Introduction to Language Theory and Compilation: Exercises

Session 9: LR(0) and LR(k) parsing

ULB

Introduction

The idea behind LR parsing is the same as for bottom-up parsing:

- Reduce a string of terminals and variables (pushed on a stack at an earlier stage) into a variable.
  - We'll read the production rules "in reverse".
  - The right hand side of a rule, which will be used to reduce, is called a *handle*.

Example

$$
\begin{aligned}
S' &\rightarrow S\$ \\
S &\rightarrow Saa \\
S &\rightarrow a \\
S &\rightarrow \varepsilon
\end{aligned}
$$

Stack	*Input*	*Action*	*Output*
$\vdash$	$aa\$$	R3	
$\vdash S$	$aa\$$	S	3
$\vdash Sa$	$a\$$	S	3
$\vdash Saa$	$\$$	R1	3
$\vdash S$	$\$$	S	3,1
$\vdash S\$$	ε	Accept	3,1

We've seen that choosing between shifting and reducing isn't easy. . .

LR(k) grammars

Let $G = \langle V, T, P, S\rangle$ be a grammar. Consider its augmented version $G' = \langle V', T, P', S'\rangle$. G' is said to be LR(k) for $k \geqslant 0$ if the following three conditions:

1. $S' \overset{*}{\Rightarrow}_{G'} \gamma A x \Rightarrow_{G'} \gamma\alpha x$
2. $S' \overset{*}{\Rightarrow}_{G'} \delta B y \Rightarrow_{G'} \gamma\alpha x'$
3. $\mathrm{First}^k(x) = \mathrm{First}^k(x')$

imply that $\gamma A x' = \delta B y$ (in other words, $\gamma = \delta$, $A = B$ and $x' = y$).

Canonical finite state machine (CFSM)

- We can build a canonical finite state machine (CFSM) that reflects the decisions made by an LR parser.
- Each state contains several *items*, which are production rules where we add • that represent how far we've come in the parsing process.
  - Part of these items form the *kernel*.
  - The other items are obtained by *closure*.
- The state machine will allow us to build the action tables needed by the parser.

ULB

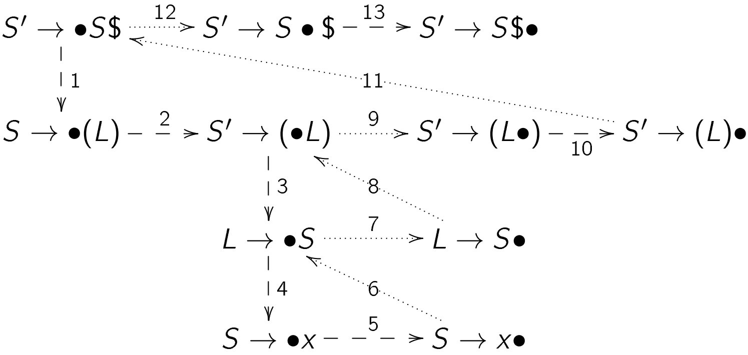
Example

Consider the following augmented grammar:

$$
\begin{array}{cccl}
(0) & S' & \rightarrow & S\$ \\
(1) & S & \rightarrow & (L) \\
(2) & S & \rightarrow & x \\
(3) & L & \rightarrow & S \\
(4) & L & \rightarrow & L, S
\end{array}
$$

Stolen from: *"Modern compiler implementation in Java"*, A. W. Appel

Example – recognise (x)



The $\bullet$ represents *how far* the parser has come.

Example (ctd.)

- We want to recognize a word that can be derived from S'

Example (ctd.)

$S' \to \bullet S\$$

- We want to recognize a word that can be derived from S'
- We must thus consume $S\$\ldots$

Example (ctd.)

$S' \rightarrow \bullet S\$$

Kernel

The $\bullet$ represents how far parsing has come.

- We want to recognize a word that can be derived from S'
- We must thus consume $S\$\ldots$

ULB

Example (ctd.)

$S' \rightarrow \bullet S\$$

Kernel

The $\bullet$ represents how far parsing has come.

- We want to recognize a word that can be derived from S'
- We must thus consume $S\$\ldots$
- But S isn't a *terminal!*

Example (ctd.)

$S' \to \bullet S\$$

Closure

The $\bullet$ represents how far parsing has come.

- We want to recognize a word that can be derived from S'
- We must thus consume $S\$$...
- But S isn't a *terminal!*

Example (ctd.)

$S' \rightarrow \bullet S\$$
$S \rightarrow \bullet(L)$
$S \rightarrow \bullet x$

Closure

The $\bullet$ represents how far parsing has come.

- We want to recognize a word that can be derived from S'
- We must thus consume $S\$$...
- But S isn't a *terminal!*
- To recognize S, we have to start by consuming (or x.

ULB

Example (ctd.)

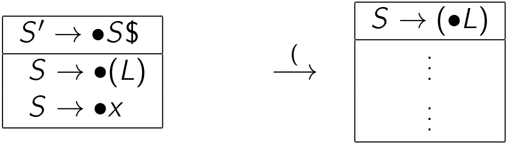
Transitions

$S' \rightarrow \bullet S\$$
$S \rightarrow \bullet (L)$ $S \rightarrow \bullet x$

ULB

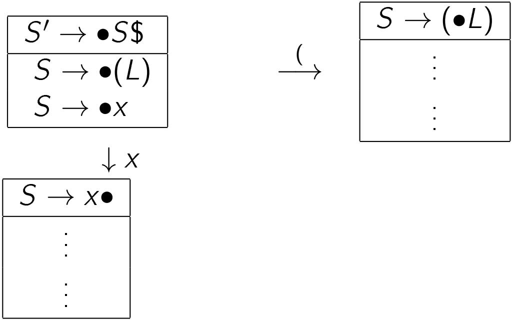
Example (ctd.)

Transitions

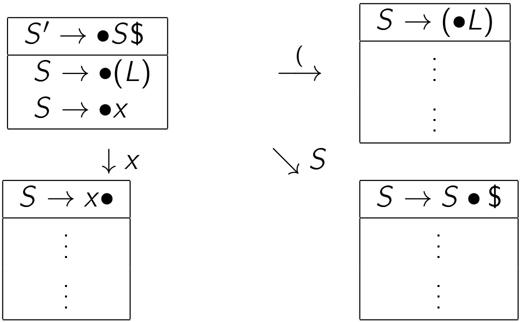


Example (ctd.)

Transitions



Transitions



- We want to recognize a word that can be derived from L

Example (ctd.)

$S \rightarrow (\bullet L)$
$L \rightarrow \bullet S$ $L \rightarrow \bullet L, S$

Closure (1)

- We want to recognize a word that can be derived from L
- Thus, we must consume L or $S\$\ldots$

ULB

Example (ctd.)

$S \to (\bullet L)$
$L \to \bullet S$
$L \to \bullet L, S$
$S \to \bullet(L)$
$S \to \bullet x$

Closure (2)

- We want to recognize a word that can be derived from L
- Thus, we must consume L or $S\$\ldots$
- We thus do another closure step!

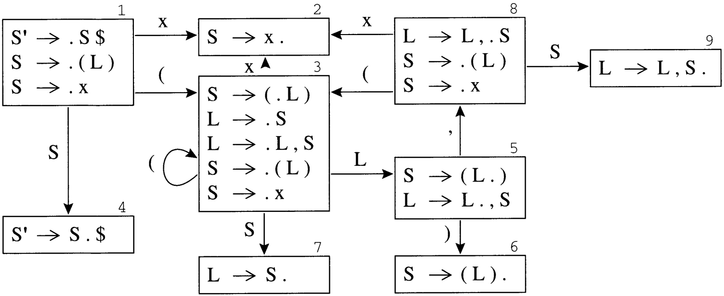
ULB

Example (ctd.)

$S \rightarrow x\bullet$
⋮ ⋮

- In this state, nothing needs to be added by closure.
- If we get here, it means we have recognized S.
- The parser can thus proceed with a *Reduce* action.

Example (ctd.)


1
S' → . S $
S → . (L)
S → . x
2
S → x .
3
S → (. L)
L → . S
L → . L , S
S → . (L)
S → . x
4
S' → S . $
5
S → (L .)
L → L . , S
6
S → (L) .
7
L → S .
8
L → L , . S
S → . (L)
S → . x
9
L → L , S .
x
(
S
L
)
,

Example (ctd.)

State	*Action*
1	Shift
2	Reduce
3	Shift
4	Accept
5	Shift

State	*Action*
6	Reduce
7	Reduce
8	Shift
9	Reduce

LR(0) CFSM – algorithms

```
Closure(I) begin
    repeat
        I' ← I ;
        foreach item [A → α • Bβ] ∈ I, B → γ ∈ G' do
            I ← I ∪ [B → •γ] ;
    until I' = I ;
    return(I) ;
end

Transition(I,X) begin
    return(Closure({[A → αX • β] | [A → α • Xβ] ∈ I})) ;
end
```

ULB

LR(0) CFSM – algorithms

```
Items(G') begin
    C ← Closure({[S' → •S$]}) ;
    repeat
        C' ← C;
        foreach I ∈ C, X ∈ T' ∪ V' do
            C ← C ∪ Transition(I,X) ;
    until C' = C ;
end
```

LR(0) parser – algorithms

To build the action table, we use the following process:

foreach *state s of the CFSM* **do**
  if *s contains* $A \rightarrow \alpha \bullet a\beta$ **then**
    $\text{Action}[s] \leftarrow \text{Action}[s] \cup \text{Shift}$;
  else if *s contains* $A \rightarrow \alpha\bullet$ *that is the* i^{th} *rule* **then**
    $\text{Action}[s] \leftarrow \text{Action}[s] \cup \text{Reduce}_i$;
  else if *s contains* $S' \rightarrow S\$\bullet$ **then**
    $\text{Action}[s] \leftarrow \text{Action}[s] \cup \text{Accept}$;

Exercise 1

$$
\begin{array}{llll}
(0) & S' \rightarrow S\$ & (5) & C \rightarrow Fg \\
(1) & S \rightarrow aCd & (6) & C \rightarrow CF \\
(2) & S \rightarrow bD & (7) & F \rightarrow z \\
(3) & S \rightarrow Cf & (8) & D \rightarrow y \\
(4) & C \rightarrow eD & &
\end{array}
$$

- Give the corresponding LR(0) CFSM and its action table.

LR(0) parser – algorithm

- The parser uses a *stack* on which it *pushes* symbols as well as the current state number.
- This allows it to return to the right state upon *reductions*.
- The consumed *string* is accepted if we reach the final state (whose sole action is to accept).
- We represent an LR(0) parser's configuration with a triplet : $\langle$stack, input, output$\rangle$.
- Initially, we have $\langle \vdash 0, \omega, \varepsilon \bot \rangle$

LR(0) parser – transitions

begin

Considering we have $\langle \vdash \gamma s, ax, y\perp \rangle$:

if $\mathrm{Action}[s] = \mathrm{Shift}$ **then**

goto $\langle \vdash \gamma s a \mathrm{Successor}[s, a], x, y\perp \rangle$;

else if $\mathrm{Action}[s] = \mathrm{Reduce}_j$ *par* $A \rightarrow \alpha$ **then**

Having $\langle \vdash \gamma s' x_1 s_1 x_2 s_2 \ldots x_n s, x, y\perp \rangle$ and $\alpha = x_1 x_2 \ldots x_n$:

goto $\langle \vdash \gamma s' A \mathrm{Successor}[s', A], x, jy\perp \rangle$;

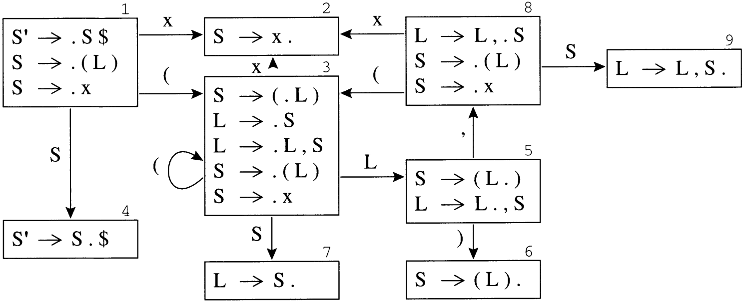
else if $\mathrm{Action}[s] = \mathrm{Accept}$ **then**

`return(` OK `)` ;

else `return(`*Error*`)` ;

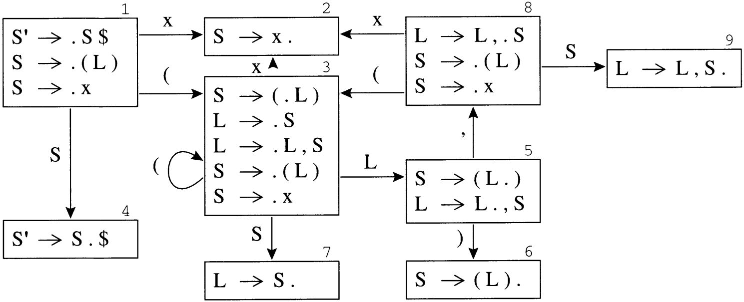
end

Example – recognizing (x)



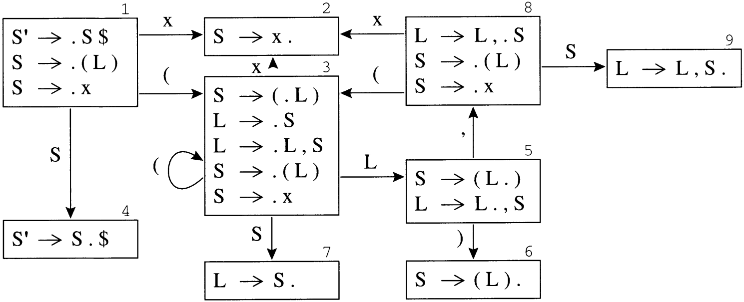
Config.: $\langle$ 1, (x)$, $\rangle$ Action: Shift

Example – recognizing (x)



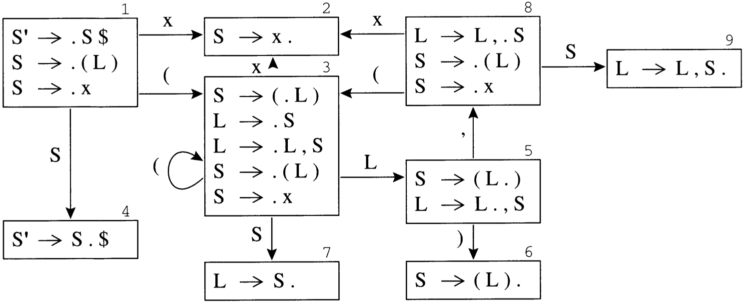
Config.: ⟨ 1(3, x)$, ⟩ Action: Shift

Example – recognizing (x)



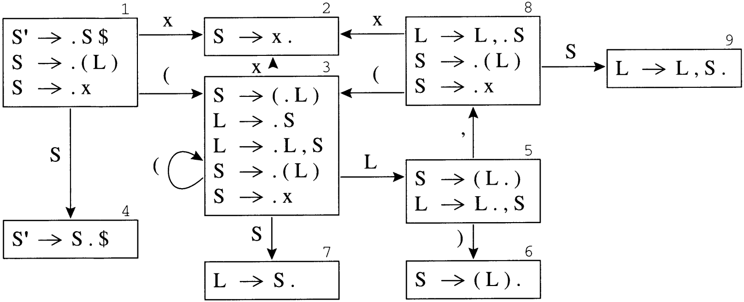
Config.: $\langle$ 1(3x2,)\$, $\rangle$ Action: Reduce 2

Example – recognizing (x)



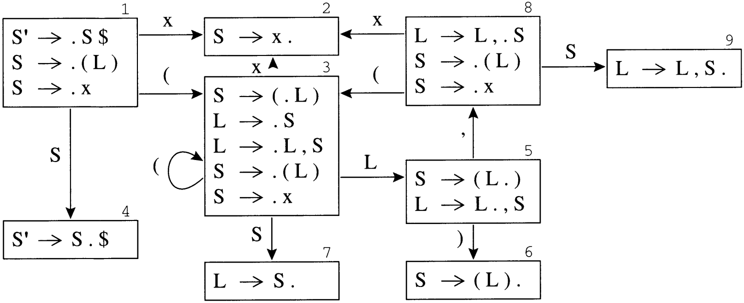
Config.: $\langle$ 1(3*S*7,)\$, 2 $\rangle$ Action: Reduce 3

Example – recognizing (x)



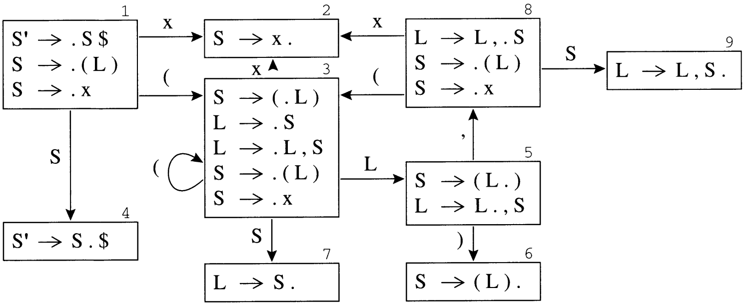
Config.: $\langle\ 1(3L5, \quad)\$, \quad 2\ 3\ \rangle$ Action: Shift

Example – recognizing (x)



Config.: $\langle$ 1(3L5)6,)\$, 2 3 $\rangle$ Action: Reduce 1

Example – recognizing (x)



Config.: $\langle$ 1S4, $, 2 3 1 $\rangle$ Action: Accept

Exercise 2

Simulate the parser you built during the previous exercise on the following string : `aeyzzd`

Introducing LR(k) grammars

- Difference with LR(0) : we must now account for the lookahead symbols.
- For example:
  - Consider the case where a CFSM state contains both $A \to \alpha_1 \bullet \alpha_2$ and $B \to \gamma\bullet$
  - We have a shift-reduce conflict.
  - If we do not have the characters of $\mathrm{First}^k(\alpha_2)$ on input, we know we should not attempt shifting.
  - In which *context* can we be sure we'll never make a mistake?

ULB

Introducing LR(k) grammars

- We have to remember a context.
- The items of the CFSM will now have the following shape:

$$[A \rightarrow \alpha_1 \bullet \alpha_2, u]$$

  - u represents the context, i.e. the set of strings of k terminals that can follow productions of $A \rightarrow \alpha_1\alpha_2$.
  - We start off with $[S' \rightarrow \bullet S\$, \varepsilon]$
- We have to adapt our algorithms, action tables, etc.

ULB

```
Closure(I) begin
    repeat
        I' ← I ;
        foreach item [A → α • Bβ,σ] ∈ I, B → γ ∈ G' do
            foreach u ∈ First^k(βσ) do
                I ← I ∪ [B → •γ,u];
    until I' = I ;
    return(I) ;
end
Transition(I,X) begin
    return(Closure({[A → αX • β,u] | [A → α • Xβ,u] ∈ I})) ;
end
```

We build the action table as follows:

foreach *state s of the CFSM* **do**
- **if** s *contains* $[A \rightarrow \alpha \bullet a\beta, u]$ **then**
  - **foreach** $u \in First^k(a\beta u)$ **do**
    - $\mathrm{Action}[s,u] \leftarrow \mathrm{Action}[s,u] \cup \mathrm{Shift}$;
- **else if** s *contains* $[A \rightarrow \alpha\bullet,\ u\ ]$, *that is the* i^{th} *rule* **then**
  - $\mathrm{Action}[s,u\ ] \leftarrow \mathrm{Action}[s,u] \cup \mathrm{Reduce}_i$;
- **else if** s *contains* $[S' \rightarrow S\$\bullet,\varepsilon]$ **then**
  - $\mathrm{Action}[s,\cdot] \leftarrow \mathrm{Action}[s,\cdot] \cup \mathrm{Accept}$;

Exercise 3

Build the LR(1) parser for the following grammar:

$$
\begin{array}{ll}
(1) & S' \rightarrow S\$ \\
(2) & S \rightarrow A \\
(3) & A \rightarrow bB \\
(4) & A \rightarrow a \\
(5) & B \rightarrow cC \\
(6) & B \rightarrow cCe \\
(7) & C \rightarrow dAf
\end{array}
$$

Is the grammar LR(0)? Explain.

Exercise 4

Build the LR(1) parser for the following grammar:

$$
\begin{array}{ll}
(1) & S' \rightarrow S\$ \\
(2) & S \rightarrow SaSb \\
(3) & S \rightarrow c \\
(4) & S \rightarrow \varepsilon
\end{array}
$$

Simulate it on the following input: `abacb`.