LALR Parsing Handout 2

We now construct an LALR parser for a very simple algebraic grammar G where the only variable is the start symbol E, the terminal alphabet is $\{a, +, *, (,)\}$, and the productions of G are given in Figure 1. We will use the string $w = (a + a) * a \in G$ to guide us in filling out the Action and Goto tables of an LALR parser for G. The parse tree for w is shown in Figure 2.

1. $E \longrightarrow E + E$

2. $E \longrightarrow E * E$

3. $E \longrightarrow (E)$

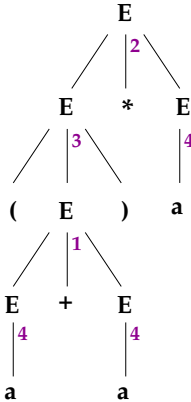4. $E \longrightarrow \mathbf{a}$

Figure1: G

Figure2: Parse Tree of (a+a)∗a

$L(G)$ is a the language of algebraic expressions where addition and multiplication are the only operations, and where **a** stands for any variable name, such as x, y, or z. Thus, the expression $x + y * z$ will be simplified to $a + a * a$

The output of the parser is the *reverse rightmost derivation* of the input string. If the input string is $(a + a) * a$ the leftmost derivation is

$$E \overset{2}{\Rightarrow} E * E \overset{3}{\Rightarrow} (E) * E \overset{1}{\Rightarrow} (E + E) * E \overset{4}{\Rightarrow} (a + E) * E \overset{4}{\Rightarrow} (a + a) * E \overset{4}{\Rightarrow} (a + a) * a$$

and the rightmost derivation is

$$E \overset{2}{\Rightarrow} E * E \overset{4}{\Rightarrow} E * a \overset{3}{\Rightarrow} (E) * a \overset{1}{\Rightarrow} (E + E) * a \overset{4}{\Rightarrow} (E + a) * a \overset{4}{\Rightarrow} (a + a) * a$$

The leftmost derivation is obtained by preorder visitation of the internal nodes of the parse tree and the reverse rightmost derivation, which is the output of the parser is the string 441342, obtained by postorder visitation of the internal nodes of the parse tree.

We first give the i.d. sequence of a non-deterministic PDA, a bottom up parser, with input w. We show the stack and the input file at each step, but we do not show the state of the PDA. The stack is to the left of : and the unread input is to the right. Since the parser is non-deterministic, we don't need to worry about a bottom-of-the-stack symbol or an EOF symbol, which we write as $.

```
    stack : input    output
1.        : (a+a)*a
2.      ( : a+a)*a
3.     (a : +a)*a
4.     (E : +a)*a   4
5.    (E+ : a)*a    4
```

```
6.   (E+a  :  )*a     4
7.   (E+E  :  )*a     44
8.    (E+  :  )*a     44
9.     (E  :  )*a     441
10.    (E  :  )*a     441
11.   (E)  :  *a      441
12.    (E  :  *a      441
13.     (  :  *a      441
14.     E  :  *a      4413
15.    E*  :  a       4413
16.   E*a  :          4413
17.   E*E  :          44134
18.    E*  :          44134
19.     E  :          44134
20.     E  :          441342
```

The PDA state, which is not shown, is used to keep track of intermediate states when more than one symbol needs to be popped. For example, at step 7, the string E+E needs to be popped off and then replaced with E, giving an output of 1. However, under our rules, that takes three steps, 8, 9, and 10. from 19. to 20., E is popped off and then E is pushed. It's a different E, however. The E that is pushed is at the root of the parse tree, while the E that is popped is the rightmost child of the root.

But we really want the parser to be deterministic. We handle that by having a stack state, a positive integer, after each symbol on the stack. We also have stack state 0 at the bottom.

The LALR parser is a 1-lookahead parser, meaning that it can look (peek) at the next input symbol without reading it. If there are no more input symbols, the lookahead returns the EOF symbol, which we represent by $. Each step of the LALR parser starts with peeking at the top stack state and the next input symbol. The parser then consults the action table, where there is a row for each stack state and a column for each possible lookahead symbol.

Initially, the stack has no grammar symbols, although it contains the stack state "0." We now describe one step of the LALR parser.

If the entry of the action table is "sn" where n is some stack state, the parser reads the next input symbol and then pushes that symbol onto the stack, and then pushes the stack state "n" onto the stack. We call that step "shift n."

If the entry of the action table is "rn," we look at production number "n" in the grammar. The parser then successively pops off the symbols of the right hand side of that production, along with their stack states. This exposes a new stack state, say "j." The parser then pushes the variable symbol, say "V," which is the left hand side of the production. The parser then must push a stack state, which is the entry in the Goto table in row "j" and column "V." The parser outputs "n." We call this step "reduce n."

The entry of the action table in row "1" and column "$" is **halt**. If the top stack state is "1" and the input file is empty, the stack contains only one symbol, the start symbol. The parser pops the start symbol and then halts. The stack and input file will then both be empty, except that the stack state will be 0.

If at any step, the parser tries to read a blank entry of the Action or Goto table, that is an error, that

is, the input string is not a member of the language.

Stack States

Our first step is to write *stack states*, inside the left hand side of each production. During the computation, the bottom item on the stack is the stack state 0. The other items are alternating grammar symbols and stack states. If a grammar symbol is on the stack, the stack state right above it is one of the subscripts of that symbol on the right hand side of some production.

We will not discuss how those stack states are computed. You will learn it in the compiler course. Remember that states 0 and 1 are reserved.

1. $E \longrightarrow E_{1,7} +_2 E_3$

2. $E \longrightarrow E_{1,3,7} *_4 E_5$

3. $E \longrightarrow (_6 E_7)_8$

4. $E \longrightarrow \mathbf{a}_9$

The stack state 0 is always the bottom item on the stack, and is used nowhere else. The stack state 1 is only used with the start symbol, and then only when the start symbol is at the bottom of the stack.

Filling the Action and Goto Tables

We start with blank Action and Goto Tables, and then fill in entries as we walk through a parsings of various strings of the language. If we pick the right set of strings, we will fill in all non-blank entries.

	ACTION						GOTO
	a	+	*	(	)	$	E
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							

Our first string will be $w = (\mathrm{a} + \mathrm{a}) \star \mathrm{a}$. Here is the i.d. sequence.

$\$_0 : (\mathrm{a} + \mathrm{a}) \star \mathrm{a}\$$

We shift (and then push stack state 6, since the stack state after (must be 6. Thus $\mathrm{Action}(0, ( ) = \mathrm{s6}$.

$\$_0 (_6 : \mathrm{a} + \mathrm{a}) \star \mathrm{a}\$$

Action(6,a) = s9, since the stack state after symbol a must be 9.

$\$_0(_6a_9 \ : \ +a) \star a\$$

Action(9,+) = r4 since the handle at the top of the stack is the right side of production 4. We pop a and push E. The output of this reduction step is 4.

When we push E onto stack state 6, the new stack state is 7, which we can see by examining production 3. Thus Goto(6,E) = 7.

$\$_0(_6E_7 \ : \ +a) \star a\$$

We read and push + and the new stack state is 2. Thus Action(7,+) = s2.

$\$_0(_6E_7+_2 \ : \ a) \star a\$$

We read and push a and the new stack state is 9, thus Action(2,a) = s9.

$\$_0(_6E_7 +_2 a_9 \ : \ ) \star a\$$

We must reduce a to E, thus $\text{Action}(9,)) = r4$. In production 1, the last stack state is 3, and thus Goto(2,E) = 3. The output is 4.

$\$_0(_6E_7 +_2 E_3 \ : \ ) \star a\$$

It is now time to do "reduce 1." Pop off E+E and push E on top of (and then stack state 7. Thus $\text{Action}(3,)) = r1$. We already knew that Goto(6,E) = 7. The output is 1.

$\$_0(_6E_7 \ : \ ) \star a\$$

We shift the right parenthesis, therefore $\text{Action}(7,)) = s8$.

$\$_0(_6E_7)_8 \ : \ \star a\$$

We need to reduce 3. Thus Action(8,$\star$) = r3. Since we are pushing E onto stack state 0, the new stack state must be 1, Thus Goto(0,E) = 1. The output is 3.

$\$_0E_1 \ : \ \star a\$$

Shift $\star$. Thus Action(1,$\star$) = s4.

$\$_0E_1\star_4 \ : \ a\$$

Shift a. Thus Action(4,a) = s9.

$\$_0E_1 \star_4 a_9 \ : \ \$$

We must reduce 4, and thus Action(9,$\$$) = r4. From production 2, we see that Goto(4,E) = 5. The output is 4.

$\$_0E_1 \star_4 E_5 \ : \ \$$

We pop the three symbols E$\star$E and their stack states, and then push E with stack state 1. Thus Action(5,$\$$) = r2. We already know that Goto(0,E) = 1. The output is 2.

$\$_0E_1 \ : \ \$$

Action(1,$\$$) = **halt**.

$\$_0 \ : \ \$$

The stack and the input file are empty. The output is 441342. The tables are now partially filled in:

			ACTION				GOTO
	a	+	*	(	)	$	E
0				s6			1
1						halt	
2	s9						3
3					r1		
4	s9						5
5						r2	
6	s9						7
7		s2			s8		
8			r3				
9		r4			r4	r4	

There are additional entries in the Action table, obtained by parsing other strings.

- Parsing a+a+a shows that Action(0,a) = s9 and Action(3,+) = r1.

- Parsing a⋆a+a shows that Action(1,⋆ = s4, Action(1,+) = s2, Action(2,$) = r1, and Action(5.+) = r2.

- Parsing a+a⋆a shows that Action(3,⋆) = s4.

- Parsing a+(a) shows that Action$(2, ()$ = s6.

- Parsing a⋆(a) shows that Action$(4, ()$ = s6.

- Parsing ((a)) shows that Action$(6, ()$ = s6, Action$(8,))$ = r3, and Action(8,$) = r3.

- Parsing (a⋆a) shows that Action(9,⋆) = r4, Action$(5,))$ = r2, and Action(7,⋆) = s4.

- Parsing (a)+a shows that Action(8,+) = r3.

- Parsing (a)⋆a shows that Action(8,⋆) = r3.

- Parsing a⋆a⋆a shows that Action(5,⋆) = r2.

Completed Action and Goto tables:

	ACTION						GOTO
	a	+	*	(	)	$	E
0	s9			s6			1
1		s2	s4			**halt**	
2	s9			s6			3
3		r1	s4		r1	r1	
4	s9			s6			5
5		r2	r2		r2	r2	
6	s9			s6			7
7		s2	s4		s8		
8		r3	r3		r3	r3	
9		r4	r4		r4	r4	

Answer these questions.

1. Which entry guarantees that addition is left-associative?

2. Which entry guarantees that multiplication is left-associative?

3. Which two entries guarantee that multiplication has precedence over addition?