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 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 actions are labeled $1 \ldots 3$ 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 stack contains grammar symbols, and each of those symbols must have an associated *stack state* written as a subscript. The bottom of stack symbol has the subscript 0. In this example, 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 the 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. $E \to a_6$

	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 LALR parser has 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 necessarily reading it. The parser can also peek at the top stack state without necessarily poppling 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.

Each row of the table is headed by a stack state, a number from $0 \ldots 6$ in this case. The columns of the ACTION table are labeled by the possible input symbols, including the bottom-of-stack symbol \$. Each column of the GOTO table is headed by a variable of the grammar; in this example, there is only one variable, the start symbol E.

A step operates as follows.

1. Peek at the top stack state and the next input symbol, and follow the instructions in the appropriate entry. A blank entry means that that combination of stack state and input symbol will never occur if the input string is a generated by G.
2. There are three kinds of actions, halt, shift, and reduce. HALT means that the parser if finished. The input file will be empty and the stack will consist of the bottom-of-stack symbol with stack state 0, followed by the start symbol with stack state 1, *i.e.,* E_1 in our example.
3. The action *shift* is written as s followed by a stack state N. At this action the current input symbol is read, then pushed onto the stack, and given the stack state N.
4. If the action is r following by a number, that number must be the label of one of the productions. The top one or more symbols in the stack will be the right hand side of that production, along with stack states. That string is called a *handle*. The entire handle is popped, and then the left-hand side of the production is pushed. The left hand side will be a variable. It will be given a stack state as determined by the GOTO table, which depends on the stack state onto which the variable is pushed. The production label is then written to the output file.

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 $w = a * a + a$. The rightmost derivation of w 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] reverse rightmost derivation of the inut.

The sequence of instantaneous desciptions of the LALR parser is shown below, where the stack is shown in the first column, bottom of the stack to the left, top to the right. The remaining output is shown in the second column, and the current output string in the third. The fourth column shows the action taken at that 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			

[1] Just the production labels.

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 **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 all ambiguities, computing only one derivation for each string in the language. In any derivation produced by the parser, addition and multiplication are both left associative. Left associativity of addition is guaranteed by the entry r1 in row 3, in the column headed by the plus sign. 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.

An Unambiguous Grammar

G is ambiguous. If an expression contains more than one operator, there are multiple parse trees. Ambiguity is resolved by accoiativity and precedence of operators, and parentheses can be introduced to override precedence. The LALR parser can be defined, as above, to enforce associativeity and precedence, but these ambiguities can also be resolved by using an unambiguous grammar. The grammar G_2 below generates the same language as G,but is unambigous. The three variables of G_2 are E (expression), T (term) and F (factor).

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. $F \to a_8$

G_2 enforces precedence of multiplication over addition and left-associativity of both operators. For example, we now have G_2 rightmost derivations of $a + a + a$, $a * a * a$, and $a + a * a$:

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

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

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

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

	ACTION				GOTO		
	a	$+$	$*$	$	E	T	F
0	s8				1	4	7
1				HALT			
2	s8					3	7
3		r1	s5	r1			
4		r2	s5	r2			
5	s8						6
6		r3	r3	r3			
7		r4	r4	r4			
8		r5	r5	r5			

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,” and a represents any other statement, such as an assignment statement.

I have annotated the grammar with stack states. When you take the compiler class, you will learn an algorithm for computing these 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	\$	E
0						
1					HALT	
2						
3						
4						
5						
6						
7						
8						

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

$\$_0$	$iiwaea\$$		

The output is the reverse rightmost derivation 43421.