Course 8

LR(k) parsing

Important notices

- Monday 29.11.2021 – course from 7.30 am (if NOT a free day)

- No course on 13.12.2021 – so we will have course

  on 06.12.2021 from 11.10

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

- <u>Enhanced grammar</u>

- $G = (N, \Sigma, P, S)$
- $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$, $S' \notin N$

S' does NOT appear in any rhp

LR(k)

- Ascendent

- Linear – COST? – what we compute to obtain linear algorithm?

- **Definition 1**: If in a cfg $G = (N, \Sigma, P, S)$ we have

  $S \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow_r \alpha\beta w$, 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 $A \rightarrow \alpha\beta$, α was detected (α is in head of stack) and it is expected to detect β.

- **Definition 3**: LR(k) item $[A \rightarrow \alpha.\beta,u]$ is ***valid for the live prefix*** $\gamma\alpha$ if:

  $S \overset{*}{\Rightarrow}_r \gamma A w \Rightarrow_r \gamma\alpha\beta w$

  $u = FIRST_k(w)$

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

$$\left.\begin{array}{ll} 1. & S' \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow_r \alpha\beta w \\ 2. & S' \overset{*}{\Rightarrow}_r \gamma B x \Rightarrow_r \alpha\beta y \\ 3. & FIRST_k(w) = FIRST_k(y) \end{array}\right\} \Rightarrow \alpha = \gamma \textit{ AND } A = B \textit{ AND } x = y$$

- $[A \rightarrow \alpha\beta.,u]$ – special case: prefix is all rhp - apply reduce

- Otherwise $[A \rightarrow \alpha.\beta,u]$ – apply shift

Consequence 1: state is important –

should be stored by parsing method

$\Rightarrow$Working stack:

$\$s_{init}X_1s_1 \ldots X_ms_m$

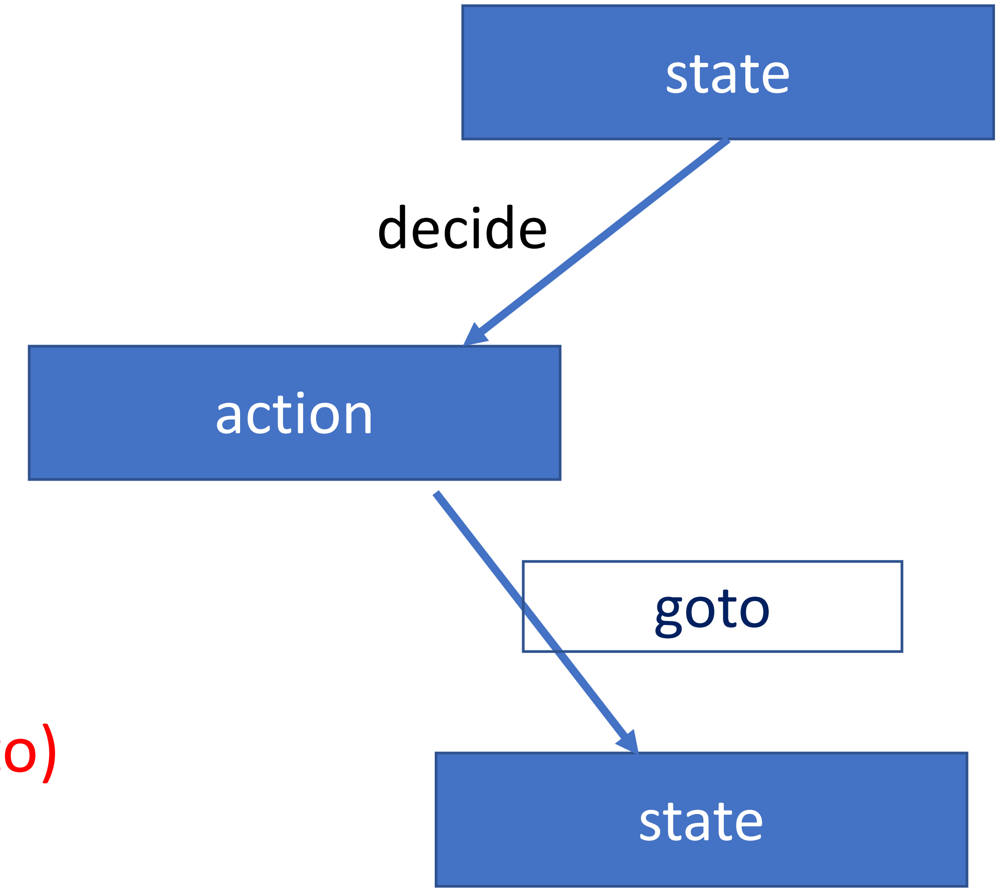
where: \$ - mark empty stack

$X_i \in N \cup \Sigma$

s_i - states

Consequence 2: the action takes the

parsing process to another state (goto)



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?

- $[A \to \alpha.B\beta,u]$ valid for live prefix $\gamma\alpha$ =>

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

$$u = FIRST_k(w)$$

- $B \to \delta \in P$ => $S \overset{*}{\Rightarrow}_{dr} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w \overset{*}{\Rightarrow}_{dr} \boldsymbol{\gamma\alpha\delta} w'$

=> $[B \to .\delta,u]$ 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 îmbogăţită
OUTPUT: C = closure(I);
$C := \{I\};$
repeat
  for $\forall[A \to \alpha.\underline{B}\beta] \in C$ **do**
    for $\underline{\forall B \to \gamma} \in P$ **do**
      if $[B \to .\gamma] \notin C$ **then**
        $C = C \cup [B \to .\gamma]$
      end if
    end for
  end for
until C nu se mai modifică

Function *goto_LR(0)*

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

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

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

Algorithm *ColCan_LR(0)*

INPUT: G'- gramatica îmbogăţită
OUTPUT: C - colecţia canonică de stări

$\mathcal{C} := \emptyset;$
$s_0 := closure(\{[S' \to .S]\})$
$\mathcal{C} := \mathcal{C} \cup \{s_0\};$
repeat
    for $\forall s \in \mathcal{C}$ **do**
        for $\forall X \in N \cup \Sigma$ **do**
            if $goto(s, X) \neq \emptyset$ and $goto(s, X) \notin \mathcal{C}$ **then**
                $\mathcal{C} = \mathcal{C} \cup goto(s, X)$
            end if
        end for
    end for
until $\mathcal{C}$ 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* goto(s_i, X) = s_j *then* **goto(s_i, X) = s_j**
5. otherwise = **error**

	action	goto				
		S	A	...	a	b ...
s_0						
s_1						
⋮						
s_n						

Remarks

1) Initial state of parser = state containing [S' → .S]
2) No shift from accept state:

   if s is accept state *then* goto(s, X) = ∅, ∀X ∈ N ∪ Σ.

3) *If* in state ***s*** action is reduce *then* goto(s, X) = ∅, ∀X ∈ N ∪ Σ.
4) Argument G': Let G = ({S},{a,b,c},{S → aSbS,S → c},S)

   states [S → aSbS.] and [S → 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' = (NU{S'}, $\boldsymbol{\Sigma}$, P U {S'->S},S')
  - LR(0) table
  - Input sequence w =$a_1 \ldots a_n$
- OUTPUT:

  if (w ∈L(G)) *then* **string of productions**

  else **error & location of error**

LR(0) configurations

$$(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\pi})$$

where:

- α = working stack
- β = input stack
- π = output (result) stack

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

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

Moves

1. Shift

if action(s_m)= shift AND head(β)=a_i AND goto(s_m,a_i)=s_j **then**

$$(\$s_0x_1 \ldots x_m s_m, a_i \ldots a_n\$, \pi) \vdash (\$s_0x_1 \ldots x_m s_m a_i s_j, a_{i+1} \ldots a_n\$, \pi)$$

2. Reduce

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

$$(\$s_0 \ldots x_m s_m, a_i \ldots a_n\$, \pi) \vdash (\$s_0 \ldots x_{m-p} s_{m-p} A s_j, a_i \ldots a_n\$, l\pi)$$

3. Accept

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

4. Error - otherwise

LR(0) Parsing Algorithm

INPUT:

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

- 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
Config := (alpha,beta,phi);
Repeat
    if action(state)='shift' then
        ActionShift(config)
    else
        if action(state) ='reduce l" then
            ActionReduce(config)
        else
            if action(state)='accept' then
                write(" success", phi);
                end := true;
            if action(state) = 'error' then
                write(" error")
                end := true
Until end
```