1 Exercises

Exercise 1. Consider the following grammar:

$$S ::= \texttt{(}L\texttt{)} \mid \texttt{a}$$
$$L ::= L\ S \mid \epsilon$$

a. What language does this grammar describe?

b. Show the derivation for the sentence `( a ( ) ( a ( a ) ) )`.

c. Derive an equivalent LL(1) grammar.

Exercise 2. Show that the following grammar is ambiguous:

$$S ::= \texttt{a}\ S\ \texttt{b}\ S \ \mid\ \texttt{b}\ S\ \texttt{a}\ S \ \mid \epsilon$$

Exercise 3. Consider the following context-free grammar:

$$S ::= A\ \texttt{a}$$
$$A ::= \texttt{b}\ \texttt{d}\ B \mid \texttt{e}\ B$$
$$B ::= \texttt{c}\ A \mid \texttt{d}\ B \mid \epsilon$$

a. Compute first and follow sets for S, A and B.

b. Construct an LL(1) parsing table for this grammar.

c. Show the steps in the parse for `bdcea`.

Exercise 4. Consider the following grammar:

$$S ::= L\ \texttt{=}\ R$$
$$S ::= R$$
$$L ::= \texttt{*}\ R$$
$$L ::= \texttt{i}$$
$$R ::= L$$

a. Construct the canonical LR(1) collection.

b. Construct the Action and Goto tables.

c. Show the steps in the parse for `*i=i`.

Exercise 5. Suppose we wish to add support for the do statement in *j*--.

```
statement ::= block
            | DO statement WHILE parExpression SEMI
            | IF parExpression statement [ELSE statement]
            | RETURN [expression] SEMI
            | SEMI
            | WHILE parExpression statement
            | statementExpression SEMI
```

What changes will you need to make in the hand-written and JavaCC parsers in the *j*-- code tree?

2 Solutions

Solution 1. a. The grammar describes parenthesized strings, such as `()`, `(a)`, `(a(a))`, and so on.

b. A derivation for the sentence `( a ( ) ( a ( a ) ) )`:

$$
\begin{aligned}
S &::= \texttt{(}\ L\ \texttt{)} \\
&::= \texttt{(}\ LS\ \texttt{)} \\
&::= \texttt{(}\ LSS\ \texttt{)} \\
&::= \texttt{(}\ LSSS\ \texttt{)} \\
&::= \texttt{(}\ SSS\ \texttt{)} \\
&::= \texttt{(a}\ SS\ \texttt{)} \\
&::= \texttt{(a ( )}\ S\ \texttt{)} \\
&::= \texttt{(a ( ) (}\ L\ \texttt{))} \\
&::= \texttt{(a ( ) (}\ LS\ \texttt{))} \\
&::= \texttt{(a ( ) (}\ LSS\ \texttt{))} \\
&::= \texttt{(a ( ) (}\ SS\ \texttt{))} \\
&::= \texttt{(a ( ) (a}\ S\ \texttt{))} \\
&::= \texttt{(a ( ) (a (}\ L\ \texttt{)))} \\
&::= \texttt{(a ( ) (a (}\ LS\ \texttt{)))} \\
&::= \texttt{(a ( ) (a (}\ S\ \texttt{)))} \\
&::= \texttt{(a ( ) (a (a)))}
\end{aligned}
$$

c. An equivalent LL(1) grammar:

$$
\begin{aligned}
&S ::= \texttt{(}\ L\ \texttt{)} \\
&S ::= \texttt{a} \\
&L ::= X' \\
&X' ::= SX' \\
&X' ::= \epsilon
\end{aligned}
$$

Solution 2.

a. Two leftmost derivations (shown below) are possible for the sentence `abab`, and hence the grammar is ambiguous.

$$
\begin{aligned}
S &::= \texttt{a}S\texttt{b}S \\
&::= \texttt{ab}S\texttt{a}S\texttt{b}S \\
&::= \texttt{aba}S\texttt{b}S \\
&::= \texttt{abab}S \\
&::= \texttt{abab}
\end{aligned}
$$

$$
\begin{aligned}
S &::= \texttt{a}S\texttt{b}S \\
&::= \texttt{ab}S \\
&::= \texttt{aba}S\texttt{b}S \\
&::= \texttt{abab}S \\
&::= \texttt{abab}
\end{aligned}
$$

Solution 3. The grammar with numbered rules:

1. $S ::= A\texttt{a}$
2. $A ::= \texttt{bd}B$
3. $A ::= \texttt{e}B$
4. $B ::= \texttt{c}A$
5. $B ::= \texttt{d}B$
6. $B ::= \epsilon$

a. First and follow sets:

$\text{first}(S) = \{\texttt{b}, \texttt{e}\}$
$\text{first}(A) = \{\texttt{b}, \texttt{e}\}$
$\text{first}(B) = \{\texttt{c}, \texttt{d}, \epsilon\}$

$\text{follow}(S) = \{\texttt{\#}\}$
$\text{follow}(A) = \{\texttt{a}\}$
$\text{follow}(B) = \{\texttt{a}\}$

b. LL(1) parse table:

	a	b	c	d	e	#
S		1			1	
A		2			3	
B	6		4	5		

c. The steps in parsing `bcdea`:

Stack	Input	Output
#S	bdcea#	1
#aA	bdcea#	2
#aBdb	bdcea#	
#aBd	dcea#	
#aB	cea#	4
#aAc	cea#	
#aA	ea#	3
#aBe	ea#	
#aB	a#	6
#a	a#	
#	#	✓

Solution 4. Augmented grammar:

0. S' ::= S
1. S ::= L = R
2. S ::= R
3. L ::= * R
4. L ::= i
5. R ::= L

a. The canonical LR(1) collection:

$$s_0 = \{[S' ::= \cdot S, \texttt{\#}], [S ::= \cdot L\texttt{=}R, \texttt{\#}], [S ::= \cdot R, \texttt{\#}], [L ::= \cdot\texttt{*}R, \texttt{=/\#}], [L ::= \cdot\texttt{i}, \texttt{=/\#}], [R ::= \cdot L, \texttt{\#}]\}$$

$$\text{goto}(s_0, S) = \{[S' ::= S\cdot, \texttt{\#}]\} = s_1$$

$$\text{goto}(s_0, L) = \{[S ::= L \cdot \texttt{=}R, \texttt{\#}], [R ::= L\cdot, \texttt{\#}]\} = s_2$$

$$\text{goto}(s_0, R) = \{[S ::= R\cdot, \texttt{\#}]\} = s_3$$

$$\text{goto}(s_0, \texttt{*}) = \{[L ::= \texttt{*} \cdot R, \texttt{=/\#}], [R ::= \cdot L, \texttt{=/\#}], [L ::= \cdot\texttt{*}R, \texttt{=/\#}], [L ::= \cdot\texttt{i}, \texttt{=/\#}]\} = s_4$$

$$\text{goto}(s_0, \texttt{i}) = \{[L ::= \texttt{i}\cdot, \texttt{=/\#}]\} = s_5$$

$\mathrm{goto}(s_2, \texttt{=}) = \{[S ::= L\texttt{=} \cdot R, \texttt{\#}], [R ::= \cdot L, \texttt{\#}], [L ::= \cdot\texttt{*}R, \texttt{\#}], [L ::= \cdot\texttt{i}, \texttt{\#}], \} = s_6$

$\mathrm{goto}(s_4, L) = \{[R ::= L\cdot, \texttt{=/\#}]\} = s_7$

$\mathrm{goto}(s_4, R) = \{[L ::= \texttt{*}R\cdot, \texttt{=/\#}]\} = s_8$

$\mathrm{goto}(s_4, \texttt{*}) = \{[L ::= \texttt{*} \cdot R, \texttt{=/\#}], [R ::= \cdot L, \texttt{=/\#}], [L ::= \cdot\texttt{*}R, \texttt{=/\#}], [L ::= \cdot\texttt{i}, \texttt{=/\#}]\} = s_4$

$\mathrm{goto}(s_4, \texttt{i}) = \{[L ::= \texttt{i}\cdot, \texttt{=/\#}]\} = s_5$

$\mathrm{goto}(s_6, L) = \{[R ::= L\cdot, \texttt{\#}]\} = s_9$

$\mathrm{goto}(s_6, R) = [S ::= L\texttt{=}R\cdot, \texttt{\#}]\} = s_{10}$

$\mathrm{goto}(s_6, \texttt{*}) = \{[L ::= \texttt{*} \cdot R, \texttt{\#}], [R ::= \cdot L, \texttt{\#}], [L ::= \cdot\texttt{*}R, \texttt{\#}], [L ::= \cdot\texttt{i}, \texttt{\#}]\} = s_{11}$

$\mathrm{goto}(s_6, \texttt{i}) = \{[L ::= \texttt{i}\cdot, \texttt{\#}]\} = s_{12}$

$\mathrm{goto}(s_{11}, L) = \{[R ::= L\cdot, \texttt{\#}]\} = s_9$

$\mathrm{goto}(s_{11}, R) = [L ::= \texttt{*}R\cdot, \texttt{\#}]\} = s_{13}$

$\mathrm{goto}(s_{11}, \texttt{*}) = \{[L ::= \texttt{*} \cdot R, \texttt{\#}], [R ::= \cdot L, \texttt{\#}], [L ::= \cdot\texttt{*}R, \texttt{\#}], [L ::= \cdot\texttt{i}, \texttt{\#}]\} = s_{11}$

$\mathrm{goto}(s_{11}, \texttt{i}) = \{[L ::= \texttt{i}\cdot, \texttt{\#}]\} = s_{12}$

b. The Action and Goto tables:

	Action				Goto		
	=	*	i	#	S	L	R
0		$s4$	$s5$		1	2	3
1				✓			
2	$s6$			$r5$			
3				$r2$			
4		$s4$	$s5$			7	8
5	$r4$			$r4$			
6		$s11$	$s12$			9	10
7	$r5$			$r5$			
8	$r3$			$r3$			
9				$r5$			
10				$r1$			
11		$s11$	$s12$			9	13
12				$r4$			
13				$r3$			

c. The steps in parsing `*i=i`:

Stack	Input	Output
0	`*i=i#`	$s4$
0*4	`i=i#`	$s5$
0*4i5	`=i#`	$r4$
0*4L7	`=i#`	$r5$
0*4R8	`=i#`	$r3$
0L2	`=i#`	$s6$
0L2=6	`i#`	$s12$
0L2=6i12	`#`	$r4$
0L2=6L9	`#`	$r5$
0L2=6R10	`#`	$r1$
0S1	`#`	✓

Solution 5.

grammar

```
statement ::= block
            | DO statement WHILE parExpression SEMI
            | IF parExpression statement [ELSE statement]
            | RETURN [expression] SEMI
            | SEMI
            | WHILE parExpression statement
            | statementExpression SEMI
```

Parser.java

```java
    private JStatement statement() {
        int line = scanner.token().line();
        if (see(LCURLY)) {
            return block();
        } else if (have(DO)) {
            JStatement statement = statement();
            mustBe(WHILE);
            JExpression test = parExpression();
            mustBe(SEMI);
            return new JDoStatement(line, statement, test);
        } else if (have(IF)) {
            JExpression test = parExpression();
            JStatement consequent = statement();
            JStatement alternate = have(ELSE) ? statement() : null;
            return new JIfStatement(line, test, consequent, alternate);
        } else if (have(RETURN)) {
            if (have(SEMI)) {
                return new JReturnStatement(line, null);
            } else {
                JExpression expr = expression();
                mustBe(SEMI);
                return new JReturnStatement(line, expr);
            }
        } else if (have(SEMI)) {
            return new JEmptyStatement(line);
        } else if (have(WHILE)) {
            JExpression test = parExpression();
            JStatement statement = statement();
            return new JWhileStatement(line, test, statement);
        } else {
            // Must be a statementExpression.
            JStatement statement = statementExpression();
            mustBe(SEMI);
            return statement;
        }
    }
```

j--.jj

```
private JStatement statement():
{
    int line = 0;
    JStatement statement = null;
    JExpression test = null;
    JStatement consequent = null;
    JStatement alternate = null;
    JStatement body = null;
    JExpression expr = null;
}
{
    try {
        statement = block() |
        <DO> { line = token.beginLine; }
        body = statement()
        <WHILE>
        test = parExpression()
        <SEMI>
        { statement = new JDoStatement( line, body, test ); } |
        <IF> { line = token.beginLine; }
        test = parExpression()
        consequent = statement()

        // Even without the lookahead below, which is added to
        // suppress JavaCC warnings, dangling if-else problem is
        // resolved by binding the alternate to the closest
        // consequent.
```

```
        [
            LOOKAHEAD( <ELSE> )
            <ELSE> alternate = statement()
        ]
        { statement =
            new JIfStatement( line, test, consequent, alternate ); } |
        <RETURN> { line = token.beginLine; }
        [
            expr = expression()
        ]
        <SEMI>
        { statement = new JReturnStatement( line, expr ); } |
        <SEMI>
        { statement = new JEmptyStatement( line ); } |
        <WHILE> { line = token.beginLine; }
        test = parExpression()
        body = statement()
        { statement = new JWhileStatement( line, test, body ); } |
        // Must be a statementExpression.
        statement = statementExpression()
        <SEMI>
    }
    catch ( ParseException e ) {
        recoverFromError( new int[] { SEMI, EOF }, e );
    }
    { return statement; }
}
```