

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



shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\, c\, A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\, A\, B\, e$

$$
\begin{array}{llll}
S & \Longrightarrow_{rm} a\, A\, B\, e & \Longrightarrow_{rm} & a\, A\, d\, e \\
 & \Longrightarrow_{rm} a\, b\, c\, A\, d\, e & \Longrightarrow_{rm} & a\, b\, c\, c\, d\, e
\end{array}
$$

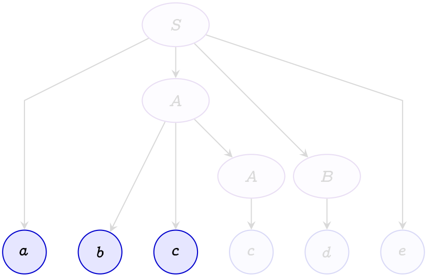
Shift-reduce Parsing

Example:

Input String: $a\,b\,c \bullet c\,d\,e$

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

shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$



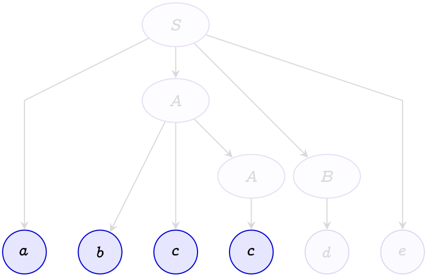
$$\begin{array}{lclcl} S & \Longrightarrow_{rm} & a\,A\,B\,e & \Longrightarrow_{rm} & a\,A\,d\,e \\ & \Longrightarrow_{rm} & a\,b\,c\,A\,d\,e & \Longrightarrow_{rm} & a\,b\,c\,c\,d\,e \end{array}$$

Shift-reduce Parsing

Example:

Input String: $a\,b\,c\,c \bullet d\,e$

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



shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$

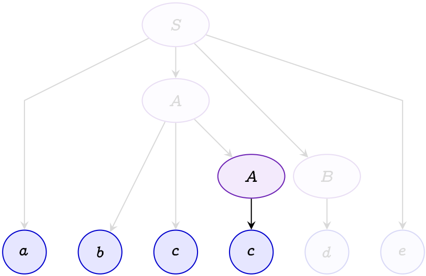
$$\begin{array}{lllll} S & \Longrightarrow_{rm} & a\,A\,B\,e & \Longrightarrow_{rm} & a\,A\,d\,e \\ & \Longrightarrow_{rm} & a\,b\,c\,A\,d\,e & \Longrightarrow_{rm} & a\,b\,c\,c\,d\,e \end{array}$$

Shift-reduce Parsing

Example:

Input String: $a\,b\,c\,c \bullet d\,e$

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



shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$

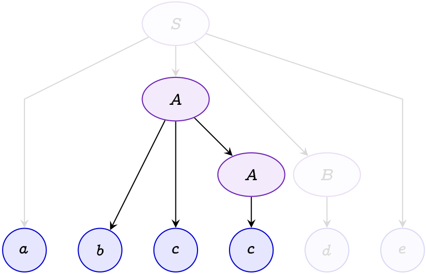
$$
\begin{array}{lllll}
S & \Longrightarrow_{rm} & a\,A\,B\,e & \Longrightarrow_{rm} & a\,A\,d\,e \\
 & \Longrightarrow_{rm} & a\,b\,c\,A\,d\,e & \Longrightarrow_{rm} & a\,b\,c\,c\,d\,e
\end{array}
$$

Shift-reduce Parsing

Example:

Input String: $a\,b\,c\,c \bullet d\,e$

$$
\begin{aligned}
S &\rightarrow a\,A\,B\,e \\
A &\rightarrow b\,c\,A \mid c \\
B &\rightarrow d
\end{aligned}
$$



shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$

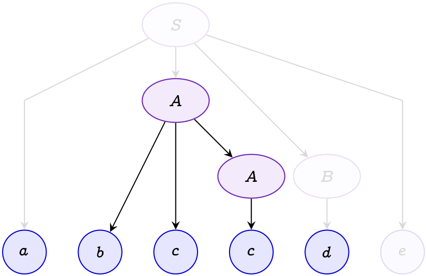
$$
\begin{aligned}
S &\Longrightarrow_{rm} a\,A\,B\,e \Longrightarrow_{rm} a\,A\,d\,e \\
&\Longrightarrow_{rm} a\,b\,c\,A\,d\,e \Longrightarrow_{rm} a\,b\,c\,c\,d\,e
\end{aligned}
$$

Shift-reduce Parsing

Example:

Input String: $a\,b\,c\,c\,d \bullet e$

$$
\begin{aligned}
S &\rightarrow a\,A\,B\,e \\
A &\rightarrow b\,c\,A \mid c \\
B &\rightarrow d
\end{aligned}
$$



shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$

$$
\begin{aligned}
S &\Longrightarrow_{rm} a\,A\,B\,e \Longrightarrow_{rm} a\,A\,d\,e \\
&\Longrightarrow_{rm} a\,b\,c\,A\,d\,e \Longrightarrow_{rm} a\,b\,c\,c\,d\,e
\end{aligned}
$$

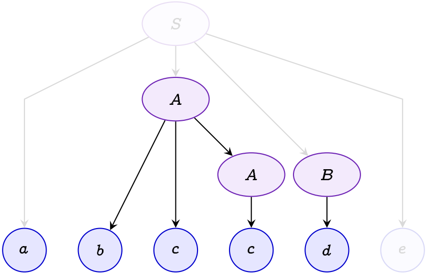
Shift-reduce Parsing

Example:

Input String: $a\,b\,c\,c\,d \bullet e$

$$S \rightarrow a\,A\,B\,e$$
$$A \rightarrow b\,c\,A \mid c$$
$$B \rightarrow d$$

shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$



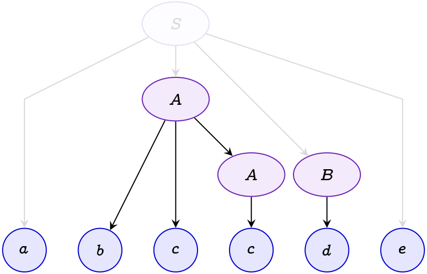
$$S \Longrightarrow_{rm} a\,A\,B\,e \Longrightarrow_{rm} a\,A\,d\,e$$
$$\Longrightarrow_{rm} a\,b\,c\,A\,d\,e \Longrightarrow_{rm} a\,b\,c\,c\,d\,e$$

Shift-reduce Parsing

Example:

Input String: $a\,b\,c\,c\,d\,e\,\bullet$

$$
\begin{aligned}
S &\rightarrow a\,A\,B\,e \\
A &\rightarrow b\,c\,A \mid c \\
B &\rightarrow d
\end{aligned}
$$



shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$

$$
\begin{aligned}
S &\Longrightarrow_{rm} a\,A\,B\,e \Longrightarrow_{rm} a\,A\,d\,e \\
&\Longrightarrow_{rm} a\,b\,c\,A\,d\,e \Longrightarrow_{rm} a\,b\,c\,c\,d\,e
\end{aligned}
$$

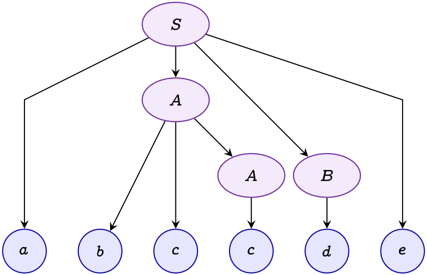
Shift-reduce Parsing

Example:

Input String: $a\,b\,c\,c\,d\,e\,\bullet$

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

shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$



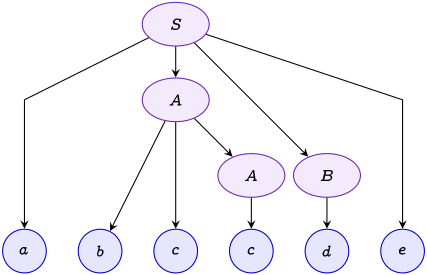
$$\begin{array}{lllll} S & \Longrightarrow_{rm} & a\,A\,B\,e & \Longrightarrow_{rm} & a\,A\,d\,e \\ & \Longrightarrow_{rm} & a\,b\,c\,A\,d\,e & \Longrightarrow_{rm} & a\,b\,c\,c\,d\,e \end{array}$$

Shift-reduce Parsing

Example:

Input String:

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



shift
shift
shift
shift
reduce $A \rightarrow c$
reduce $A \rightarrow b\,c\,A$
shift
reduce $B \rightarrow d$
shift
reduce $S \rightarrow a\,A\,B\,e$

$$
\begin{array}{lllll}
S & \Longrightarrow_{rm} & a\,A\,B\,e & \Longrightarrow_{rm} & a\,A\,d\,e \\
 & \Longrightarrow_{rm} & a\,b\,c\,A\,d\,e & \Longrightarrow_{rm} & a\,b\,c\,c\,d\,e
\end{array}
$$

Shift-reduce Parsing

- Sentential form is called right-sentential form, if it occurs in the rightmost derivation of some sentence.
- A handle of a right-sentential form γ is a production rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.
- Equivalently, a handle is a substring β, such that it matches RHS of some production and $S \Longrightarrow^{\star}_{rm} \delta A w \Longrightarrow_{rm} \delta \beta w = \gamma$, where $\beta, \gamma, \delta \in V^{\star}$ and $w \in T^{\star}$.
- The process of discovering a handle and reducing it to the appropriate LHS is called "handle pruning".
- NB! In the case of an unambiguous grammar, rightmost derivations, and hence handles, are unique.

Shift-reduce Parsing

Example: given the grammar

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

and the rightmost derivation

$$S \Longrightarrow_{rm} a\,A\,B\,e \Longrightarrow_{rm} a\,A\,d\,e \Longrightarrow_{rm} a\,b\,c\,A\,d\,e$$

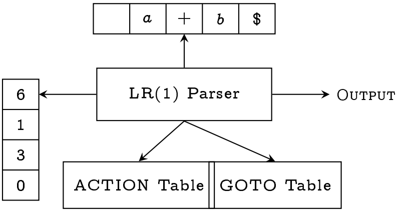
The handle of the right-sentential form $abcAde$ is bcA.

Shift-reduce Parsing

- In a stack based implementation of shift-reduce parser, the handle will always eventually appear on top of the stack.
- A viable prefix of a right-sentential form γ is any prefix of γ ending no farther right than the right end of the handle of γ.
- Viable prefixes are possible stacks of the shift-reduce parser!
- NB! The "language of viable prefixes" is regular!
- Hence, there is a finite automaton accepting viable prefixes.
- This automaton is an essential ingredient of all LR parsing techniques.

LR Parsing

Structure of LR parser:



LR Parsing

Skeleton of LR(1) parser:

$push(\text{Invalid});\ push(s_0);\ found := \texttt{false};$
$token := nextWord();$
while $found \neq$ **true do** {
  $s := top();$
  if $ACTION[s, token] =$ **reduce**$(A \rightarrow \beta)$ **then**
    $pop(2 * |\beta|);$
  $s := top();\ push(A);\ push(GOTO[s, A]);$
  else if $ACTION[s, token] =$ **shift**(s_i) **then**
    $push(token);\ push(s_i);$
    $token := nextWord();$
  else if $ACTION[s, token] =$ **accept** & $token = \$$ **then**
    $found := \texttt{true};$
  else report error;
}
report success;

LR Parsing

- LR(0)-item (or simply item) is a production rule with a dot in the RHS.
- An item $[A \rightarrow \alpha \cdot \beta]$ is valid for a viable prefix φ if there is a rightmost derivation $S \Longrightarrow^{\star}_{rm} \delta A w \Longrightarrow_{rm} \delta \alpha \beta w$ and $\delta \alpha = \varphi$.
- Item in the form $[A \rightarrow \cdot \alpha]$ is an initial item and in the form $[A \rightarrow \alpha \cdot]$ is a complete item.

LR Parsing

- Example: given a grammar

$$\begin{array}{lcl} S & \rightarrow & a\,A\,c \\ A & \rightarrow & A\,b \mid \varepsilon \end{array}$$

  Its LR(0)-items are:

$$\begin{array}{ll} [S \rightarrow \cdot\ a\,A\,c] & [A \rightarrow \cdot\ A\,b] \\ [S \rightarrow a \cdot A\,c] & [A \rightarrow A \cdot b] \\ [S \rightarrow a\,A \cdot c] & [A \rightarrow A\,b\,\cdot] \\ [S \rightarrow a\,A\,c\,\cdot] & [A \rightarrow \cdot] \end{array}$$

- NB! For each CF-grammar the set of LR(0)-items is finite.

LR Parsing

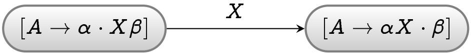
- A valid item $[A \to \cdot \beta\gamma]$ means that the input seen so far is consistent with the use of $A \to \beta\gamma$ immediately after the symbol on top of the stack.
- A valid item $[A \to \beta \cdot \gamma]$ means that the input seen so far is consistent with the use of $A \to \beta\gamma$ at this point of the parse, and that the parser has already recognized β.
- A complete valid item $[A \to \beta\gamma \cdot]$ means that the parser has seen $\beta\gamma$, and that this is consistent with reducing to A.

LR Parsing

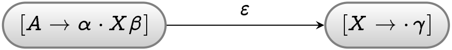
- If $[A \to \alpha \cdot]$ is a complete valid item for a viable prefix γ, then $A \to \alpha$ might have been used to derive γw from $\delta A w$.
- However, in general, it might be also that this was not a case.
  - $[A \to \alpha \cdot]$ may be valid because of a different rightmost derivation $S \Longrightarrow^{\star}_{rm} \delta A w' \Longrightarrow_{rm} \gamma w'$.
  - There could be several complete valid items for γ.
  - There could be a handle of γw that includes some symbols of w.
- A context-free grammar for which knowing a complete valid item is enough to determine the previous right-sentential form is called LR(0).

LR Parsing

- A nondeterministic LR(0)-automaton is a NFA, where states are items and there are two kinds of transitions:
  - for every pair of items $[A \to \alpha \cdot X\beta]$ and $[A \to \alpha X \cdot \beta]$, a transition labelled by a (terminal or nonterminal) symbol X.

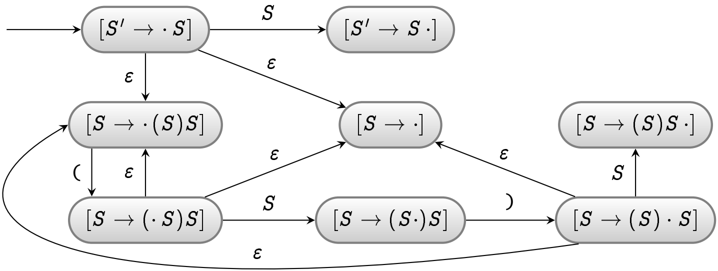


  - for every pair of items $[A \to \alpha \cdot X\beta]$ and $[X \to \cdot\gamma]$, a transition labelled by the empty symbol ε.



- The grammar is augmented with a new start symbol S' and a single rule $S' \to S$.
- A state containing the item $[S' \to \cdot S]$ is the starting state of the automaton.

LR Parsing



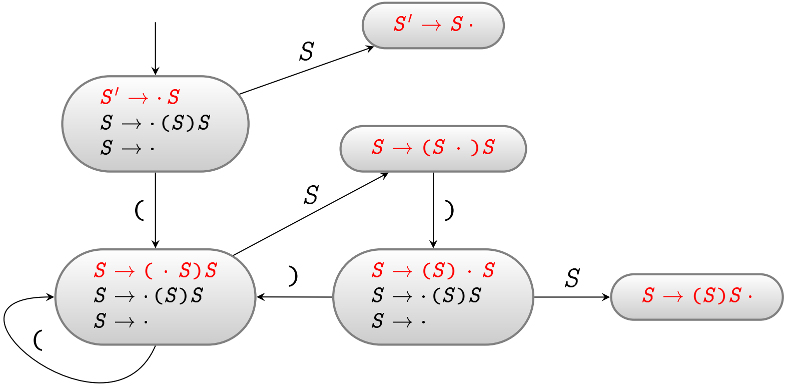
$$
\begin{aligned}
S' &\to S \\
S &\to (\ S\ )\ S \ \mid\ \varepsilon
\end{aligned}
$$

$[S' \to \cdot S]$
$[S' \to S \cdot]$
$[S \to \cdot (\ S\ )\ S]$
$[S \to (\ \cdot S\ )\ S]$
$[S \to (\ S \cdot\ )\ S]$
$[S \to (\ S\ ) \cdot S]$
$[S \to (\ S\ )\ S \cdot]$
$[S \to \cdot]$

LR Parsing

- DFA can be constructed from NFA by subset construction.
- DFA has sets of items as its states.
- Items in sets are called kernel items if they originate as targets of non-ε-transitions.
- Items added to the state during the ε-closure step are called closure items.
- Kernel items uniquely determine the state.

LR Parsing



$S' \to \cdot S$
$S \to \cdot (S) S$
$S \to \cdot$
S
$S' \to S \cdot$
(
S
$S \to (S \cdot) S$
)
$S \to (\cdot S) S$
$S \to \cdot (S) S$
$S \to \cdot$
(
)
$S \to (S) \cdot S$
$S \to \cdot (S) S$
$S \to \cdot$
S
$S \to (S) S \cdot$

LR Parsing

LR(0) parsing algorithm:

while true do {
 $state := top();$
 if $\exists[A \to \alpha \cdot X\beta] \in state \ \wedge X \in T$ **then**
  $Y := getToken();$
  if $\exists[A \to \alpha \cdot Y\beta] \in state$ **then**
   shift$(Y); push([A \to \alpha Y \cdot \beta]);$
  else error;
 else if $\exists[A \to \gamma \cdot] \in state$ **then**
   if $A = S' \wedge \gamma = S$ **then accept**;
   else $pop(2 * |\gamma|);$
     $state := top();$
     if $\exists[B \to \alpha \cdot A\beta] \in state$ **then**
      $push(A); \ push([B \to \alpha A \cdot \beta]);$
     else error;
}

LR Parsing

- A grammar is LR(0) grammar iff any complete item $[A \to \alpha\cdot] \in \mathit{state}$ is the only item of the state.
  - Shift-reduce conflict: if $\exists [B \to \alpha \cdot \beta] \in \mathit{state}$;
  - Reduce-reduce conflict: if $\exists [B \to \beta\cdot] \in \mathit{state}$.
- Reduce states: states containing a complete item.
- Shift states: all other states.

LR Parsing

- SLR(1) = Simple LR(1) parsing.
- Uses the DFA of sets of LR(0) items.
- Uses the next lookahead token in the input string to direct its actions.
- Similar to LR(0) parsing, except that decision on which token to use is delayed until the last possible moment.
  - Consults the input token before a shift to make sure that an appropriate DFA transition exists.
  - Uses the Follow set of a nonterminal to decide if a reduction should be performed.
- Effective extension to LR(0) parsing that is powerful enough to handle many practical languages.

LR Parsing

SLR(1) parsing algorithm:

while true do {
  $state := top();\ \ X := getToken();$
  if $\exists[A \to \alpha \cdot X\beta] \in state$ **then**
    shift(X); $push([A \to \alpha X \cdot \beta]);$
  else if $\exists[A \to \gamma \cdot] \in state \wedge X \in follow(A)$ **then**
    if $A = S' \wedge \gamma = S \wedge X = \$$ **then accept**;
    else $pop(2 * |\gamma|);$
      $state := top();$
      if $\exists[B \to \alpha \cdot A\beta] \in state$ **then**
        $push(A);\ \ push([B \to \alpha A \cdot \beta]);$
      else error;
  else error;
}

LR Parsing

- A grammar is SLR(1) iff it does not have the following two conflicts: for all states
  - Shift-reduce conflict:

$$\begin{aligned} \forall[A \rightarrow \alpha \cdot X\beta] \in \mathit{state} \wedge X \in T \\ \Rightarrow \quad \neg(\exists[B \rightarrow \gamma\cdot] \wedge X \in \mathit{follow}(B)) \end{aligned}$$

  - Reduce-reduce conflict:

$$\forall[A \rightarrow \alpha\cdot] \wedge [B \rightarrow \beta\cdot] \Rightarrow \mathit{follow}(A) \cap \mathit{follow}(B) = \emptyset$$

LR Parsing

- LR(k)-item is a pair $[A \to \alpha \cdot \beta,\ w]$, where $A \to \alpha\beta$ is a production rule and $w \in T^\star$ is a word of length $|w| \leq k$.
- $[A \to \cdot\beta\gamma,\ w]$ means that the input seen so far is consistent with the use of $A \to \beta\gamma$ immediately after the symbol on top of the stack.
- $[A \to \beta \cdot \gamma,\ w]$ means that the input seen so far is consistent with the use of $A \to \beta\gamma$ at this point of the parse, and that the parser has already recognized β.
- $[A \to \beta\gamma\cdot,\ w]$ means that the parser has seen $\beta\gamma$, and that lookahead symbol of w is consistent with reducing to A.

Shift-reduce Parsing

- Example: given a grammar

$$\begin{array}{lcl} S & \rightarrow & a\,A\,c \\ A & \rightarrow & A\,b \mid \varepsilon \end{array}$$

Its LR(k)-items for lookahead string w are:

$$\begin{array}{ll} [S \rightarrow \cdot\ a\,A\,c,\ \ w] & [A \rightarrow \cdot\ A\,b,\ \ w] \\ [S \rightarrow a \cdot A\,c,\ \ w] & [A \rightarrow A \cdot b,\ \ w] \\ [S \rightarrow a\,A \cdot c,\ \ w] & [A \rightarrow A\,b\,\cdot,\ \ w] \\ [S \rightarrow a\,A\,c\,\cdot,\ \ w] & [A \rightarrow \cdot,\ \ w] \end{array}$$

- NB! For each CF-grammar the set of LR(k)-items is finite.

LR Parsing

Finding a closure of LR(1)-items

- $\mathit{Closure}(\mathcal{S})$ adds all the items implied by items already in the set of items $\mathcal{S}$.
  - An item $[A \to \beta \cdot B\delta,\ a]$ implies $[B \to \cdot\tau,\ x]$ for each production $B \to \tau$ and symbol $x \in \mathit{first}(\delta a)$.
- Algorithm:

  $\mathit{Closure}(\mathcal{S})$ {
    while ($\mathcal{S}$ is still changing) **do** {
      $\forall[A \to \beta \cdot B\delta,\ a] \in \mathcal{S}$
        $\forall B \to \tau \in P$
          $\forall b \in \mathit{first}(\delta a)$
            if $[B \to \cdot\tau,\ b] \notin \mathcal{S}$ **then**
              $\mathcal{S} := \mathcal{S} \cup \{[B \to \cdot\tau,\ b]\}$
    }
  }
- The algorithm terminates, as the set of items is finite.

LR Parsing

- $Goto(s, X)$ computes the state that the parser would reach if it recognized $X \in V$ while in state s:

  $$Goto(s, X) = Closure(\,\{[A \rightarrow \beta X \cdot \delta,\ a] \mid [A \rightarrow \beta \cdot X \delta,\ a] \in s\}\,)$$

- Building the Canonical Collection:

  $s_0 := Closure(\{[S' \rightarrow \cdot S,\ \$]\});$
  $\mathcal{S} := \{s_0\};\ \ \Delta := \emptyset;\ \ k := 1;$
  while ($\mathcal{S}$ is still changing) **do** {
  $\quad \forall s_j \in \mathcal{S},\ \ x \in V$
  $\qquad s_k := Goto(s_j, x);$
  $\qquad \Delta := \Delta \cup \{(s_j, x) \mapsto s_k\};$
  $\qquad$ **if** $s_k \notin \mathcal{S}$ **then**
  $\qquad\quad \mathcal{S} := \mathcal{S} \cup \{s_k\};\ \ k := k + 1;$
  }

LR Parsing

Example:

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & T - E \mid T \\
T & \rightarrow & F * T \mid F \\
F & \rightarrow & n
\end{array}
$$

Symbol	*first*
S	$\{n\}$
E	$\{n\}$
T	$\{n\}$
F	$\{n\}$
$-$	$\{-\}$
$*$	$\{*\}$
n	$\{n\}$

Start state:

$$
\begin{array}{lcll}
s_0 & = & & Closure(\{[S \rightarrow \cdot E,\ \$]\}) \\
 & = & \{ & [S \rightarrow \cdot E,\ \$],\ [E \rightarrow \cdot T - E,\ \$],\ [E \rightarrow \cdot T,\ \$], \\
 & & & [T \rightarrow \cdot F * T,\ \$],\ [T \rightarrow \cdot F * T,\ -],\ [T \rightarrow \cdot F,\ \$], \\
 & & & [T \rightarrow \cdot F,\ -],\ [F \rightarrow \cdot n,\ \$], [F \rightarrow \cdot n,\ -],\ [F \rightarrow \cdot n,\ *] \quad \}
\end{array}
$$

LR Parsing

1st iteration:

$$
\begin{array}{rcll}
s_1 & = & & Goto(s_0, E) \\
 & = & \{ & [S \rightarrow E\cdot,\ \$] \hspace{6em} \} \\
s_2 & = & & Goto(s_0, T) \\
 & = & \{ & [E \rightarrow T \cdot - E,\ \$],\ [E \rightarrow T\cdot,\ \$] \hspace{3em} \} \\
s_3 & = & & Goto(s_0, F) \\
 & = & \{ & [T \rightarrow F \cdot * T,\ \$],\ [T \rightarrow F \cdot * T,\ -], \\
 & & & [T \rightarrow F\cdot,\ \$],\ [T \rightarrow F\cdot,\ -] \hspace{3em} \} \\
s_4 & = & & Goto(s_0, n) \\
 & = & \{ & [F \rightarrow n\cdot,\ \$],\ [F \rightarrow n\cdot,\ -],\ [F \rightarrow n\cdot,\ *] \quad \}
\end{array}
$$

LR Parsing

2nd iteration:

$$
\begin{array}{rcll}
s_5 & = & & Goto(s_2, -) \\
& = & \{ & [E \to T - \cdot E,\ \$],\ [E \to \cdot T - E,\ \$],\ [E \to \cdot T,\ \$], \\
& & & [T \to \cdot F * T,\ -],\ [T \to \cdot F,\ -], \\
& & & [T \to \cdot F * T,\ \$],\ [T \to \cdot F,\ \$], \\
& & & [F \to \cdot n,\ *],\ [F \to \cdot n,\ -],\ [F \to \cdot n,\ \$] \quad \} \\
s_6 & = & & Goto(s_3, *) \\
& = & \{ & [T \to F * \cdot T,\ \$],\ [T \to F * \cdot T,\ -], \\
& & & [T \to \cdot F * T,\ \$],\ [T \to \cdot F * T,\ -], \\
& & & [T \to \cdot F,\ \$],\ [T \to \cdot F,\ -], \\
& & & [F \to \cdot n,\ \$],\ [F \to \cdot n,\ -],\ [F \to \cdot n,\ *] \quad \}
\end{array}
$$

3rd iteration:

$$
\begin{array}{rcll}
s_7 & = & & Goto(s_5, E) \\
& = & \{ & [E \to T - E \cdot,\ \$] \quad \} \\
s_8 & = & & Goto(s_6, T) \\
& = & \{ & [T \to F * T \cdot,\ \$],\ [T \to F * T \cdot,\ -] \quad \}
\end{array}
$$

LR Parsing

Transition relation Δ

State	E	T	F	$-$	$*$	n
0	1	2	3			4
1						
2				5		
3					6	
4						
5	7	2	3			4
6		8	3			4
7						
8						

LR Parsing

Generation of LR(1)-tables:

$\forall s_x \in \mathcal{S}$
  $\forall i \in s_x$
    if $i = [A \rightarrow \alpha \cdot a\beta,\ b], \Delta(s_x, a) = s_k,\ a \in T$ **then**
      $ACTION[x, a] :=$ **shift**(k);
    else if $i = [S' \rightarrow S \cdot,\ \$]$ **then**
      $ACTION[x, a] :=$ **accept**;
    else if $i = [A \rightarrow \beta \cdot,\ a]$ **then**
      $ACTION[x, a] :=$ **reduce**$(A \rightarrow \beta)$;
  $\forall A \in N$
    if $\Delta(s_x, A) = s_k$ **then**
      $GOTO[x, A] := k$;

LR Parsing

LR(1)-tables for the example grammar:

	ACTION				*GOTO*		
	n	$-$	$*$	$	E	T	F
0	**shift**(4)				1	2	3
1				**accept**			
2		**shift**(5)		**reduce**(3)			
3		**reduce**(5)	**shift**(6)	**reduce**(5)			
4		**reduce**(6)	**reduce**(6)	**reduce**(6)			
5	**shift**(4)				7	2	3
6	**shift**(4)					8	3
7				**reduce**(2)			
8		**reduce**(4)		**reduce**(4)			