LALR Parsing Handout 1

Some, but not all, context-free languages can be parsed with an LALR parser. The input of the parser is a string in the language, while the output is an abbreviated (meaning a sequence of production labels). reverse rightmost derivation of the input.

Here is a context-free grammar G for a “toy” algebraic language, whose start symbol is E (for *expression*), followed by the Action and Goto tables for an LALR parser for G. The productions of G are labeled 1–3. in this example.

1. $E \to E + E$
2. $E \to E * E$
3. $E \to a$

The symbol a represents any variable.

The parser has a stack, which, at any step, contains bottom of stack symbol, \$, and grammar symbols. Each of those symbols has an associated *stack state* which we write as a subscript. In our LALR parser of G, the stack states are $0 \ldots 6$. The state 0 is reserved for the bottom of stack symbol.

We annotate the right hand sides of each production with stack states:
1. $E \to E_{1,3,5} +_2 E_3$
2. $E \to E_{1,3,5} *_4 E_5$
3. $E \to a_6$

When you take the compiler class, you will learn an algorithm for computing these stack states.

The LALR parser contains three parts: the stack which grows and shrinks, the input file from which symbols are read one at a time, and the output file. We use \$ for both bottom of stack and end of file. We assume 1-lookahead, *i.e.,* the parser can peek at the next input symbol without reading it. The parser can also peek at the top stack state without popping it.

Steps of the LALR Parser. The LALR parser operates in steps. At each step the parser peeks at the top stack state and the next input symbol, which may be either a terminal of the language or the end of file symbol, then follows the instructions given by the Action and Goto tables, as shown below.

	ACTION				GOTO
	a	$+$	$*$	\$	E
0	s6				1
1		s2	s4	HALT	
2	s6				3
3		r1	s4	r1	
4	s6				5
5		r2	r2	r2	
6		r3	r3	r3	

The headers of rows of both the Action and Goto tables are the stack states. numbers 0 to 6 in this case. The headers of the columns of the Action table are the possible input symbols, including the end-of-file symbol \$. The headers of the columns of the Goto tables are the variables of the gammar. in this example, there is only one variable, the start symbol E. The stack state 0 is used only on top of the bottom-of-stack symbol, \$. Here is the sequence of steps.

1. Peek at the top stack state and the next input symbol (the lookahead), and read the action in at that in that row and column. A blank entry in that table means that that combination of stack state and input symbols will never occur if the input string is a generated by G.
2. Execute that action. There are three kinds of actions, halt, shift, and reduce.
   (a) HALT means that the parser if finished. The input file will consist of the end-of-file symbol, and the stack will consist of the bottom-of-stack symbol with stack state 0, followed by the start symbol with stack state 1. That is, in our example, the stack is $\$_0E_1$. The output file will be the string of productions of the rightmost derivation of the input, written in reverse order.
   (b) The action *shift*, written as sN where N is a stack state, means to "shift" the lookahead symbol to the stack, that is, read the lookahead and then push it followed by the stack state N.
   (c) The action *reduce* is the reverse of a production of the grammar, written rK, where K is the label of a production. At this time, the top several symbols of the stack should match the right hand side of production K. That string of symbols is called a *handle*. The handle is entirely popped, and the left hand side of production K, a variable, say V, is pushed, followed by a stack state as determined by the Goto table. The label K is then appended to the output file. The new stack state, pushed onto V is the entry under the column header V in the Goto table, and in the row for the stack state **before** we push V, but **after** we pop the handle.

Here is an example of the reduce action. Suppose the stack is $\$E_1 +_2 E_3 *_4 E_5$ and the lookahead symbol is "+". The Action table tells us to execute Action r2. Production 2 is $E \to E * E$, and the handle $E_3 *_4 E_5$ is popped, exposing stack state 2. The left hand side of the production is E which is pushed, followed by the stack state 3, which is the entry of the Goto table in column E and row 2. The new stack is $\$E_1 +_2 E_3$, and the production number 2 is appended to the output file. The lookahead symbol is not read, hance remains "+". At any given step, the stack, the remaining input, and the output constitute the **id** (instantaneous description) of the parser.

Example Computations. We show the computation of our LALR for two input strings. For the first example, let the input string be $a * a + a$. The rightmost derivation of that string is:

$$E \overset{1}{\Rightarrow} E + \underline{E} \overset{3}{\Rightarrow} \underline{E} + a \overset{2}{\Rightarrow} E * \underline{E} + a \overset{3}{\Rightarrow} \underline{E} * a + a \overset{3}{\Rightarrow} a * a + a$$

The output is 33231, the abbreviated[1] rightmost derivation of the input, written in reverse order.

The sequence of instantaneous desciptions of the LALR parser is shown below, where the stack, bottom to top, is shown in the first column. The remaining output is shown in the second column,

[1]Just the production labels.

and the current output string in the third. The fourth column shows the action taken at each step.

$\$_0$	$a*a+a\$$		
$\$_0 a_6$	$*a+a\$$		$s6$
$\$_0 E_1$	$*a+a\$$	3	$r3$
$\$_0 E_1 *_4$	$a+a\$$	3	$s4$
$\$_0 E_1 *_4 a_6$	$+a\$$	3	$s6$
$\$_0 E_1 *_4 E_5$	$+a\$$	33	$r3$
$\$_0 E_1$	$+a\$$	332	$r2$
$\$_0 E_1 +_2$	$a\$$	332	$s2$
$\$_0 E_1 +_2 a_6$	$\$$	332	$s6$
$\$_0 E_1 +_2 E_3$	$\$$	3323	$r3$
$\$_0 E_1$	$\$$	33231	$r1$

HALT

For our second example, let the input string be $w = a + a * a * a + a$. The output is 333232131, the reverse rightmost derivation of the input. The computation of the LALR parser consists of the **id** sequence:

$\$_0$	$a+a*a*a+a\$$		
$\$_0 a_6$	$+a*a*a+a\$$		$s6$
$\$_0 E_1$	$+a*a*a+a\$$	3	$r3$
$\$_0 E_1 +_2$	$a*a*a+a\$$	3	$s2$
$\$_0 E_1 +_2 a_6$	$*a*a+a\$$	3	$s6$
$\$_0 E_1 +_2 E_3$	$*a*a+a\$$	33	$r3$
$\$_0 E_1 +_2 E_3 *_4$	$a*a+a\$$	33	$s4$
$\$_0 E_1 +_2 E_3 *_4 a_6$	$*a+a\$$	33	$s6$
$\$_0 E_1 +_2 E_3 *_4 E_5$	$*a+a\$$	333	$r3$
$\$_0 E_1 +_2 E_3$	$*a+a\$$	3332	$r2$
$\$_0 E_1 +_2 E_3 *_4$	$a+a\$$	3332	$s4$
$\$_0 E_1 +_2 E_3 *_4 a_6$	$+a\$$	3332	$s6$
$\$_0 E_1 +_2 E_3 *_4 E_5$	$+a\$$	33323	$r3$
$\$_0 E_1 +_2 E_3$	$+a\$$	333232	$r2$
$\$_0 E_1$	$+a\$$	3332321	$r1$
$\$_0 E_1 +_2$	$a\$$	3332321	$s2$
$\$_0 E_1 +_2 a_6$	$\$$	3332321	$s6$
$\$_0 E_1 +_2 E_3$	$\$$	33323213	$r3$
$\$_0 E_1$	$\$$	333232131	$r1$

HALT

1. Sketch the parse tree.

2. The grammar is ambigous, but the parser resolves ambiguities, computing a unique derivation for any string in the language. Left associativity of addition is guaranteed by the entry r1 in row 3, in the column headed by the "+". Which entry of the action table guarantees that multiplication is left associative?

3. Which two entries in the action table cause multiplication to have precedence over addition?

4. Write the computation of the parser if the input is $a + a + a * a$. Use the same array format used for our two examples above.

$\$_0$	$a + a + a * a\$$		
$\$_0 E_1$	$\$$	3313321	$r1$

HALT

An Unambiguous Grammar

The grammar G is ambiguous; if an expression contains more than one operator, it has multiple parse trees. Instead of making the correct choices in the Action table, we can use an unambiguous grammar, such as G_2 given below. The grammar G_2 below generates the same language as G. We use The three variables: E (expression), T (term) and F (factor). Here is G_2:

1. $E \to E +_2 T_3$
2. $E \to T_4$
3. $T \to T *_5 F_6$
4. $T \to F_7$
5. $F \to a_8$

Here are the rightmost derivations of $a + a + a$, $a * a * a$, and $a + a * a$ using G_2.

$$E \overset{1}{\Rightarrow} E{+}T \overset{4}{\Rightarrow} E{+}F \overset{5}{\Rightarrow} E{+}a \overset{2}{\Rightarrow} E{+}T{+}a \overset{4}{\Rightarrow} E{+}F{+}a \overset{2}{\Rightarrow} E{+}a{+}a \overset{4}{\Rightarrow} T{+}a{+}a \overset{1}{\Rightarrow} F{+}a{+}a \overset{5}{\Rightarrow} a{+}a{+}a$$

$$E \overset{2}{\Rightarrow} T \overset{3}{\Rightarrow} T * F \overset{5}{\Rightarrow} T * a \overset{3}{\Rightarrow} T * F * a \overset{5}{\Rightarrow} T * a * a \overset{4}{\Rightarrow} F * a * a \overset{5}{\Rightarrow} a * a * a$$

$$E \overset{1}{\Rightarrow} E + T \overset{3}{\Rightarrow} E + T * F \overset{3}{\Rightarrow} E + T * a \overset{3}{\Rightarrow} E + F * a \overset{3}{\Rightarrow} E + a * a \overset{3}{\Rightarrow} T + a * a \overset{3}{\Rightarrow} F + a * a \overset{3}{\Rightarrow} a + a * a$$

5. Write the G_2 rightmost derivation of $a * a + a$.

6. Fill in the Action and Goto tables for an LALR parser for G_2.

	ACTION				GOTO		
	a	$+$	$*$	$\$$	E	T	F
0							
1				HALT			
2							
3							
4							
5							
6							
7							
8							

Dangling Else

When there is a “else” after two ”if”s, which ”if” does the ”else” pair with? Here is CF grammar, G_3, which isolates this problem. The start symbol S is the only variable. The symbol i represents “if(condition)”, e represents “else,” w represents “while(condition)” and a represents any other statement, such as an assignment statement.

I have annotated the grammar with stack states.

1. $S \to i_2 S_3$
2. $S \to i_2 S_3 e_4 S_5$
3. $S \to w_6 S_7$
4. $S \to a_8$

7. Fill in the Action and Goto tables for an LALR parser for G_3.

	ACTION					GOTO
	a	i	e	w	$\$$	S
0						
1					HALT	
2						
3						
4						
5						
6						
7						
8						

8. Which entry, or entries, solve the dangling else problem?

9. Walk through the actions of the LALR parser for the input string *iiwaea*.

$\$_0$	$iiwaea\$$		
$\$_0S_1$	$\$$	43421	$r1$

HALT

The output is the reverse rightmost derivation 43421.