SLR and LR(1) Parsing

Handout written by Maggie Johnson and revised by Julie Zelenski.

LR(0) Isn't Good Enough

LR(0) is the simplest technique in the LR family. Although that makes it the easiest to learn, these parsers are too weak to be of practical use for anything but a very limited set of grammars. The examples given at the end of the LR(0) handout show how even small additions to an LR(0) grammar can introduce conflicts that make it no longer LR(0). The fundamental limitation of LR(0) is the zero, meaning no lookahead tokens are used. It is a stifling constraint to have to make decisions using only what has already been read, without even glancing at what comes next in the input. If we could peek at the next token and use that as part of the decision-making, we will find that it allows for a much larger class of grammars to be parsed.

SLR(1)

We will first consider SLR(1) where the S stands for simple ☺. SLR(1) parsers use the same LR(0) configurating sets and have the same table structure and parser operation, so everything you've already learned about LR(0) applies here. The difference comes in assigning table actions, where we are going to use one token of lookahead to help arbitrate among the conflicts. If we think back to the kind of conflicts we encountered in LR(0) parsing, it was the reduce actions that cause us grief. A state in an LR(0) parser can have at most one reduce action and cannot have both shift and reduce instructions. Since a reduce is indicated for any completed item, this dictates that each completed item must be in a state by itself. But let's revisit the assumption that if the item is complete, the parser must choose to reduce. Is that always appropriate? If we peeked at the next upcoming token, it may tell us something that invalidates that reduction. If the sequence on top of the stack could be reduced to the non-terminal `A`, what tokens do we expect to find as the next input? What tokens would tell us that the reduction is not appropriate? Perhaps `Follow(A)` could be useful here!

The simple improvement that SLR(1) makes on the basic LR(0) parser is to reduce only if the next input token is a member of the follow set of the non-terminal being reduced. When filling in the table, we don't assume a reduce on all inputs as we did in LR(0), we selectively choose the reduction only when the next input symbols in a member of the follow set. To be more precise, here is the algorithm for SLR(1) table construction (note all steps are the same as for LR(0) table construction except for 2a)

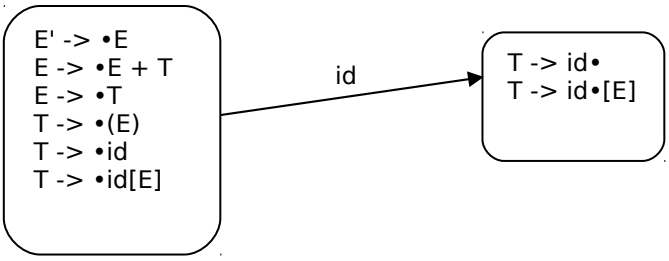
1. Construct F = {I_0, I_1, ... I_n}, the collection of LR(0) configurating sets for G'.
2. State i is determined from I_i. The parsing actions for the state are determined as follows:
    a) If A -> u• is in I_i then set Action[i,a] to reduce A -> u for all a in Follow(A) (A is not S').
    b) If S' -> S• is in I_i then set Action[i,$] to accept.
    c) If A -> u•av is in I_i and successor(I_i, a) = I_j, then set Action[i,a] to shift j (a must be a terminal).
3. The goto transitions for state i are constructed for all non-terminals A using the rule: If successor(I_i, A) = I_j, then Goto [i, A] = j.
4. All entries not defined by rules 2 and 3 are errors.
5. The initial state is the one constructed from the configurating set containing S' -> •S.

In the SLR(1) parser, it is allowable for there to be both shift and reduce items in the same state as well as multiple reduce items. The SLR(1) parser will be able to determine which action to take as long as the follow sets are disjoint.

Let's consider those changes at the end of the LR(0) handout to the simplified expression grammar that would have made it no longer LR(0). Here is the version with the addition of array access:

```
E' -> E
E  -> E + T  | T
T  -> (E) | id | id[E]
```

Here are the first two LR(0) configurating sets entered if id is the first token of the input.

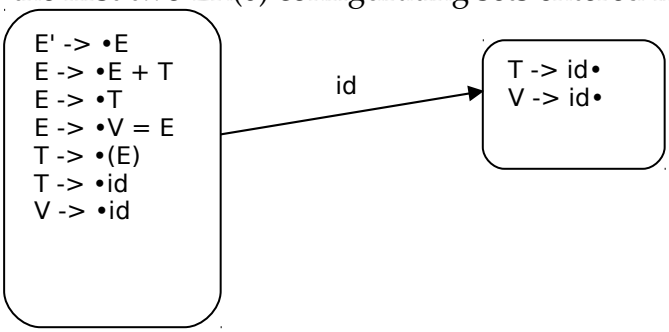


In an LR(0) parser, the set on the right has a shift-reduce conflict. However, an SLR(1) will compute Follow(T) = { +)] $ } and only enter the reduce action on those tokens. The input [will shift and there is no conflict. Thus this grammar is SLR(1) even though it is not LR(0).

Similarly, the simplified expression grammar with the assignment addition:

```
E' -> E
E  -> E + T  | T | V = E
T  -> (E) | id
V  -> id
```

Here are the first two LR(0) configurating sets entered if id is the first token of the input.



In an LR(0) parser, the set on the right has a reduce-reduce conflict. However, an SLR(1) parser will compute Follow(T) = { +) $ } and Follow(V) = { = } and thus can distinguish which reduction to apply depending on the next input token. The modified grammar is SLR(1).

SLR(1) Grammars

A grammar is SLR(1) if the following two conditions hold for each configurating set:

1. For any item A -> u•xv in the set, with terminal x, there is no complete item B -> w• in that set with x in Follow(B). In the tables, this translates no shift-reduce conflict on any state. This means the successor function for x from that set either shifts to a new state or reduces, but not both.
2. For any two complete items A -> u• and B -> v• in the set, the follow sets must be disjoint, e.g. Follow(A) ∩ Follow(B) is empty. This translates to no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which using only one token of lookahead.

All LR(0) grammars are SLR(1) but the reverse is not true, as the two extensions to our expression grammar demonstrated. The addition of just one token of lookahead and use of the follow set greatly expands the class of grammars that can be parsed without conflict.

SLR(1) Limitations

The SLR(1) technique still leaves something to be desired, because we are not using all the information that we have at our disposal. When we have a completed configuration (i.e., dot at the end) such as X -> u•, we know that this corresponds to a situation in which we have u as a handle on top of the stack which we then can reduce, i.e., replacing u by X. We allow such a reduction whenever the next symbol is in Follow(X). However, it may be that we should not reduce for every symbol in Follow(X), because the symbols below u on the stack preclude u being a handle for reduction in this case. In other words, SLR(1) states only tell us about the sequence on top of the stack, not what is below it on the stack. We may need to divide an SLR(1) state into separate states to differentiate the possible means by which that sequence has appeared on the stack. By carrying more information in the state, it will allow us to rule out these invalid reductions.Consider this example from Aho/Sethi/Ullman that defines a small grammar for assignment statements, using the non-terminal L for l-value and R for r-value and * for contents-of.

```
S' -> S
S -> L = R
S -> R
L -> *R
L -> id
R -> L
```

I_0:
S' -> •S
S -> •L = R
S -> •R
L -> •*R
L -> •id
R -> •L

I_1:
S' -> S•

I_2:
S -> L• = R
R -> L•

I_3:
S -> R•

I_4:
L -> *•R
R -> •L
L -> •*R
L -> •id

I_5:
L -> id•

I_6:
S -> L =•R
R -> •L
L -> •*R
L -> •id

I_7:
L -> *R•

I_8:
R -> L•

I_9:
S -> L = R•

Consider parsing the expression id = id. After working our way to configurating set I_2 having reduced the first id to L, we have a choice upon seeing = coming up in the input. The first item in the set wants to set Action[2,=] be shift 6, which corresponds to moving

on to find the rest of the assignment. However, = is also in Follow(R) because S => L=R => *R = R. Thus, the second configuration wants to reduce in that slot R->L. This is a shift-reduce conflict but not because of any problem with the grammar. A SLR parser does not remember enough left context to decide what should happen when it encounters a = in the input having seen a string reducible to L. Although the sequence on top of the stack could be reduced to R, we don't want to choose this reduction because there is no possible right sentential form that begins R = ... (there is one beginning *R = ... which is not the same). Thus, the correct choice is to shift.

It's not further lookahead that the SLR tables are missing—we don't need to see additional symbols beyond the first token in the input, we have already seen the information that allows us to determine the correct choice. What we need is to retain a little more of the left context that brought us here. In this example grammar, the only time we should consider reducing by production R->L is during a derivation that has already seen a * or an =. Just using the entire follow set is not discriminating enough as the guide for when to reduce. The follow set contains symbols that can follow R in any position within a valid sentence but it does not precisely indicate which symbols follow R at this particular point in a derivation. So we will augment our states to include information about what portion of the follow set is appropriate given the path we have taken to that state.

We can be in state 2 for one of two reasons, we are trying to build from S -> L = R or from S -> R -> L. If the upcoming symbol is =, then that rules out the second choice and we must be building the first, which tells us to shift. The reduction should only be applied if the next input symbol is $. Even though = is Follow(R) because of the other contexts that an R can appear, in this particular situation, it is not appropriate because when deriving a sentence S -> R -> L, = cannot follow R.

Constructing LR(1) parsing tables

LR or *canonical LR* parsing incorporates the required extra information into the state by redefining configurations to include a terminal symbol as an added component. LR(1) configurations have the general form:

$$A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j \ , a$$

This means we have states corresponding to $X_1 \ldots X_i$ on the stack and we are looking to put states corresponding to $X_{i+1} \ldots X_j$ on the stack and then reduce, but only if the token following X_j is the terminal a. a is called the *lookahead* of the configuration. The lookahead only comes into play with LR(1) configurations with a dot at the right end:

$A \rightarrow X_1 \ldots X_j \bullet, a$

This means we have states corresponding to $X_1 \ldots X_j$ on the stack but we may only reduce when the next symbol is a. The symbol a is either a terminal or $ (end of input marker). With SLR(1) parsing, we would reduce if the next token was any of those in Follow(A). With LR(1) parsing, we reduce only if the next token is exactly a. We may have more than one symbol in the lookahead for the configuration, as a convenience, we list those symbols separated by a forward slash. Thus, the configuration $A \rightarrow \underline{u}\bullet, a/b/c$ says that it is valid to reduce $\underline{u}$ to A only if the next token is equal to a, b, or c. The configuration lookahead will always be a subset of Follow(A).

Recall the definition of a *viable prefix* from the previous handout. Viable prefixes are those prefixes of right sentential forms that can appear on the stack of a shift-reduce parser. Formally we say that a configuration $[A \rightarrow \underline{u}\bullet\underline{v}, a]$ is valid for a viable prefix α if there is a rightmost derivation $S \Rightarrow^* \beta A\underline{w} \Rightarrow^* \beta\underline{uvw}$ where $\alpha = \beta\underline{u}$ and either a is the first symbol of $\underline{w}$ or $\underline{w}$ is ∂ and a is $.

For example:

$S \rightarrow ZZ$
$Z \rightarrow xZ \mid y$

There is a rightmost derivation $S \Rightarrow^* xxZxy \Rightarrow xxxZxy$. We see that configuration $[Z \rightarrow \underline{x}\bullet Z, x]$ is valid for viable prefix $\alpha = xxx$ by letting $\beta = xx$, $A = Z$, $\underline{w} = xy$, $\underline{u} = x$ and $\underline{v} = Z$. Another example is from the rightmost derivation $S \Rightarrow^* ZxZ \Rightarrow ZxxZ$, making $[Z \rightarrow \underline{x}\bullet Z, \$]$ valid for viable prefix Zxx.

Often we have a number of LR(1) configurations that differ only in their lookahead components. The addition of a lookahead component to LR(1) configurations allows us to make parsing decisions beyond the capability of SLR(1) parsers. There is, however, a big price to be paid. There will be more distinct configurations and thus many more possible configurating sets. This increases the size of the goto and action tables considerably. In the past when memory was smaller, it was difficult to find storage-efficient ways of representing these tables, but now this is not as much of an issue. Still, it's a big job building LR tables for any substantial grammar by hand.

The method for constructing the configurating sets of LR(1) configurations is essentially the same as for SLR, but there are some changes in the closure and successor operations because we must respect the configuration lookahead. To compute the closure of an LR(1) configurating set I:

Repeat the following until no more configurations can be added to state I:
— For each configuration [A -> u•Bv, a] in I, for each production B -> w in G', and for each terminal b in First(va) such that [B -> •w, b] is not in I: add [B -> •w, b] to I.

What does this mean? We have a configuration with the dot before the non-terminal B. In LR(0), we computed the closure by adding all B productions with no indication of what was expected to follow them. In LR(1), we are a little more precise— we add each B production but insist that each have a lookahead of va. The lookahead will be First(va) since this is what follows B in this production. Remember that we can compute first sets not just for a single non-terminal, but also a sequence of terminal and non-terminals. First(va) includes the first set of the first symbol of v and then if that symbol is nullable, we include the first set of the following symbol, and so on. If the entire sequence v is nullable, we add the lookahead a already required by this configuration.

The successor function for the configurating set I and symbol X is computed as this:

> Let J be the configurating set [A -> uX•v, a] such that [A -> u•Xv, a] is in I.
> successor(I,X) is the closure of configurating set J.

We take each production in a configurating set, move the dot over a symbol and close on the resulting production. This is basically the same successor function as defined for LR(0), but we have to propagate the lookahead when computing the transitions.
We construct the complete family of all configurating sets F just as we did before. F is initialized to the set with the closure of [S' -> S, $]. For each configurating set I and each grammar symbol X such that successor(I,X) is not empty and not in F, add successor (I,X) to F until no other configurating set can be added to F.

Let's consider an example. The augmented grammar below that recognizes the regular language a*ba*b (this example from pp. 231-236 Aho/Sethi/Ullman).

```
0) S' -> S
1) S -> XX
2) X -> aX
3) X  -> b
```

Here is the family of LR configuration sets:

I_0: S' –> •S, $
S –> •XX, $
X –> •aX, a/b
X –> •b, a/b

I_1: S' –> S•, $

I_2: S –> X•X, $
X –> •aX, $
X –> •b, $

I_3: X –> a•X, a/b
X –> •aX, a/b
X –> •b, a/b

I_4: X –> b•, a/b

I_5: S –> XX•, $

I_6: X –> a•X, $
X –> •aX, $
X –> •b, $

I_7: X –> b•, $

I_8: X –> aX•, a/b

I_9: X –> aX•, $

The above grammar would only have seven SLR states, but has ten in canonical LR. We end up with additional states because we have split states that have different lookaheads. For example, states 3 and 6 are the same except for lookahead, state 3 corresponds to the context where we are in the middle of parsing the first X, state 6 is the second X. Similarly, states 4 and 7 are completing the first and second X respectively. In SLR, those states are not distinguished, and if we were attempting to parse a single b by itself, we would allow that to be reduced to X, even though this will not lead to a valid sentence. The SLR parser will eventually notice the syntax error, too, but the LR parser figures it out a bit sooner.

To fill in the entries in the action and goto tables, we use a similar algorithm as we did for SLR(1), but instead of assigning reduce actions using the follow set, we use the specific lookaheads. Here are the steps to build an LR(1) parse table:

1. Construct F = {I_0, I_1, ... I_n}, the collection of configurating sets for the augmented grammar G' (augmented by adding the special production S' –> S).
2. State i is determined from I_i. The parsing actions for the state are determined as follows:
   a) If [A –> u•, a] is in I_i then set Action[i,a] to reduce A –> u (A is not S').
   b) If [S' –> S•, $] is in I_i then set Action[i,$] to accept.
   c) If [A –> u•av, b] is in I_i and succ(I_i, a) = I_j, then set Action[i,a] to shift j (a must be a terminal).
3. The goto transitions for state i are constructed for all non-terminals A using the rule: If succ(I_i, A) = I_j, then Goto [i, A] = j.
4. All entries not defined by rules 2 and 3 are errors.

The initial state is the one constructed from the configurating set containing S' -> •S. Following the algorithm using the configurating sets given above, we construct this canonical LR parse table:

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		

Let's parse an example string baab. It is a valid sentence in this language as shown by this leftmost derivation:

```
S ->  XX
      bX
      baX
      baaX
      baab
```

STACK STACK	REMAINING INPUT	PARSER ACTION
S_0	baab$	Shift S_4
S_0S_4	aab$	Reduce 3) X -> b, goto S_2
S_0S_2	aab$	Shift S_6
$S_0S_2S_6$	ab$	Shift S_6
$S_0S_2S_6S_6$	b$	Shift S_7
$S_0S_2S_6S_6S_7$	$	Reduce 3) X -> b, goto S_9
$S_0S_2S_6S_6S_9$	$	Reduce 2) X -> aX, goto S_9
$S_0S_2S_6S_9$	$	Reduce 2) X -> aX, goto S_5
$S_0S_2S_5$	$	Reduce 1) S -> XX, goto S_1
S_0S_1	$	Accept

Now, let's consider what the states mean. S_4 is where X -> b is completed; S_2 and S_6 is where we are in the middle of processing the 2 a's; S_7 is where we process the final b; S_9

is where we complete the X -> aX production; S_5 is where we complete S -> XX; and S_1 is where we accept.

LR(1) grammars

Every SLR(1) grammar is a canonical LR(1) grammar, but the canonical LR(1) parser may have more states than the SLR(1) parser. An LR(1) grammar is not necessarily SLR(1), the grammar given earlier is an example. Because an LR(1) parser splits states based on differing lookaheads, it may avoid conflicts that would otherwise result if using the full follow set.

A grammar is LR(1) if the following two conditions are satisfied for each configurating set:

1. For any item in the set [A -> u•xv, a] with x a terminal, there is no item in the set of the form [B -> v•, x]. In the action table, this translates no shift-reduce conflict for any state. The successor function for x either shifts to a new state or reduces, but not both.
2. The lookaheads for all complete items within the set must be disjoint, e.g. set cannot have both [A -> u•, a] and [B -> v•, a] This translates to no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which is appropriate from the next input token.

As long as there is a unique shift or reduce action on each input symbol from each state, we can parse using an LR(1) algorithm. The above state conditions are similar to what is required for SLR(1), but rather than the looser constraint about disjoint follow sets and so on, canonical LR(1) computes a more precise notion of the appropriate lookahead within a particular context and thus is able to resolve conflicts that SLR(1) would encounter.

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.