SLR and LR(1) Parsing

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 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 final item, this dictates that each final 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.

- 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.

SLR(1) Grammars

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

1. For any item A –> $\underline{u}$•x$\underline{v}$ in the set, with terminal x, there is no complete item B –> $\underline{w}$• 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 –> $\underline{u}$• and B –> $\underline{v}$• in the set, the follow sets must be disjoint, that is Follow(A) (intersection) Follow(B) must be 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 –> $\underline{u}$•, we know that this corresponds to a situation in which we have $\underline{u}$ as a handle on top of the stack which we then can reduce, i.e., replacing $\underline{u}$ 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 $\underline{u}$ on the stack preclude $\underline{u}$ 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 the example below 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	I_5:	L –> id•
	S –> •L = R		
	S –> •R	I_6:	S –> L =•R
	L –> •*R		R –> •L
	L –> •id		L –> •*R
	R –> •L		L –> •id
I_1:	S' –> S•	I_7:	L –> *R•
I_2:	S –> L• = R	I_8:	R –> L•
	R –> L•	I_9:	S –> L = R•
I_3:	S –> R•		
I_4:	L –> *•R		
	R –> •L		
	L –> •*R		
	L -> •id		

- 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:

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

This means we have states corresponding to $X_1...X_i$ on the stack and we are looking to put states corresponding to $X_{i+1}...X_j$ 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:

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

This means we have states corresponding to $X_1...X_j$ 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 or in curly braces. Thus, the configuration :

$$\textbf{A} \rightarrow \underline{\textbf{u}}\bullet, \textbf{a/b/c}$$
$$\textbf{or}$$
$$\textbf{A -> u, \{a,b,c\}}$$

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).

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.

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.