# A COMPARISON BY EXAMPLE OF LR(1) WITH SLR(1) PARSERS

Robert Heckendorn
University of Idaho

Let's look in detail at the example from page 215 in the book:

```
<stmt> -> <call-stmt> | <assign-stmt>
<call-stmt> -> identifier
<assign-stmt> -> <var> := <exp>
<var> -> <var> [ <exp> ] | identifier
<exp> -> <var> | number
```

A stand-in for the grammar above is:

```
S -> id | V = E
V -> id
E -> V | n
```

where the capital letters are nonterminals.
Let's create an SLR(1) parser for this grammar.
This is easy. We use the LR(0) items and build
a NFA then DFA and code from that as follows:

consider the following augmented and split out grammar:

```
0) S'-> S            // augmented grammar
1) S -> id
2) S -> V = E
3) V -> id
4) E -> V
5) E -> n
```

That leads to these LR(0) items:

```
0) S' -> . S
1) S' -> S .
2) S -> . id
3) S -> id .
4) S -> . V = E
5) S -> V . = E
6) S -> V = . E
7) S -> V = E .
8) V -> . id
9) V -> id .
10) E -> . V
11) E -> V .
12) E -> . n
13) E -> n .
```

we can now group these together into a NFA. Here are the transitions between the 14 states above. Note that dot before a terminal yeild a transition on a terminal and a dot before a nonterminal leads to an epsilon transition for each possible substitution for the nonterminal as well as a substitution for on each completed nonterminal. Progress through the NFA goes from start symbol to a "complete item".

```
0 -> 1  on  S
2 -> 3  on  id
4 -> 5  on  V
5 -> 6  on  =
6 -> 7  on  E
8 -> 9  on  id
10 -> 1  on 1 V
12 -> 1  on 3 n
0 -> 2  on  \epsilon
0 -> 4  on  \epsilon
4 -> 8  on  \epsilon
6 -> 1  on 0 \epsilon
6 -> 1  on 2 \epsilon
10 -> 8  on  \epsilon
```

We now convert the NFA to a DFA by starting with the start symbol:

```
    STATES                TRANSITIONS
0) S' -> . S              0->1 on S
0) S -> . id              0->2 on V
0) S -> .V = E            0->3 on id
0) V -> . id

1) S' -> S .

2) S -> V . = E           2->4 on =

3) S -> id .
3) V -> id .

4) S -> V = . E           4->5 on E
4) E -> . V               4->3 on id
4) E -> . n               4->6 on n
4) V -> . id              4->7 on V

5) S -> V = E .

6) E -> n .
```

7) E -> V .

We can now write the code for the parser.

1) for every noncomplete item (dot in the middle) we expect one
of the symbols following the dot to be the next token. That
is the lookahead that will tell us what state to go to next.

2) for every complete item (dot at the end) we expect to
simply pop off the matching tokens on the stack. Look at
the state and element from the lhs and goto that new state.
For example in state 5 above we will pop the three tokens V = E
from the stack and look at the revealed new-state on the top
of the stack. Then take the lhs of the production in state 5
which is S and the new-state and get the next state.

This works for most things above but something is funny with state 3.
We pop off the id giving us the new state but what is the lhs we use?!
Is it S or V?? We don't know which reduce to use. This is a
reduce-reduce error. (Compare this with the dangling else which gives
us a shift-reduce.)

Let's try a different approach. Let's include the next token in
the state structure and see if that helps. It might because
we are including this from the beginning.

consider the grammar again:
```
0) S'-> S $           // augmented grammar with end-of-input token
1) S -> id
2) S -> V = E
3) V -> id
4) E -> V
5) E -> n
```

Now let's create LR(1) items (see page 217).
LR(1) items are LR(0) items with a single lookahead token tag.
THis token will be the next token we scan in after the complete
pattern is matched. That is, for a production A -> .B the
LR(1) items are [A -> . B, z] where z is in the follow(A).

and guide the parse.
We tag each production depending on where the dot is.

1) If we have a production with a dot before an X

[a -> b . X c, z]

where X is any symbol then we preserve the lookahead token z

[a -> b . X c, z]  transitions to [a -> b X . c, z] on X

X can be a terminal or nonterminal.

2) If we have

[a -> b . X c, z]

where X is a nonterminal we also have an \epsilon transition

[a -> b . X c, z]  transitions to [ X -> Y, z' ]

for every z' in the First(cz) that is the first set of
the string of token c concatenated to the token z.  This
is First(c) if c can never be empty.

Note: for every
[a -> b . X , z]  transitions to [ X -> Y, z ] exactly as a special case
of rule 2.

We can construct the LR(1) items given the above rules and
the firstOfList function we described in class:

0) S' -> . S,$
1) S' -> S .,$
2) S -> . id,$
3) S -> id .,$
4) S -> .V = E,$
5) S -> V . = E,$
6) S -> V = . E,$
7) S -> V = E .,$
8) V -> . id,$
9) V -> . id,=
10) V -> id .,=
11) V -> id .,$
12) E -> . V,$
13) E -> V .,$
14) E -> . n,$
15) E -> n .,$

the transitions lead us to this DFA (see page 223)

| STATES | TRANSITIONS |
|---|---|
| 0) S' -> . S,$ | 0->1 on S |
| 0) S -> . id,$ | 0->2 on V |
| 0) S -> .V = E,$ | 0->3 on id |
| 0) V -> . id,= | |

```
1) S' -> S .,$

2) S -> id .,$
2) V -> id .,=

3) S -> V . = E,$          3->4 on =

4) S -> V = . E,$          4->5 on E
4) E -> . V,$              4->8 on id
4) E -> . n,$              4->7 on n
4) V -> . id,$             4->6 on V

5) S -> V = E .,$

6) E -> V .,$

7) E -> n .,$

8) V -> id .,$
```

Note that it is very clear what to do at every state and
when there is a reduce-reduce conflict from the SLR we
get a two choices of productions that are decided by the
lookahead char.  So the LR parser saved the day.

Why this works is that in the case of reducing an id to an S the S
must be followed by a $ while a V must be followed by an =.  These two
sets do not overlap.  In an LR(1) parser a reduce-reduce conflict must
now look something more like the following in the same set:

```
k) S -> id .,$
k) S -> id .,=
k) V -> id .,*
k) V -> id .,=
```

When the lookahead is = we can't decide between the two productions.

The important thing to see is that what can follow a V
fol(V) = {=, $}
and
fol{S} = {$}
overlap as you can see below:

Here are the S cases:

```
2) S -> . id,$
3) S -> id .,$
4) S -> .V = E,$
```

```
5) S -> V . = E,$
6) S -> V = . E,$
7) S -> V = E .,$

Here are the V cases:
8) V -> . id,$
9) V -> . id,=
10) V -> id .,=
11) V -> id .,$

In an SLR grammar the reductions 3 and 10 occur in the same
node (node 2) and which reduction to use would be
dictated by the rule in the SLR algorithm:

"if state contains a complete items  A ::= X.,  B::=Y., C::=Z. then
   chose A if input \isin Follow(A), B if input \isin Follow(B), etc"

Follow(S) = {$}
Follow(V) = {=, $}

This is a reduce/reduce error since the follow sets are not disjoint.

In an LR(1) grammar we start with three different nodes in the NFA

[S -> id., $]
[V -> id., $]
[V -> id., =]

and in the DFA become two nodes.  see page 223 for complete example:

node 2:
[S -> id., $]
[V -> id., =]

node 8:
[V -> id., $]

In node 2 the lookahead symbol tells us which reduction to use.

Let's look at this from a different view.  Where can
an id occur?

S -> id | V = E
V -> id
E -> V | n

it can occur in three places.   A statement can be:
```

```
id $
id = E $
V = id $

Look at figure 5.8:

id $          state 2 with lookahead $
id = E $      state 2 with lookahead =
V = id $      state 8

distinguishing between
id $
V = id $

used to be a problem since in an SLR grammar
state 4 used to go to state 2 on an id.
Now there is a new separate state 8 for id.
This is for processing the id at the end of
an assignment.

This would lump

[S -> id., $]
[V -> id., $]
[V -> id., =]

together and cause a confusion about what to do.  An extra state
allows us to record that we are at the end of an assignment so you
must want to do V -> id and not S -> id.
```