LALR Parsing

Handout written by Maggie Johnson, revised by Julie Zelenski.

Motivation

Because a canonical LR(1) parser splits states based on differing lookahead sets, it can have many more states than the corresponding SLR(1) or LR(0) parser. Potentially it could require splitting a state with just one item into a different state for each subset of the possible lookaheads; in a pathological case, this means the entire power set of its follow set (which theoretically could contain all terminals—yikes!). It never actually gets that bad in practice, but a canonical LR(1) parser for a programming language might have an order of magnitude more states than an SLR(1) parser. Is there something in between?

With LALR (*lookahead LR*) parsing, we attempt to reduce the number of states in an LR(1) parser by merging similar states. This reduces the number of states to the same as SLR(1), but still retains some of the power of the LR(1) lookaheads. Let's examine the LR(1) configurating sets from an example given in the LR parsing handout.

```
    S' –> S
    S –> XX
    X –> aX
    X  –> b
I0:  S' –> •S, $
     S –> •XX, $
     X –> •aX, a/b
     X –> •b, a/b

I1:  S' –> S•, $

I2:  S –> X•X, $
     X –> •aX, $
     X –> •b, $

I3:  X –> a•X, a/b
     X –> •aX, a/b
     X –> •b, a/b

I4:  X –> b•, a/b

I5:  S –> XX•, $

I6:  X –> a•X, $
     X –> •aX, $
     X –> •b, $

I7:  X –> b•, $

I8:  X –> aX•, a/b

I9:  X –> aX•, $
```

Notice that some of the LR(1) states look suspiciously similar. Take I_3 and I_6 for example. These two states are virtually identical—they have the same number of items, the core of each item is identical, and they differ only in their lookahead sets. This observation may make you wonder if it possible to merge them into one state. The

same is true of I_4 and I_7, and I_8 and I_9. If we did merge, we would end up replacing those six states with just these three:

I_{36}: X –> a•X, a/b/$
X –> •aX, a/b/$
X –> •b, a/b/$

I_{47}: X –> b•, a/b/$

I_{89}: X –> aX•, a/b/$

But isn't this just SLR(1) all over again? In the above example, yes, since after the merging we coincidentally end up with the complete follow sets as the lookahead. This is not always the case however. Consider this example:

S' –> S
S –> Bbb | aab | bBa
B –> a

I_0: S' –> •S, $
S –> •Bbb, $
S –> •aab, $
S –> •bBa, $
B –> •a, b

I_1: S' –> S•, $

I_2: S –> B•bb, $

I_3: S –> a•ab, $
B –> a•, b
....

In an SLR(1) parser there is a shift-reduce conflict in state 3 when the next input is anything in Follow(B)which includes a and b. In LALR(1), state 3 will shift on a and reduce on b. Intuitively, this is because the LALR(1) state "remembers" that we arrived at state 3 after seeing an a. Thus we are trying to parse either Bbb or aab. In order for that first a to be a valid reduction to B, the next input has to be exactly b since that is the only symbol that can follow B in this particular context. Although elsewhere an expansion of B can be followed by an a, we consider only the subset of the follow set that can appear here, and thus avoid the conflict an SLR(1) parser would have.

LALR Merge Conflicts

Can merging states in this way ever introduce new conflicts? A shift-reduce conflict cannot exist in a merged set unless the conflict was already present in one of the original LR(1) configurating sets. When merging, the two sets must have the same core items. If the merged set has a configuration that shifts on a and another that reduces on a, both configurations must have been present in the original sets, and at least one of those sets had a conflict already.

Reduce-reduce conflicts, however, are another story. Consider the following grammar:

S' –> S
S –> aBc | bCc | aCd | bBd
B –> e
C –> e

The LR(1) configurating sets are as follows:

I_0: S' –> •S, $
S –> •aBc, $
S –> •bCc, $
S –> •aCd, $
S –> •bBd, $

I_1: S' –> S•, $

I_2: S –> a•Bc, $
S –> a•Cd, $
B –> •e, c
C –> •e, d

I_3: S –> b•Cc, $

I_3: S –> b•Cc, $
S –> b•Bd, $
C –> •e, c
B –> •e, d

I_4: S –> aB•c, $

I_5: S –> aC•d, $

I_6: B –> e•, c
C –> e•, d

I_7: S –> bC•c, $

I_8: S –> bB•d, $

I_9: B –> e•, d
C –> e•, c

I_{10}: S –> aBc•, $

I_{11}: S –> aCd•, $

I_{12}: S –> bCc•, $

I_{13}: S –> bBd•, $

We try to merge I_6 and I_9 since they have the same core items and they only differ in lookahead:

I_{69}: C –> e, c/d
B –> e, d/c

However, this creates a problem. The merged configurating set allows a reduction to either B or C when next token is c or d. This is a reduce-reduce conflict and can be an unintended consequence of merging LR(1) states. When such a conflict arises in doing a merging, we say the grammar is not LALR(1).

LALR Table Construction

A LALR(1) parsing table is built from the configurating sets in the same way as canonical LR(1); the lookaheads determine where to place reduce actions. In fact, if there are no mergable states in the configuring sets, the LALR(1) table will be identical to the corresponding LR(1) table and we gain nothing.

In the common case, however, there will be states that can be merged and the LALR table will have fewer rows than LR. The LR table for a typical programming language may have several thousand rows, which can be merged into just a few hundred for LALR. Due to merging, the LALR(1) table seems more similar to the SLR(1) and LR(0) tables, all three have the same number of states (rows), but the LALR may have fewer reduce actions—some reductions are not valid if we are more precise about the lookahead. Thus, some conflicts are avoided because an action cell with conflicting actions in SLR(1) or LR(0) table may have a unique entry in an LALR(1) once some erroneous reduce actions have been eliminated.

Brute Force?

There are two ways to construct LALR(1) parsing tables. The first (and certainly more obvious way) is to construct the LR(1) table and merge the sets manually. This is sometimes referred as the "brute force" way. If you don't mind first finding all the multitude of states required by the canonical parser, compressing the LR table into the LALR version is straightforward.

1. Construct all canonical LR(1) states.
2. Merge those states that are identical if the lookaheads are ignored, i.e., two states being merged must have the same number of items and the items have the same core (i.e., the same productions, differing only in lookahead). The lookahead on merged items is the union of the lookahead from the states being merged.
3. The successor function for the new LALR(1) state is the union of the successors of the merged states. If the two configurations have the same core, then the original successors must have the same core as well, and thus the new state has the same successors.
4. The action and goto entries are constructed from the LALR(1) states as for the canonical LR(1) parser.

Let's do an example, eh?

Consider the LR(1) table for the grammar given on page 1 of this handout. There are nine states.

State on top of stack	Action			Goto	
	a	b	$	S	X
0	s3	s4		1	2
1			Acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Looking at the configurating sets, we saw that states 3 and 6 can be merged, so can 4 and 7, and 8 and 9. Now we build this LALR(1) table with the six remaining states:

State on top of stack	Action			Goto	
	a	b	$	S	X
0	S36	s47		1	2
1			acc		
2	S36	s47			5
36	S36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

The More Clever Approach

Having to compute the LR(1) configurating sets first means we won't save any time or effort in building an LALR parser. However, the work wasn't all for naught, because when the parser is executing, it can work with the compressed table, thereby saving memory. The difference can be an order of magnitude in the number of states.

However there is a more efficient strategy for building the LALR(1) states called *step-by-step merging*. The idea is that you merge the configurating sets as you go, rather than waiting until the end to find the identical ones. Sets of states are constructed as in the LR(1) method, but at each point where a new set is spawned, you first check to see

whether it may be merged with an existing set. This means examining the other states to see if one with the same core already exists. If so, you merge the new set with the existing one, otherwise you add it normally.

Here is an example of this method in action:

S' –> S
S –> V = E
E –> F | E + F
F –> V | int | (E)
V –> id

Start building the LR(1) collection of configurating sets as you would normally:

I_0: S' –> •S, $
S –> •V = E, $
V –> •id, =

I_1: S' –> S•, $

I_2: S' –> V• = E, $

I_3: V –> id•, =

I_4: S –> V =•E, $
E –> •F, $/+
E –> •E + F, $/+
F –>•V, $/+
F –>•int, $/+
F –>•(E), $/+
V –>•id, $/+

I_5: S –> V = E•, $
E –> E• + F, $/+

I_6: E –> F•, $/+

I_7: F–> V•, $/+

I_8: F–> int•, $/+

I_9: F–> (•E), $/+
E –> •F,)/+
E –> •E + F,)/+
F –> •V,)/+
F –> •int,)/+
F –> •(E),)/+
V –> •id)/+

I_{10}: F–> (E•), $/+
E –> E• + F,)/+

When we construct state I_{11}, we get something we've seen before:

I_{11}: E –>F•,)/+

It has the same core as I_6, so rather than add a new state, we go ahead and merge with that one to get:

I_{611}: E –>F•, $/+/)

We have a similar situation on state I_{12} which can be merged with state I_7. The algorithm continues like this, merging into existing states where possible and only adding new states when necessary. When we finish creating the sets, we construct the table just as in LR(1).

LALR(1) Grammars

A formal definition of what makes a grammar LALR(1) cannot be easily encapsulated in a set of rules, because it needs to look beyond the particulars of a production in isolation to consider the other situations where the production appears on the top of the stack and what happens when we merge those situations. Instead we state that what makes a grammar LALR(1) is the absence of conflicts in its parser. If you build the parser and it is conflict-free, it implies the grammar is LALR(1) and vice-versa.

LALR(1) is a subset of LR(1) and a superset of SLR(1). A grammar that is not LR(1) is definitely not LALR(1), since whatever conflict occurred in the original LR(1) parser will still be present in the LALR(1). A grammar that is LR(1) may or may not be LALR(1) depending on whether merging introduces conflicts. A grammar that is SLR(1) is definitely LALR(1). A grammar that is not SLR(1) may or may not be LALR(1) depending on whether the more precise lookaheads resolve the SLR(1) conflicts. LALR(1) has proven to be the most used variant of the LR family. The weakness of the SLR(1) and LR(0) parsers mean they are only capable of handling a small set of grammars. The expansive memory needs of LR(1) caused it to languish for several years as a theoretically interesting but intractable approach. It was the advent of LALR(1) that offered a good balance between the power of the specific lookaheads and table size. The popular tools **`yacc`** and **`bison`** generate LALR(1) parsers and most programming language constructs can be described with an LALR(1) grammar (perhaps with a little grammar massaging or parser trickery to skirt some isolated issues).

Error Handling

As in LL(1) parsing tables, we can implement error processing for any of the variations of LR parsing by placing appropriate actions in the parse table. Here is a parse table for a simple arithmetic expression grammar with error actions inserted into what would have been the blank entries in the table.

E –> E + E | E * E | (E) | id

State on top of stack	Action						Goto
	id	+	*	(	)	$	E
0	s3	**e1**	**e1**	s2	**e2**	**e1**	1
1	**e3**	s4	s5	**e3**	**e2**	acc	
2	s3	**e1**	**e1**	s2	**e2**	**e1**	6
3	**e3**	r4	r4	**e3**	r4	r4	
4	s3	**e1**	**e1**	s2	**e2**	**e1**	7
5	s3	**e1**	**e1**	s2	**e2**	**e1**	8
6	**e3**	s4	s5	**e3**	s9	**e4**	
7	**e3**	r1	s5	**e3**	r1	r1	
8	**e3**	r2	r2	**e3**	r2	r2	
9	**e3**	r3	r3	**e3**	r3	r3	

Error e1 is called from states 0, 2, 4, 5 when we encounter an operator. All of these states expect to see the beginning of an expression, i.e., an id or a left parenthesis. One way to fix is for the parser to act as though id was seen in the input and shift state 3 on the stack (the successor for id in these states), effectively faking that the necessary token was found. The error message printed might be something like "missing operand". Error e2 is called from states 0, 1, 2, 4, 5 on finding a right parenthesis where we were expecting either the beginning of a new expression (or potentially the end of input for state 1). A possible fix: remove right parenthesis from the input and discard it. The message printed could be "unbalanced right parenthesis."

Error e3 is called from state 1, 3, 6, 7, 8, 9 on finding id or left parenthesis. What were these states expecting? What might be a good fix? How should you report the error to the user? Error e4 is called from state 6 on finding $. What is a reasonable fix? What do you tell the user?

Bibliography

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

K. Loudon, Compiler Construction. Boston, MA: PWS, 1997

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.