Compile Principle - HW of Chapter 4

解雲暄　3190105871

In this file, `EPSILON` represents ϵ, i.e. the empty string.

> 4.8 Consider the grammar:
>
> ```
> lexp -> atom | list
> atom -> number | identifier
> list -> (lexp-seq)
> lexp-seq -> lexp-seq lexp | lexp
> ```
>
> (a) Remove the left recursion
>
> (b) Construct the First and Follow set of the nonterminals of the resulting grammar.
>
> (c) Show the grammar is LL(1).
>
> (d) Construct the LL(1) table for the resulting grammar.
>
> (e) Show the actions of the corresponding parser, given the following input string `(a (b (2)) (c))`.

(a) Production `lexp-seq -> lexp-seq lexp | lexp` is left-recursive. We rewrite it to be:

`lexp-seq -> lexp seq`, `seq -> lexp seq | EPSILON`. That is, the grammar after removing the left recursion is:

```
lexp -> atom | list
atom -> number | identifier
list -> (lexp-seq)
lexp-seq -> lexp seq
seq -> lexp seq | EPSILON
```

(b)

First Sets:

Terminals: *number*, *identifier*, *(* and *)*.

We can first get to know that $First(atom) = \{number,\ identifier\}$ by production 2 and that $First(list) = \{(\}$ by production 3.

From production 1 we know that $First(lexp) = First(atom) \cup First(list) = \{number,\ identifier,\ (\}$, and from production 4 we know that $First(lexp\text{-}seq) = First(lexp) = \{number,\ identifier,\ (\}$.

From production 5 we know that $First(seq) = First(lexp) \cup \{\epsilon\} = \{number,\ identifier,\ (,\ \epsilon\}$

Follow Sets:

As `lexp` is the start symbol, there is $\$ \in Follow(lexp)$.

From production 1 we can know that $Follow(lexp) \subseteq Follow(atom)$ and $Follow(lexp) \subseteq Follow(list)$.

From production 3 we can know that $) \in Follow(lexp\text{-}seq)$

From production 4 we can know that $First(seq) - \{\epsilon\} \subseteq Follow(lexp)$, and as $\epsilon \in First(seq)$, there is also $Follow(lexp\text{-}seq) \subseteq Follow(lexp)$. It can also be known that $Follow(lexp\text{-}seq) \subseteq Follow(seq)$.

Therefore the total first and follow sets are:

Non-terminals	**First Sets**	**Follow Sets**
lexp	$\{number,\ identifier,\ (\}$	$\{number,\ identifier,\ (,\ )\ , \$\}$
atom	$\{number,\ identifier\}$	$\{number,\ identifier,\ (,\ )\ , \$\}$
list	$\{\ (\ \}$	$\{number,\ identifier,\ (,\ )\ , \$\}$
lexp-seq	$\{number,\ identifier,\ (\}$	$\{\ )\ \}$
seq	$\{number,\ identifier,\ (,\ \epsilon\}$	$\{\ )\ \}$

(c) We can construct the LL(1) parsing table with the first and follow sets above: (the LHS i.e. Left Hand Side of each production has been omitted as they are shown in the leftmost column of the table)

LHS	**number**	**identifier**	**(**	**)**	**$**
lexp	`->atom`	`->atom`	`->list`		
atom	`->number`	`->identifier`			
list			`->(lexp-seq)`		
lexp-seq	`->lexp seq`	`->lexp seq`	`->lexp seq`		
seq	`->lexp seq`	`->lexp seq`	`->lexp seq`	`->EPSILON`	

As we can see, there is no conflicting entries in the table, or in other words, there is at most 1 production in each table entry. This satisfies the definition of LL(1) grammar. Therefore the grammar is LL(1).

(d) Has been finished above in question (c).

(e)

Stack	Input	Action
lexp $	`(a(b(2))(c))$`	`->list`
list $	`(a(b(2))(c))$`	`->(lexp-seq)`
(lexp-seq) $	`(a(b(2))(c))$`	**Match** `(`
lexp-seq) $	`a(b(2))(c))$`	`->lexp seq`
lexp seq) $	`a(b(2))(c))$`	`->atom`
atom seq) $	`a(b(2))(c))$`	`->identifier`
identifier seq) $	`a(b(2))(c))$`	**Match** `a`
seq) $	`(b(2))(c))$`	`->lexp seq`
lexp seq) $	`(b(2))(c))$`	`->list`
list seq) $	`(b(2))(c))$`	`->(lexp-seq)`
(lexp-seq) seq) $	`(b(2))(c))$`	**Match** `(`
lexp-seq) seq) $	`b(2))(c))$`	`->lexp seq`
lexp seq) seq) $	`b(2))(c))$`	`->atom`
atom seq) seq) $	`b(2))(c))$`	`->identifier`
identifier seq) seq) $	`b(2))(c))$`	**Match** `b`
seq) seq) $	`(2))(c))$`	`->lexp seq`
lexp seq) seq) $	`(2))(c))$`	`->list`
list seq) seq) $	`(2))(c))$`	`->(lexp-seq)`
(lexp-seq) seq) seq) $	`(2))(c))$`	**Match** `(`
lexp-seq) seq) seq) $	`2))(c))$`	`->lexp seq`
lexp seq) seq) seq) $	`2))(c))$`	`->atom`
atom seq) seq) seq) $	`2))(c))$`	`->number`
number seq) seq) seq) $	`2))(c))$`	**Match** `2`
seq) seq) seq) $	`))(c))$`	`->EPSILON`
) seq) seq) $	`))(c))$`	**Match** `)`
seq) seq) $	`)(c))$`	`->EPSILON`
) seq) $	`)(c))$`	**Match** `)`
seq) $	`(c))$`	`->lexp seq`
lexp seq) $	`(c))$`	`->list`

Stack	Input	Action
list seq) $	`(c))$`	`->(lexp-seq)`
(lexp-seq) seq) $	`(c))$`	**Match** `(`
lexp-seq) seq) $	`c))$`	`->lexp seq`
lexp seq) seq) $	`c))$`	`->atom`
atom seq) seq) $	`c))$`	`->identifier`
identifier seq) seq) $	`c))$`	**Match** `c`
seq) seq) $	`))$`	`->EPSILON`
) seq) $	`))$`	**Match** `)`
seq) $	`)$`	`->EPSILON`
) $	`)$`	**Match** `)`
$	`$`	**ACCEPT**

> 4.12 a. Can an LL(1) grammar be ambiguous? Why or why not?
>
> b. Can an ambiguous grammar be LL(1)? Why or why not?
>
> c. Must an unambiguous grammar be LL(1)? Why or why not?

(a) No. For each possible pair of non-terminal on the top of the stack (i.e. leftmost non-terminal) and the input token (i.e. the pending terminal), there is at most 1 production to use. So it's impossible for an input string to get 2+ different parsing trees. So LL(1) grammar must not be ambiguous.

(b) No. If the grammar is ambiguous, then there must be some points where 2+ choices are available, which means that there will be 2+ productions in an entry of the parsing table. But this violates the definition of LL(1) grammar. So no ambiguous grammar can be LL(1).

(c) No. Left-recursive grammars or grammars without left-factoring may be unambiguous but they are still not LL(1). Other grammars which are complex enough maybe also not LL(1). In fact, most programming language CFGs used now are not LL(1) because of their complexity.