Introduction to Language Theory and Compilation
Exercises
Session 9: LR(0) and LR(k) parsing

Reminders

A LR-parser is a bottom-up parser unlike a LL-parser which is a top-down parser. The R means rightmost deviation unlike the second L in a LL-parser. The similarities between LR and LL are that the first L stands for the reading order (left to right) and they use k lookahead tokens in order to avoid backtracking. The notation is $LR(k)$.

Canonical finite state machine (CFSM)

A **CFSM** expresses the decisions made by an LR-parser. As shown in Figure 1, each state contains three kind of items:

State *ID* The unique identifier of the current state.

Kernel The current rule(s) that the parser is using.

Closure The rules derived from the kernel.

State *ID*
Kernel
Closure

Figure 1: Generic state

For instance, from the grammar in Figure 2.1, the state 1 will be the Figure 2.2. The kernel is the start variable $S\prime$ where the marker $\bullet$ is put before the production. This marker specify how far we have come in the parsing process. Because the state 1 has to read the variable (non-terminal symbol) S, the closure operation add some rules as items in state 1. These rules are all productions of S (because it is the symbol we have to read) where the $\bullet$ will be in the first position. The parser still have to read the terminals '**(**' and '**x**' from the closure and the non-terminal '**S**' from the kernel.
By reading the '**(**' from state 1, the parser arrives in state 2 (Figure 2.3) and the kernel consists of all rules from state 1 for which the parser should read a '**(**' (in the complete version of the state machine, a transition from state 1 to 2 has the label '**(**'). The closure adds all productions of L but because L produces another non-terminal S, the closure also adds all productions of S. The parser still have to read the terminals '**(**', '**x**' and the non-terminals '**S**', '**L**' from the closure and the non-terminal '**L**' from the kernel. Note that if the parser reads a '**(**' from state 2, it goes into state 2.
By reading the '**x**' from state 1, the parser arrives in state 3 (Figure 2.4) and the kernel is empty because all rules from state 1 does not contain a S to read.
By reading the '**L**' from state 2, the parser arrives in state 4 (Figure 2.5) and the marker is put after the L. All other states are produced in a similar fashion.

(0)	$S\prime \to S\$$
(1)	$S \to (L)$
(2)	$S \to x$
(3)	$L \to S$
(4)	$L \to L,S$

1 Initial grammar

State 1
$S\prime \to \bullet S\$$
$S \to \bullet(L)$ $S \to \bullet x$

2 Initial state

State 2
$S \to (\bullet L)$
$L \to \bullet S$ $L \to \bullet L,S$ $S \to \bullet(L)$ $S \to \bullet x$

3 when a (is read from state 1 or itself

State 3
$S \to x\bullet$

4 when a **x** is read from state 1 or 2

State 2
$S \to (L\bullet)$ $L \to L\bullet,S$

5 when the **L** is read from state 4

Figure 2: Example of the construction of a canonical finite state machine

Action table

Remember that the three operations are **accept** when the language is accepted by the parser, **shift** when the parser reads one more token on the input and **reduce** when the parser replaces γ by A on the stack where γ is the top symbols on the stack and there exists a rule of the form $A \to \gamma$.

```
ActionTable() begin
    foreach state s of the CFSM do
        if s contains A → α • aβ then
            Action[s] ← Action[s] ∪ Shift ;
        else if s contains A → α• that is the i^th rule then
            Action[s] ← Action[s] ∪ Reduce_i ;
        else if s contains S' → S$• then
            Action[s] ← Action[s] ∪ Accept ;
end
```

With $k > 0$

The k lookahead symbols can avoid the backtracking solution when a conflict occurs. Consider the case where a CFSM state contains both $A \to \alpha_1 \bullet \alpha_2$ and $B \to \gamma\bullet$: The first rule produces a **shift** and the second rule produces a **reduce**. This is a **shift-reduce conflict**. Based on the tokens on input, we know we should or not attempt shifting.

The k parameter will introduce a **context** which is a set of terminals appended in each item of the states. These terminals are the set of tokens that can follow the production of the rules.

With u that represents the context, the algorithms become:

```
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
```

```
ActionTable() begin
    foreach state s of the CFSM do
        if s contains [A → α • aβ, u] then
            foreach u ∈ First^k(aβu) do
                Action[s, u] ← Action[s, u] ∪ Shift ;
        else if s contains [A → α•, u ], that is the i^th rule then
            Action[s, u ] ← Action[s, u] ∪ Reduce_i ;
        else if s contains [S' → S$•, ε] then
            Action[s, ·] ← Action[s, ·] ∪ Accept ;
end
```

Exercises

Ex. 1. Give the corresponding LR(0) canonical finite state machine and its action table.

(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)	$C \rightarrow eD$		

Ex. 2. Simulate with a table (stack, input, action, output) the parser you built during the exercise 1 on the following string : `aeyzzd`

Ex. 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.

Ex. 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`.