Introduction to Language Theory and Compilation: Exercises

Session 6: Grammars revisited

ULB

Grammars: quick reminder

A grammar is described by four components $\langle V, T, P, S \rangle$ where:

- V is the set of variables
- T is the set of terminals
- P is the set of production rules

$$P \subseteq (V \cup T)^* V (V \cup T)^* \times (V \cup T)^*$$

- $S \in V$ is the start symbol

Useless symbols

- A symbol $X \in (V \cup T)$ is said to be useless if there doesn't exist a derivation of the form:

$$\underbrace{S \overset{*}{\Rightarrow}}_{\text{accessible}} wXy \underbrace{\overset{*}{\Rightarrow} wxy}_{\text{productive}}$$

- To remove useless symbols:
  - We remove unproductive symbols (i.e. from which strings of terminals cannot be derived)
  - We remove inaccessible symbols
- In that order!

Useless symbols – examples

- Grammar whose language is empty

$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow aA \\ &\quad\ bC \\ C &\rightarrow Ab \end{aligned}$$

- Grammar with an *inaccessible* variable (C)

$$\begin{aligned} S &\rightarrow aAb \\ A &\rightarrow aA \\ &\quad\ b \\ C &\rightarrow bA \end{aligned}$$

Useless symbols – algorithms

Grammar `RemoveUnproductive(`**Grammar** $G = \langle V, T, P, S \rangle$`)` **begin**

$V_0 \leftarrow \emptyset$;

$i \leftarrow 0$;

repeat

$i \leftarrow i + 1$;

$V_i \leftarrow \{A \mid A \rightarrow \alpha \in P \wedge \alpha \in (V_{i-1} \cup T)^*\} \cup V_{i-1}$;

until $V_i = V_{i-1}$;

$V' \leftarrow V_i$;

$P' \leftarrow$ set of rules of P that do not contain variables in $V \setminus V'$;

`return(`$G' = \langle V', T, P', S \rangle$`)` ;

Useless symbols – algorithms (ctd.)

Grammar `RemoveInaccessible(`**Grammar** $G = \langle V, T, P, S \rangle$`)` **begin**

$V_0 \leftarrow \{S\}$; $i \leftarrow 0$;

repeat

$i \leftarrow i + 1$;

$V_i \leftarrow \{X \mid \exists\, A \rightarrow \alpha X \beta \text{ in } P \wedge A \in V_{i-1}\} \cup V_{i-1}$;

until $V_i = V_{i-1}$;

$V' \leftarrow V_i \cap V$; $T' \leftarrow V_i \cap T$;

$P' \leftarrow$ set of rules of P that only contain variables from V_i ;

`return(`$G' = \langle V', T', P', S \rangle$`)` ;

Useless symbols – algorithms (ctd.)

Grammar `RemoveUseless(`**Grammar** $G = \langle V, T, P, S \rangle$`)` **begin**

    Grammar $G_1 \leftarrow$ `RemoveUnproductive(`G`)` ;
    Grammar $G_2 \leftarrow$ `RemoveInaccessible(`G_1`)` ;
    `return(`G_2`)` ;

Exercise 1

Remove the useless symbols in the following two grammars:

a

$$
\begin{array}{lcl}
S & \rightarrow & a \mid A \\
A & \rightarrow & AB \\
B & \rightarrow & b
\end{array}
$$

b

$$
\begin{array}{lcl}
S & \rightarrow & A \\
 & & B \\
A & \rightarrow & aB \\
 & & bS \\
 & & b \\
B & \rightarrow & AB \\
 & & Ba \\
C & \rightarrow & AS \\
 & & b
\end{array}
$$

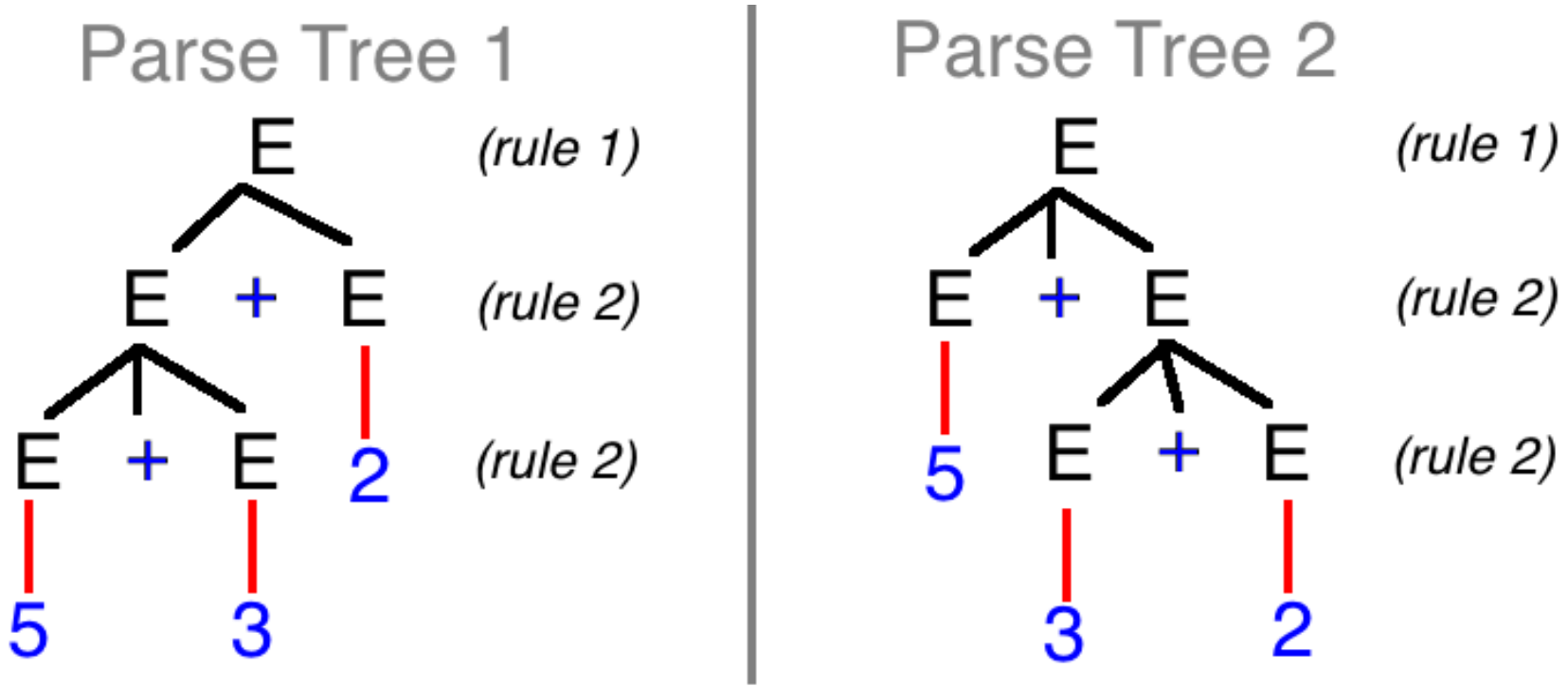
CFG transformations

- A grammar G is said to be ambiguous if there exists a word $w \in L(G)$ such that there exists at least two different parse trees for w.
- Example of an ambiguous grammar:

$$\begin{array}{rcl} E & \rightarrow & E + E \\ & & E * E \\ & & (E) \\ & & \textit{integer} \end{array}$$

- In the above example, the word $5 + 3 + 2$ has more than one possible parse tree.

CFG transformations (ctd.)



red edges illustrate the rule 4

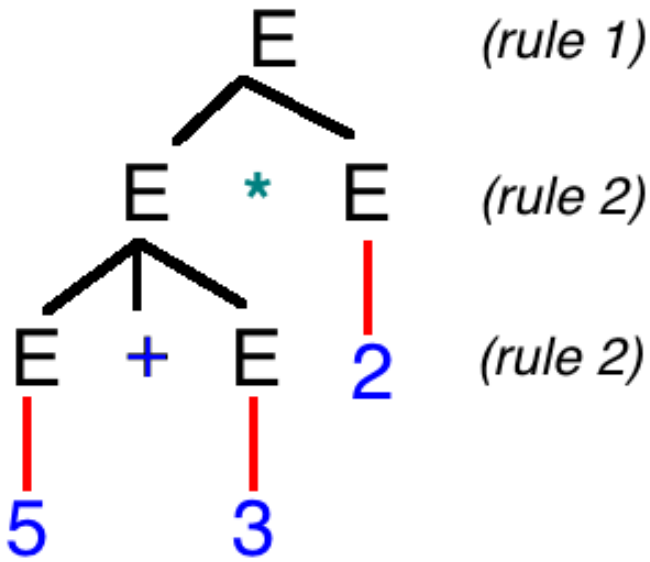
blue symbols are terminals

CFG transformations (ctd.)

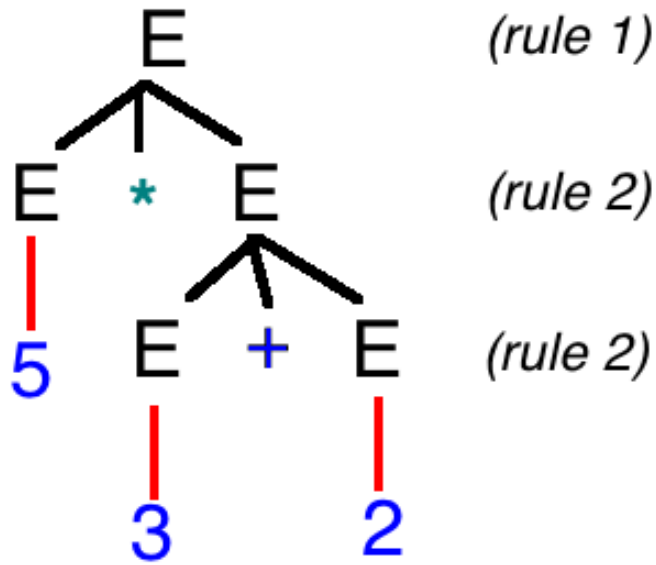
- Operator associativity: in the preceding example, the ambiguity arises from the fact that $+$ can be interpreted as left or right associative.
- Operator precedence: it can also be observed on the previous grammar that $*$ did not have precedence over $+$.

CFG transformations (ctd.)

Parse Tree 1



Parse Tree 2



red edges illustrate the rule 4

blue symbols are terminals

What happens if the first + change to a *?

$5 + 3 = 8$

$8 * 2 = 16$

$3 + 2 = 5$

$5 * 5 = 25$

Ambiguity removal

We can solve the associativity problem by transforming G into the following G' which forces left associativity:

$$\begin{array}{lcl} E & \rightarrow & E + T \\ & & E * T \\ & & T \\ T & \rightarrow & (E) \\ & & nb \end{array}$$

Ambiguity removal (ctd.)

We can then force precedence for $*$ over $+$ by transforming G' into the following G'':

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & & T \\
T & \rightarrow & T * F \\
 & & F \\
F & \rightarrow & (E) \\
 & & nb
\end{array}
$$

Exercise 2

Consider the following grammar:

$$
\begin{array}{lcl}
E & \rightarrow & E\ op\ E \\
 & & ID[E] \\
 & & ID \\
op & \rightarrow & * \\
 & & / \\
 & & + \\
 & & - \\
 & & ->
\end{array}
$$

- Show that the grammar is ambiguous.
- The desired operator precedence is the following:
  $\{[], ->\} > \{*, /\} > \{+, -\}$.
  Transform the grammar so that it accounts for operator precedence and left associativity.

Left factoring

- Left factoring aims to remove rules that share a common prefix since they make life difficult for predictive parsing.
- Sometimes, left factoring is enough to turn a given CFG into an LL(1) grammar!
- Example :

$$S \rightarrow ab \mid aa$$

is not LL(1). After left factoring, we get:

$$\begin{array}{lcl} S & \rightarrow & aN \\ N & \rightarrow & a \mid b \end{array}$$

which is LL(1)!

Left factoring – algorithm

`LeftFactor(`**Grammar** $G = \langle V, T, P, S \rangle$`)` **begin**

- **while** *G has at least two rules with the same left-hand side and a common prefix* **do**
  - Let $E = \{A \to \alpha\beta, \ldots, A \to \alpha\zeta\}$ be such a set of rules ;
  - Let $\mathcal{V}$ be a new variable;
  - $V = V \cup \mathcal{V}$;
  - $P = P \setminus E$;
  - $P = P \cup \{A \to \alpha\mathcal{V}, \mathcal{V} \to \beta, \ldots, \mathcal{V} \to \zeta\}$;

Exercise 3

Left-factor the following production rules:

<stmt> → **if** <expr> **then** <stmt-list> **end if**

<stmt> → **if** <expr> **then** <stmt-list> **else** <stmt-list> **end if**

Left recursion

- For the same reasons, it is often useful to suppress left (or right) recursion in a given grammar.
- The following grammar is left recursive:

$$S \to S\alpha \mid \beta$$

- It can be transformed into the equivalent (but not left recursive) grammar:

$$\begin{aligned} S &\to \mathcal{V}\mathcal{T} \\ \mathcal{V} &\to \beta \\ \mathcal{T} &\to \alpha\mathcal{T} \mid \varepsilon \end{aligned}$$

Left recursion removal algorithm

`RemoveLeftRecursion(`**Grammar** $G = \langle V, T, P, S \rangle$`)` **begin**

- **while** *G contains a left recursive variable A* **do**
  - Let $E = \{A \to A\alpha, A \to \beta, \ldots, A \to \zeta\}$ be the set of rules that have A as left-hand side ;
  - Let $\mathcal{U}$ and $\mathcal{V}$ be two new variables ;
  - $V = V \cup \{\mathcal{U}, \mathcal{V}\}$;
  - $P = P \setminus E$;
  - $P = P \cup \{A \to \mathcal{U}\mathcal{V}, \mathcal{U} \to \beta, \ldots, \mathcal{U} \to \zeta, \mathcal{V} \to \alpha\mathcal{V}, \mathcal{V} \to \varepsilon\}$;

Exercise 4

Apply the left recursion removal algorithm to the following grammar:

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & & T \\
T & \rightarrow & T * P \\
 & & P \\
P & \rightarrow & ID
\end{array}
$$

Exercise 5 – former exam question

Transform the following grammar into an LL(1) grammar:

$$
\begin{array}{lcl}
S & \rightarrow & aE \mid bF \\
E & \rightarrow & bE \mid \epsilon \\
F & \rightarrow & aF \mid aG \mid aHD \\
G & \rightarrow & Gc \mid d \\
H & \rightarrow & Ca \\
C & \rightarrow & Hb \\
D & \rightarrow & ab
\end{array}
$$