

# Curs 8

## LR(k) parsing

# Terms

Reminder:

rhp = right handside of production

lhp = left handside of production

- Prediction – see LL(1)
- Handle = symbols from the head of the working stack that form (in order) a rhp
- ***Shift – reduce*** parser:
- **shift** symbols to form a handle
- When a rhp is formed – **reduce** to the corresponding lhp

# LR(k)

- L = left – sequence is read from left to right
- R = right – use rightmost derivations
- k = length of prediction
- Enhanced grammar
- $G = (N, \Sigma, P, S)$
- $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S'), S' \notin N$

$S'$  does NOT appear in any rhp

- **Definition 1:** If in a cfg  $G = (N, \Sigma, P, S)$  we have  
 $S \xRightarrow{*}_r \alpha A w \Rightarrow_r \alpha \beta w$ , where  $\alpha \in (N \cup \Sigma)^*$ ,  $A \in N$ ,  $w \in \Sigma^*$ , then  
any prefix of sequence  $\alpha\beta$  is called **live prefix** in  $G$ .
- **Definition 2:** **LR(k) item** is defined as  $[A \rightarrow \alpha.\beta, u]$ , where  $A \rightarrow \alpha\beta$  is a production,  $u \in \Sigma^k$  and describe the moment in which, considering the production  $A \rightarrow \alpha\beta$ ,  $\alpha$  was detected ( $\alpha$  is in head of stack) and it is expected to detect  $\beta$ .
- **Definition 3:** LR(k) item  $[A \rightarrow \alpha.\beta, u]$  is **valid for the live prefix**  $\gamma\alpha$  if:  

$$\begin{aligned} S &\xRightarrow{*}_r \gamma A w \Rightarrow_r \gamma \alpha \beta w \\ u &= \text{FIRST}_k(w) \end{aligned}$$

**Definition 4:** A cfg  $G = (N, \Sigma, P, S)$  is LR( $k$ ), for  $k \geq 0$ , if

1.  $S' \xRightarrow{*}_r \alpha A w \Rightarrow_r \alpha \beta w$
  2.  $S' \xRightarrow{*}_r \gamma B x \Rightarrow_r \alpha \beta y$
  3.  $\text{FIRST}_k(w) = \text{FIRST}_k(y)$
- }  $\Rightarrow \alpha = \gamma \text{ AND } A = B \text{ AND } x = y$

- $[A \rightarrow \alpha\beta., u]$  – rhp detected - apply reduce
- $[A \rightarrow \alpha.\beta, u]$  - shift

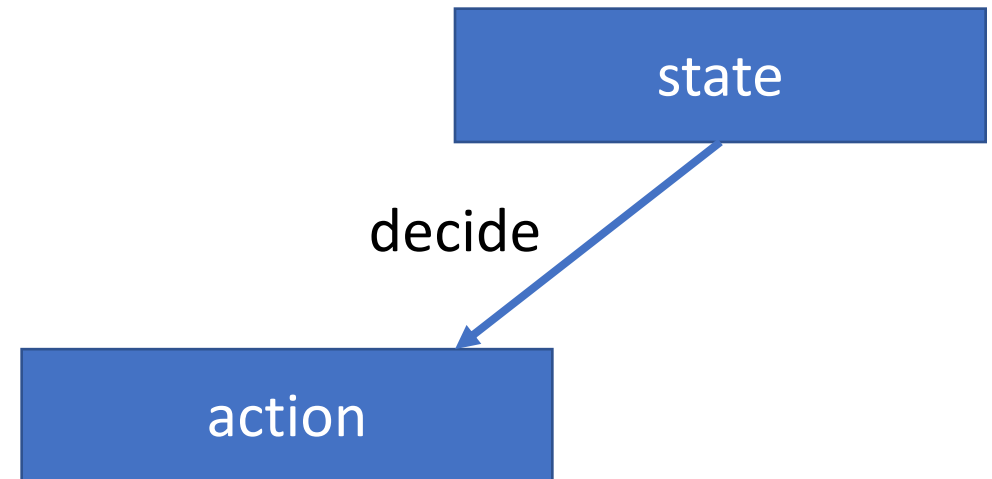
$\Rightarrow$  Working stack:

$\$s_{\text{init}}X_1s_1 \dots X_ms_m$

where: \$ - mark empty stack

$X_i \in N \cup \Sigma$

$s_i$  - states



# LR(k) principle

- Current state
- Current symbol
- prediction

uniquely determines:

- Action to be applied
- Move to a new state

=> LR(k) table – 2 parts: **action** part + **goto** part

# States

## What a state contains?

- LR items – all items corresponding to same live prefix
- *closure*

## How to go from one state to another state? How many states?

- *goto*
- *Canonical collection*



*What LR item will be in the same state?*

- $[A \rightarrow \alpha.B\beta, u]$  valid for live prefix  $\gamma\alpha \Rightarrow$

$$S \xRightarrow{*}_{dr} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w$$

$$u = FIRST_k(w)$$

- $B \rightarrow \delta \in P \Rightarrow S \xRightarrow{*}_{dr} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w \xRightarrow{*}_{dr} \gamma\alpha\delta w,$

$\Rightarrow [B \rightarrow .\delta, u]$  valid for live prefix  $\gamma\alpha$

# LR(k) parsing:

## LR(0), SLR, LR(1), LALR

- Define item
- Construct set of states
- Construct table

Executed 1 time

- 
- Parse sequence based on moves between configurations

# LR(0) Parser

- Prediction of length 0 (ignored)

1. LR(0) item:  $[A \rightarrow \alpha.\beta]$

## 2. Construct set of states

- What a state contains – Algorithm *closure\_LR(0)*
- How to move from a state to another – Function *goto\_LR(0)*
- Construct set of states – Algorithm *ColCan\_LR(0)*

Canonical collection

# Algorithm *Closure\_LR(0)*

---

**INPUT:** I-element de analiză;  $G'$ - gramatica îmbogățită

**OUTPUT:**  $C = \text{closure}(I)$ ;

$C := \{I\}$ ;

**repeat**

**for**  $\forall [A \rightarrow \alpha.B\beta] \in C$  **do**

**for**  $\forall B \rightarrow \gamma \in P$  **do**

**if**  $[B \rightarrow \cdot\gamma] \notin C$  **then**

$C = C \cup [B \rightarrow \cdot\gamma]$

**end if**

**end for**

**end for**

**until**  $C$  nu se mai modifică

---

# Function *goto*<sub>LR(0)</sub>

$$\text{goto} : P(\mathcal{E}_0) \times (N \cup \Sigma) \rightarrow P(\mathcal{E}_0)$$

where  $\mathcal{E}_0$  = set of LR(0) items

$$\text{goto}(s, X) = \text{closure}(\{[A \rightarrow \alpha X \beta] \mid [A \rightarrow \alpha \cdot X \beta] \in s\})$$

# Algorithm *ColCan\_LR(0)*

---

**INPUT:**  $G'$  - gramatica îmbogățită

**OUTPUT:**  $C$  - colecția canonică de stări

$C := \emptyset;$

$s_0 := closure(\{[S' \rightarrow .S]\})$

$C := C \cup \{s_0\};$

**repeat**

**for**  $\forall s \in C$  **do**

**for**  $\forall X \in N \cup \Sigma$  **do**

**if**  $goto(s, X) \neq \emptyset$  and  $goto(s, X) \notin C$  **then**

$C = C \cup goto(s, X)$

**end if**

**end for**

**end for**

**until**  $C$  nu se mai modifică

---

### 3. Construct LR(0) table

- one line for each state
- 2 parts:
  - Action: one column (for a state, action is unique because prediction is ignored)
  - Goto: one column for each symbol  $X \in N \cup \Sigma$



# Rules LR(0) table

1. *if  $[A \rightarrow \alpha.\beta] \in s_i$  then **action**( $s_i$ )=shift*
2. *if  $[A \rightarrow \beta.] \in s_i$  and  $A \neq S'$  then **action**( $s_i$ )=reduce  $l$ , where  $l$  = number of production  $A \rightarrow \beta$*
3. *if  $[S' \rightarrow S.] \in s_i$  then **action**( $s_i$ )=acc*
4. *if  $\text{goto}(s_i, X) = s_j$  then **goto**( $s_i, X$ ) =  $s_j$*
5. *otherwise = error*

# Remarks

- 1) Initial state of parser = state containing  $[S' \rightarrow .S]$
- 2) No shift from accept state:  
*if  $s$  is accept state then  $\text{goto}(s, X) = \emptyset, \forall X \in N \cup \Sigma$ .*
- 3) *If in state  $s$  action is reduce then  $\text{goto}(s, X) = \emptyset, \forall X \in N \cup \Sigma$ .*
- 4) Argument  $G'$ : Let  $G = (\{S\}, \{a, b, c\}, \{S \rightarrow aSbS, S \rightarrow c\}, S)$   
states  $[S \rightarrow aSbS.]$  and  $[S \rightarrow c.]$  – accept / reduce ?

# Remarks (cont)

5) A grammar is NOT LR(0) if the LR(0) table contains conflicts:

- shift – reduce conflict: a state contains items of the form  $[A \rightarrow \alpha.\beta]$  and  $[B \rightarrow \gamma.]$ , yielding to 2 distinct actions for that state
- reduce – reduce conflict: when a state contains items of the form  $[A \rightarrow \alpha\beta.]$  and  $[B \rightarrow \gamma.]$ , in which the action is reduce, but with distinct productions

## 4. Define configurations and moves

- INPUT:

- Grammar  $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$
- LR(0) table
- Input sequence  $w = a_1 \dots a_n$

- OUTPUT:

*if* ( $w \in L(G)$ )                      ***then* string of productions**  
***else* error & location of error**

# LR(0) configurations

$$(\alpha, \beta, \pi)$$

where:

- $\alpha$  = working stack
- $\beta$  = input stack
- $\pi$  = output (result) stack

Initial configuration:  
 $(\$s_0, w\$, \varepsilon)$

Final configuration:  
 $(\$s_{acc}, \$, \pi)$

# Moves

## 1. Shift

**if**  $\text{action}(s_m) = \text{shift}$  **AND**  $\text{head}(\beta) = a_i$  **AND**  $\text{goto}(s_m, a_i) = s_j$  **then**

$$(\$s_0x_1 \dots x_ms_m, a_i \dots a_n$, \pi) \vdash (\$s_0x_1 \dots x_ms_ma_is_j, a_{i+1} \dots a_n$, \pi)$$

## 2. Reduce

**if**  $\text{action}(s_m) = \text{reduce } l$  **AND**  $(l) A \rightarrow x_{m-p+1} \dots x_m$  **AND**  $\text{goto}(s_{m-p}, A) = s_j$  **then**

$$(\$s_0 \dots x_ms_m, a_i \dots a_n$, \pi) \vdash (\$s_0 \dots x_{m-p}s_{m-p}As_j, a_i \dots a_n$, l \pi)$$

## 3. Accept

**if**  $\text{action}(s_m) = \text{accept}$  **then**  $(\$s_m, \$, \pi) = \text{acc}$

## 4. Error - otherwise

# LR(0) Parsing Algorithm

## INPUT:

- LR(0) table – conflict free
- grammar  $G'$ : production numbered
- - sequence = Input sequence  $w = a_1 \dots a_n$

## • OUTPUT:

*if* ( $w \in L(G)$ )            ***then* string of productions**  
***else* error & location of error**

# LR(0) Parsing Algorithm

```
state := 0;
alpha := '$s0'; beta := 'w$'; phi := ""; end := false
Repeat
    if action(state) = 'shift' then
        a = beta.pop();
        state = goto(state, a)
        alpha.push(a, state);
    else
```

```
        if action(state) = 'reduce l' then
            search_prod(l, rhp, lhp);
            alpha.pop(rhp);
            state = goto(head(alpha.state), lhp);
            alpha.push(lhp, state);
            phi.push(l);
```

**else**

```
    if action(state) = 'accept' then
        write(" success", out);
        end := true;
    if action(state) = 'error' then
        write(" error")
        end := true
```

**Until** end